大奥特曼打小怪兽

第三十六节，目标检测之yolo源码解析

目录

在一个月前，我就已经介绍了yolo目标检测的原理，后来也把tensorflow实现代码仔细看了一遍。但是由于这个暑假事情比较大，就一直搁浅了下来，趁今天有时间，就把源码解析一下。关于yolo目标检测的原理请参考前面一篇文章：第三十五节，目标检测之YOLO算法详解。

回到顶部

公告

昵称： 大奥特曼打小怪兽
园龄： 1年6个月
粉丝： 248
关注： 9
+加关注

< 2019年8月 >

日	一	二	三	四	五	六
28	29	30	31	1	2	3
4	5	6	7	8	**9**	10
11	**12**	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类

Java基本语法(14)
OpenCV(20)
python(3)
Spring MVC(15)
tensorflow(22)
theano使用(4)
大数据(3)
机器学习(3)
精典博客系列收藏(6)
面试题(1)
深度学习(37)
信号处理(1)

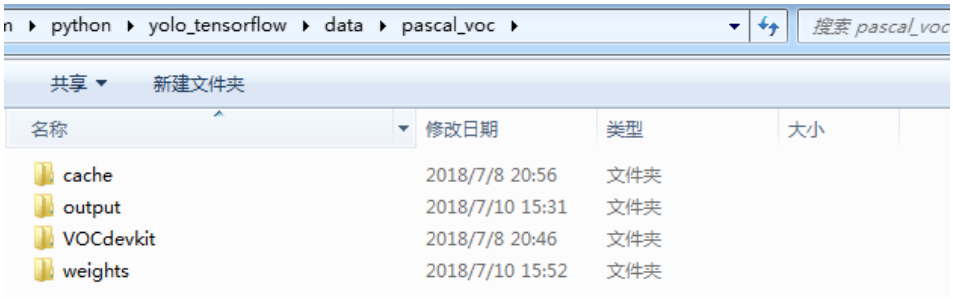
随笔档案

2019年8月(3)
2019年6月(1)
2019年5月(13)

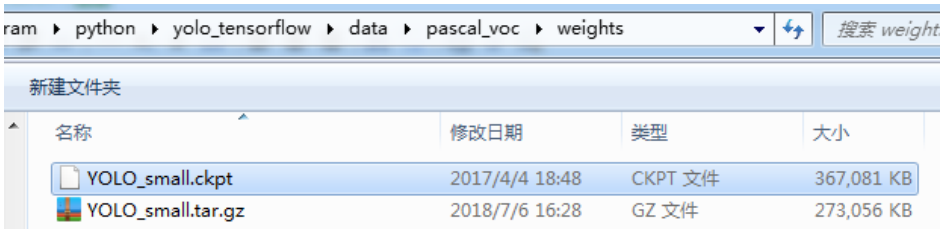
一 准备工作

在讲解源码之前，我们需要做一些准备工作：

1. 下载源码，本文所使用的yolo源码来源于网址：
   https://github.com/hizhangp/yolo_tensorflow
2. 下载训练所使用的数据集，我们仍然使用以VOC 2012数据集为例，下载地址为：
   http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCtrainval_11-May-2012.tar。
3. yolo源码所在目录下，创建一个目录data,然后在data里面创建一个pascal_voc目录，用来保存与VOC 2012数据集相关的数据，我们把下载好的数据集解压到该目录下，如下图所示，其中VOCdevkit为数据集解压得到的文件，剩余三个文件夹我们先不用理会，后面会详细介绍



4. 下载与训练模型，即YOLO_small文件，我们把下载好之后的文件解压放在weights文件夹下面。下载链接：https://drive.google.com/file/d/0B5aC8pI-akZUNVFZMmhmcVRpbTA/view?usp=sharing，需要翻墙才能下载，可以使用自-由-门翻墙软件。

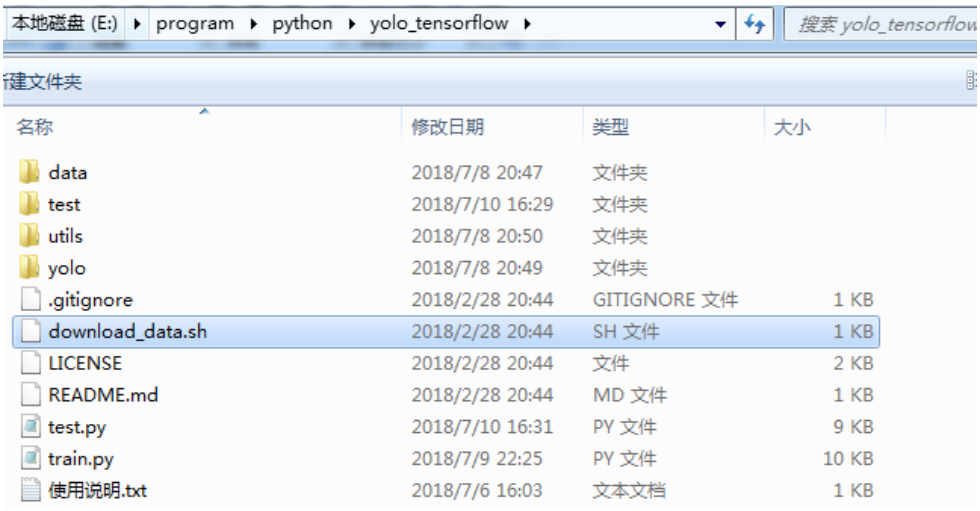


5. 根据自己的需求修改配置文件yolo/config.py。
6. 运行train.py文件，开始训练。
7. 运行test.py文件，开始测试。

回到顶部

二 yolo代码文件结构

如果你按照上面我说的步骤配置好文件之后，源代码结构就会如下图所示：



我们来粗略的介绍一下每个文件的功能：

- data文件夹，上面已经说过了，存放数据集以及训练时生成的模型，缓存文件。
- test文件夹，用来存放测试时用到的图片。
- utils文件夹，包含两个文件一个是pascal_voc.py，主要用来获取训练集图片文件，以及生成对应的标签文件，为yolo网络训练做准备。另一个文件是timer.py用来计时。

2019年4月(9)
2019年3月(6)
2019年1月(1)
2018年10月(2)
2018年9月(6)
2018年8月(11)
2018年7月(14)
2018年6月(9)
2018年5月(16)
2018年4月(23)
2018年3月(14)

最新评论

1. Re:第三十二节，使用谷歌Object Detection API进行目标检测、训练新的模型(使用VOC 2012数据集)
@ ylylyyyy应该是版本问题，你到github上看看官方文档吧...
--大奥特曼打小怪兽
2. Re:第三十二节，使用谷歌Object Detection API进行目标检测、训练新的模型(使用VOC 2012数据集)
按照您给的下载链接下载的object_detection，但是object_detection目录下根本没有train.py文件，我不知道是不是新旧版本的原因，麻烦博主看一下是什么原因。
--ylylyyyy
3. Re:第三十七节、人脸检测MTCNN和人脸识别Facenet(附源码)
你好，我想问一下为什么我的src/train_softmax.py运行速度这么慢，而且数据和您的相差那么大Epoch: [1][503/1000] Time 8.238 Loss 9.701 Xent...
--simple6x
4. Re:第六节、双目视觉之相机标定
博主真是牛B,感觉样样精通，从程序开发到人工智能图像识别。
--海大富
5. Re:第十九节，去噪自编码和栈式自编码
博主，我刚刚了解tensorflow，想问一下为什么第二层的去噪效果还不如第一的呀？
--Sir蓝天

阅读排行榜

1. 第二十一节，使用TensorFlow实现LSTM和GRU网络(16733)
2. 第三十二节，使用谷歌Object Detection API进行目标检测、训练新的模型(使用VOC 2012数据集)(16542)
3. 第三十七节、人脸检测MTCNN和人脸识别Facenet(附源码)(13327)
4. 第三十三节，目标检测之选择性搜索-Selective Search(12514)
5. 第六节、双目视觉之相机标定(11472)

评论排行榜

- yolo文件夹，也包含两个文件，config.py包含yolo网络的配置参数，yolo_net.py文件包含yolo网络的结构。
- train.py文件用来训练yolo网络。
- test.py文件用来测试yolo网络。

回到顶部

三 config.py文件讲解

我们先从配置文件说起，代码如下：

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 12 12:08:15 2018

@author: lenovo
"""

'''
配置参数
'''

import os


#
# 数据集路径，和模型检查点文件路径
#

DATA_PATH = 'data'            #所有数据所在的根目录

PASCAL_PATH = os.path.join(DATA_PATH, 'pascal_voc')    #VOC2012数据集所在的目录

CACHE_PATH = os.path.join(PASCAL_PATH, 'cache')        #保存生成的数据集标签缓冲文件所在文件夹

OUTPUT_DIR = os.path.join(PASCAL_PATH, 'output')       #保存生成的网络模型和日志文件所在的文件夹

WEIGHTS_DIR = os.path.join(PASCAL_PATH, 'weights')     #检查点文件所在的目录

#WEIGHTS_FILE = None

WEIGHTS_FILE = os.path.join(WEIGHTS_DIR, 'YOLO_small.ckpt')

#VOC 2012数据集类别名
CLASSES = ['aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus',
           'car', 'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse',
           'motorbike', 'person', 'pottedplant', 'sheep', 'sofa',
           'train', 'tvmonitor']
#使用水平镜像，扩大一倍数据集？
FLIPPED = True

'''
网络模型参数
'''

#图片大小
IMAGE_SIZE = 448

#单元格大小S   一共有CELL_SIZExCELL_SIZE个单元格
CELL_SIZE = 7

#每个单元格边界框的个数B
BOXES_PER_CELL = 2
#泄露修正线性激活函数 系数
ALPHA = 0.1
#控制台输出信息
DISP_CONSOLE = False

#损失函数 的权重设置
OBJECT_SCALE = 1.0    #有目标时，置信度权重
NOOBJECT_SCALE = 1.0 #没有目标时，置信度权重
CLASS_SCALE = 2.0     #类别权重
```


推荐排行榜

1. 第一节，TensorFlow基本用法(4)
2. 第二十一节，使用TensorFlow实现LSTM和GRU网络(3)
3. 第十二节、尺度不变特征(SIFT)(3)
4. 第七节、双目视觉之空间坐标计算(3)
5. 第三十三节，目标检测之选择性搜索-Selective Search(2)

```
COORD_SCALE = 5.0     #边界框权重



'''
训练参数设置
'''

GPU = ''
#学习率
LEARNING_RATE = 0.0001
#退化学习率衰减步数
DECAY_STEPS = 30000
#衰减率
DECAY_RATE = 0.1
STAIRCASE = True
#批量大小
BATCH_SIZE = 45
#最大迭代次数
MAX_ITER = 15000
#日志文件保存间隔步
SUMMARY_ITER = 10
#模型保存间隔步
SAVE_ITER = 500


'''
测试时的相关参数
'''
#格子有目标的置信度阈值
THRESHOLD = 0.2
#非极大值抑制 IoU阈值
IOU_THRESHOLD = 0.5
```

各个参数我已经在上面注释了，下面就不在重复了。下面我们来介绍yolo网络的构建。

回到顶部

四 yolo.py文件讲解

yolo网络的建立是通过yolo文件夹中的yolo_net.py文件的代码实现的，yolo_net.py文件定义了YOLONet类，该类包含了网路初始化(__init__())，建立网络(build_networks)和loss函数(loss_layer())等方法。

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jun 12 12:08:15 2018

@author: lenovo
"""

'''
定义YOLO网络模型
https://blog.csdn.net/qq_34784753/article/details/78803423
https://blog.csdn.net/qq1483661204/article/details/79681926
'''

import numpy as np
import tensorflow as tf
import yolo.config as cfg
import logging
import sys

slim = tf.contrib.slim

#配置logging
logging.basicConfig(format='%(asctime)s %(levelname)s %(message)s',
                    level=logging.INFO,
                    stream=sys.stdout)

class YOLONet(object):
```

1、网络参数初始化

网络的所有初始化参数包含于__init__()方法中。

```
    def __init__(self, is_training=True):
        '''
        构造函数
        利用 cfg 文件对网络参数进行初始化，同时定义网络的输入和输出 size 等信息，
        其中 offset 的作用应该是一个定长的偏移
        boundery1和boundery2 作用是在输出中确定每种信息的长度（如类别，置信度等）。
        其中 boundery1 指的是对于所有的 cell 的类别的预测的张量维度，所以是 self.cell_size *
self.cell_size * self.num_class
        boundery2 指的是在类别之后每个cell 所对应的 bounding boxes 的数量的总和，所以是
self.boundary1 + self.cell_size * self.cell_size * self.boxes_per_cell

        args:
            is_training：训练？
        '''
        #VOC 2012数据集类别名
        self.classes = cfg.CLASSES
        #类别个数C 20
        self.num_class = len(self.classes)
        #网络输入图像大小448， 448 x 448
        self.image_size = cfg.IMAGE_SIZE
        #单元格大小S=7  将图像分为SxS的格子
        self.cell_size = cfg.CELL_SIZE
        #每个网格边界框的个数B=2
        self.boxes_per_cell = cfg.BOXES_PER_CELL
        #网络输出的大小 S*S*(B*5 + C) = 1470
        self.output_size = (self.cell_size * self.cell_size) *\
            (self.num_class + self.boxes_per_cell * 5)
        #图片的缩放比例 64
        self.scale = 1.0 * self.image_size / self.cell_size
        '''#将网络输出分离为类别和置信度以及边界框的大小，输出维度为7*7*20 + 7*7*2 +
7*7*2*4=1470'''
        #7*7*20
        self.boundary1 = self.cell_size * self.cell_size * self.num_class
        #7*7*20+7*7*2
        self.boundary2 = self.boundary1 +\
            self.cell_size * self.cell_size * self.boxes_per_cell

        #代价函数 权重
        self.object_scale = cfg.OBJECT_SCALE  #1
        self.noobject_scale = cfg.NOOBJECT_SCALE  #1
        self.class_scale = cfg.CLASS_SCALE  #2.0
        self.coord_scale = cfg.COORD_SCALE  #5.0

        #学习率0.0001
        self.learning_rate = cfg.LEARNING_RATE
        #批大小 45
        self.batch_size = cfg.BATCH_SIZE
        #泄露修正线性激活函数 系数0.1
        self.alpha = cfg.ALPHA

        #偏置 形状[7,7,2]
        self.offset = np.transpose(np.reshape(np.array(
            [np.arange(self.cell_size)] * self.cell_size * self.boxes_per_cell),
            (self.boxes_per_cell, self.cell_size, self.cell_size)), (1, 2, 0))

        #输入图片占位符 [NONE,image_size,image_size,3]
        self.images = tf.placeholder(
            tf.float32, [None, self.image_size, self.image_size, 3],
            name='images')
        #构建网络 获取YOLO网络的输出(不经过激活函数的输出)  形状[None,1470]
        self.logits = self.build_network(
            self.images, num_outputs=self.output_size, alpha=self.alpha,
            is_training=is_training)

        if is_training:
            #设置标签占位符 [None,S,S,5+C]  即[None,7,7,25]
            self.labels = tf.placeholder(
                tf.float32,
                [None, self.cell_size, self.cell_size, 5 + self.num_class])
            #设置损失函数
            self.loss_layer(self.logits, self.labels)
```

```
            #加入权重正则化之后的损失函数
            self.total_loss = tf.losses.get_total_loss()
            #将损失以标量形式显示，该变量命名为total_loss
            tf.summary.scalar('total_loss', self.total_loss)
```

2、构建网络

网络的建立是通过build_network()函数实现的，网络由卷积层，池化层和全连接层组成，网络的输入维度是[None,448,448,3]，输出维度为[None,1470]。

```
def build_network(self,
                      images,
                      num_outputs,
                      alpha,
                      keep_prob=0.5,
                      is_training=True,
                      scope='yolo'):
        '''
        构建YOLO网络

        args:
            images：输入图片占位符 [None,image_size,image_size,3]  这里是[None,448,448,3]
            num_outputs：标量，网络输出节点数 1470
            alpha：泄露修正线性激活函数 系数0.1
            keep_prob：弃权 保留率
            is_training：训练？
            scope：命名空间名

        return：
            返回网络最后一层，激活函数处理之前的值  形状[None,1470]
        '''
        #定义变量命名空间
        with tf.variable_scope(scope):
            #定义共享参数  使用l2正则化
            with slim.arg_scope(
                [slim.conv2d, slim.fully_connected],
                activation_fn=leaky_relu(alpha),
                weights_regularizer=slim.l2_regularizer(0.0005),
                weights_initializer=tf.truncated_normal_initializer(0.0, 0.01)
            ):
                logging.info('image shape{0}'.format(images.shape))
                #pad_1 填充 454x454x3
                net = tf.pad(
                    images, np.array([[0, 0], [3, 3], [3, 3], [0, 0]]),
                    name='pad_1')
                logging.info('Layer pad_1  {0}'.format(net.shape))
                #卷积层conv_2 s=2    (n-f+1)/s向上取整    224x224x64
                net = slim.conv2d(
                    net, 64, 7, 2, padding='VALID', scope='conv_2')
                logging.info('Layer conv_2 {0}'.format(net.shape))
                #池化层pool_3 112x112x64
                net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_3')
                logging.info('Layer pool_3 {0}'.format(net.shape))
                #卷积层conv_4、3x3x192 s=1  n/s向上取整   112x112x192
                net = slim.conv2d(net, 192, 3, scope='conv_4')
                logging.info('Layer conv_4 {0}'.format(net.shape))
                #池化层pool_5 56x56x192
                net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_5')
                logging.info('Layer pool_5 {0}'.format(net.shape))
                #卷积层conv_6、1x1x128 s=1  n/s向上取整  56x56x128
                net = slim.conv2d(net, 128, 1, scope='conv_6')
                logging.info('Layer conv_6 {0}'.format(net.shape))
                #卷积层conv_7、3x3x256 s=1  n/s向上取整 56x56x256
                net = slim.conv2d(net, 256, 3, scope='conv_7')
                logging.info('Layer conv_7 {0}'.format(net.shape))
                #卷积层conv_8、1x1x256 s=1  n/s向上取整 56x56x256
                net = slim.conv2d(net, 256, 1, scope='conv_8')
                logging.info('Layer conv_8 {0}'.format(net.shape))
                #卷积层conv_9、3x3x512 s=1  n/s向上取整 56x56x512
                net = slim.conv2d(net, 512, 3, scope='conv_9')
                logging.info('Layer conv_9 {0}'.format(net.shape))
                #池化层pool_10 28x28x512
                net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_10')
```

```
        logging.info('Layer pool_10 {0}'.format(net.shape))
        #卷积层conv_11、1x1x256 s=1  n/s向上取整 28x28x256
        net = slim.conv2d(net, 256, 1, scope='conv_11')
        logging.info('Layer conv_11 {0}'.format(net.shape))
        #卷积层conv_12、3x3x512 s=1  n/s向上取整 28x28x512
        net = slim.conv2d(net, 512, 3, scope='conv_12')
        logging.info('Layer conv_12 {0}'.format(net.shape))
        #卷积层conv_13、1x1x256 s=1  n/s向上取整 28x28x256
        net = slim.conv2d(net, 256, 1, scope='conv_13')
        logging.info('Layer conv_13 {0}'.format(net.shape))
        #卷积层conv_14、3x3x512 s=1  n/s向上取整 28x28x512
        net = slim.conv2d(net, 512, 3, scope='conv_14')
        logging.info('Layer conv_14 {0}'.format(net.shape))
        #卷积层conv_15、1x1x256 s=1  n/s向上取整 28x28x256
        net = slim.conv2d(net, 256, 1, scope='conv_15')
        logging.info('Layer conv_15 {0}'.format(net.shape))
        #卷积层conv_16、3x3x512 s=1  n/s向上取整 28x28x512
        net = slim.conv2d(net, 512, 3, scope='conv_16')
        logging.info('Layer conv_16 {0}'.format(net.shape))
        #卷积层conv_17、1x1x256 s=1  n/s向上取整 28x28x256
        net = slim.conv2d(net, 256, 1, scope='conv_17')
        logging.info('Layer conv_17 {0}'.format(net.shape))
        #卷积层conv_18、3x3x512 s=1  n/s向上取整 28x28x512
        net = slim.conv2d(net, 512, 3, scope='conv_18')
        logging.info('Layer conv_18 {0}'.format(net.shape))
        #卷积层conv_19、1x1x512 s=1  n/s向上取整 28x28x512
        net = slim.conv2d(net, 512, 1, scope='conv_19')
        logging.info('Layer conv_19 {0}'.format(net.shape))
        #卷积层conv_20、3x3x1024 s=1  n/s向上取整 28x28x1024
        net = slim.conv2d(net, 1024, 3, scope='conv_20')
        logging.info('Layer conv_20 {0}'.format(net.shape))
        #池化层pool_21 14x14x1024
        net = slim.max_pool2d(net, 2, padding='SAME', scope='pool_21')
        logging.info('Layer pool_21 {0}'.format(net.shape))
        #卷积层conv_22、1x1x512 s=1  n/s向上取整 14x14x512
        net = slim.conv2d(net, 512, 1, scope='conv_22')
        logging.info('Layer conv_22 {0}'.format(net.shape))
        #卷积层conv_23、3x3x1024 s=1  n/s向上取整 14x14x1024
        net = slim.conv2d(net, 1024, 3, scope='conv_23')
        logging.info('Layer conv_23 {0}'.format(net.shape))
        #卷积层conv_24、1x1x512 s=1  n/s向上取整 14x14x512
        net = slim.conv2d(net, 512, 1, scope='conv_24')
        logging.info('Layer conv_24 {0}'.format(net.shape))
        #卷积层conv_25、3x3x1024 s=1  n/s向上取整 14x14x1024
        net = slim.conv2d(net, 1024, 3, scope='conv_25')
        logging.info('Layer conv_25 {0}'.format(net.shape))
        #卷积层conv_26、3x3x1024 s=1  n/s向上取整 14x14x1024
        net = slim.conv2d(net, 1024, 3, scope='conv_26')
        logging.info('Layer conv_26 {0}'.format(net.shape))
        #pad_27 填充 16x16x2014
        net = tf.pad(
            net, np.array([[0, 0], [1, 1], [1, 1], [0, 0]]),
            name='pad_27')
        logging.info('Layer pad_27 {0}'.format(net.shape))
        #卷积层conv_28、3x3x1024 s=2  (n-f+1)/s向上取整 7x7x1024
        net = slim.conv2d(
            net, 1024, 3, 2, padding='VALID', scope='conv_28')
        logging.info('Layer conv_28 {0}'.format(net.shape))
        #卷积层conv_29、3x3x1024 s=1  n/s向上取整 7x7x1024
        net = slim.conv2d(net, 1024, 3, scope='conv_29')
        logging.info('Layer conv_29 {0}'.format(net.shape))
        #卷积层conv_30、3x3x1024 s=1  n/s向上取整 7x7x1024
        net = slim.conv2d(net, 1024, 3, scope='conv_30')
        logging.info('Layer conv_30 {0}'.format(net.shape))
        #trans_31 转置[None,1024,7,7]
        net = tf.transpose(net, [0, 3, 1, 2], name='trans_31')
        logging.info('Layer trans_31 {0}'.format(net.shape))
        #flat_32 展开 50176
        net = slim.flatten(net, scope='flat_32')
        logging.info('Layer flat_32 {0}'.format(net.shape))
        #全连接层fc_33  512
        net = slim.fully_connected(net, 512, scope='fc_33')
        logging.info('Layer fc_33 {0}'.format(net.shape))
        #全连接层fc_34  4096
        net = slim.fully_connected(net, 4096, scope='fc_34')
        logging.info('Layer fc_34 {0}'.format(net.shape))
        #弃权层dropout_35 4096
```

```
            net = slim.dropout(
                net, keep_prob=keep_prob, is_training=is_training,
                scope='dropout_35')
            logging.info('Layer dropout_35 {0}'.format(net.shape))
            #全连接层fc_36 1470
            net = slim.fully_connected(
                net, num_outputs, activation_fn=None, scope='fc_36')
            logging.info('Layer fc_36 {0}'.format(net.shape))
        return net
```

3、代价函数

代价函数是通过loss_layer()实现的，在代码中，我们优化以下多部分损失函数：

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

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

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

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

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

```
def loss_layer(self, predicts, labels, scope='loss_layer'):
        '''
        计算预测和标签之间的损失函数

        args：
            predicts：Yolo网络的输出 形状[None,1470]
                      0：7*7*20：表示预测类别
                      7*7*20:7*7*20 + 7*7*2:表示预测置信度，即预测的边界框与实际边界框之间的IOU
                      7*7*20 + 7*7*2：1470：预测边界框    目标中心是相对于当前格子的，宽度和高度的开根号是相对当前整张图像的(归一化的)
            labels：标签值 形状[None,7,7,25]
                      0:1：置信度，表示这个地方是否有目标
                      1:5：目标边界框   目标中心，宽度和高度(没有归一化)
                      5:25：目标的类别
        '''
        with tf.variable_scope(scope):
            '''#将网络输出分离为类别和置信度以及边界框的大小，输出维度为7*7*20 + 7*7*2 + 7*7*2*4=1470'''
            #预测每个格子目标的类别 形状[45,7,7,20]
            predict_classes = tf.reshape(
                predicts[:, :self.boundary1],
                [self.batch_size, self.cell_size, self.cell_size, self.num_class])
            #预测每个格子中两个边界框的置信度 形状[45,7,7,2]
            predict_scales = tf.reshape(
                predicts[:, self.boundary1:self.boundary2],
                [self.batch_size, self.cell_size, self.cell_size, self.boxes_per_cell])
            #预测每个格子中的两个边界框，(x,y)表示边界框相对于格子边界框的中心 w,h的开根号相对于整个图片  形状[45,7,7,2,4]
            predict_boxes = tf.reshape(
                predicts[:, self.boundary2:],
                [self.batch_size, self.cell_size, self.cell_size, self.boxes_per_cell, 4])

            #标签的置信度,表示这个地方是否有框 形状[45,7,7,1]
            response = tf.reshape(
                labels[..., 0],
                [self.batch_size, self.cell_size, self.cell_size, 1])
            #标签的边界框 (x,y)表示边界框相对于整个图片的中心 形状[45,7,7,1，4]
            boxes = tf.reshape(
                labels[..., 1:5],
                [self.batch_size, self.cell_size, self.cell_size, 1, 4])
            #标签的边界框 归一化后 张量沿着axis=3重复两边，扩充后[45,7,7,2,4]
            boxes = tf.tile(
                boxes, [1, 1, 1, self.boxes_per_cell, 1]) / self.image_size
```

```
            classes = labels[..., 5:]

            '''
            predict_boxes_tran：offset变量用于把预测边界框predict_boxes中的坐标中心(x,y)由相对当前格子转换为相对当前整个图片

            offset，这个是构造的[7,7,2]矩阵，每一行都是[7,2]的矩阵，值为[[0,0],[1,1],[2,2],[3,3],[4,4],[5,5],[6,6]]
            这个变量是为了将每个cell的坐标对齐，后一个框比前一个框要多加1
            比如我们预测了cell_size的每个中心点坐标，那么我们这个中心点落在第几个cell_size
            就对应坐标要加几，这个用法比较巧妙，构造了这样一个数组，让他们对应位置相加
            '''
            #offset shape为[1,7,7,2]  如果忽略axis=0，则每一行都是  [[0,0],[1,1],[2,2],[3,3],[4,4],[5,5],[6,6]]
            offset = tf.reshape(
                tf.constant(self.offset, dtype=tf.float32),
                [1, self.cell_size, self.cell_size, self.boxes_per_cell])
            #shape为[45,7,7,2]
            offset = tf.tile(offset, [self.batch_size, 1, 1, 1])
            #shape为[45,7,7,2]  如果忽略axis=0 第i行为[[i,i],[i,i],[i,i],[i,i],[i,i],[i,i],[i,i]]
            offset_tran = tf.transpose(offset, (0, 2, 1, 3))
            #shape为[45,7,7,2,4]  计算每个格子中的预测边界框坐标(x,y)相对于整个图片的位置  而不是相对当前格子
            #假设当前格子为(3,3)，当前格子的预测边界框为(x0,y0)，则计算坐标(x,y) = ((x0,y0)+(3,3))/7
            predict_boxes_tran = tf.stack(
                [(predict_boxes[..., 0] + offset) / self.cell_size,         #x
                 (predict_boxes[..., 1] + offset_tran) / self.cell_size,    #y
                 tf.square(predict_boxes[..., 2]),                          #width
                 tf.square(predict_boxes[..., 3])], axis=-1)                #height

            #计算每个格子预测边界框与真实边界框之间的IOU  [45,7,7,2]
            iou_predict_truth = self.calc_iou(predict_boxes_tran, boxes)

            # calculate I tensor [BATCH_SIZE, CELL_SIZE, CELL_SIZE, BOXES_PER_CELL]
            #这个是求论文中的1ijobj参数，[45,7,7,2]     1ijobj：表示网格单元i的第j个编辑框预测器'负责'该预测
            #先计算每个框交并比最大的那个，因为我们知道，YOLO每个格子预测两个边界框，一个类别。在训练时，每个目标只需要
            #一个预测器来负责，我们指定一个预测器"负责"，根据哪个预测器与真实值之间具有当前最高的IOU来预测目标。
            #所以object_mask就表示每个格子中的哪个边界框负责该格子中目标预测？哪个边界框取值为1，哪个边界框就负责目标预测
            #当格子中的确有目标时，取值为[1,1]，[1,0],[0,1]
            #比如某一个格子的值为[1,0]，表示第一个边界框负责该格子目标的预测  [0,1]：表示第二个边界框负责该格子目标的预测
            #当格子没有目标时，取值为[0,0]
            object_mask = tf.reduce_max(iou_predict_truth, 3, keep_dims=True)
            object_mask = tf.cast(
                (iou_predict_truth >= object_mask), tf.float32) * response

            # calculate no_I tensor [CELL_SIZE, CELL_SIZE, BOXES_PER_CELL]
            # noobject_mask就表示每个边界框不负责该目标的置信度，
            #使用tf.onr_like，使得全部为1,再减去有目标的，也就是有目标的对应坐标为1,这样一减，就变为没有的了。[45,7,7,2]
            noobject_mask = tf.ones_like(
                object_mask, dtype=tf.float32) - object_mask

            # boxes_tran 这个就是把之前的坐标换回来(相对整个图像->相对当前格子)，长和宽开方(原因在论文中有说明)，后面求loss就方便。 shape为(4, 45, 7, 7, 2)
            boxes_tran = tf.stack(
                [boxes[..., 0] * self.cell_size - offset,
                 boxes[..., 1] * self.cell_size - offset_tran,
                 tf.sqrt(boxes[..., 2]),
                 tf.sqrt(boxes[..., 3])], axis=-1)

            #class_loss 分类损失，如果目标出现在网格中 response为1，否则response为0  原文代价函数公式第5项
            #该项表名当格子中有目标时，预测的类别越接近实际类别，代价值越小  原文代价函数公式第5项
            class_delta = response * (predict_classes - classes)
            class_loss = tf.reduce_mean(
                tf.reduce_sum(tf.square(class_delta), axis=[1, 2, 3]),
                name='class_loss') * self.class_scale

            # object_loss 有目标物体存在的置信度预测损失    原文代价函数公式第3项
            #该项表名当格子中有目标时，负责该目标预测的边界框的置信度越越接近预测的边界框与实际边界框之
```

```
间的IOU时，代价值越小
            object_delta = object_mask * (predict_scales - iou_predict_truth)
            object_loss = tf.reduce_mean(
                tf.reduce_sum(tf.square(object_delta), axis=[1, 2, 3]),
                name='object_loss') * self.object_scale

            #noobject_loss  没有目标物体存在的置信度的损失(此时iou_predict_truth为0)  原文代价
函数公式第4项
            #该项表名当格子中没有目标时，预测的两个边界框的置信度越接近0，代价值越小
            noobject_delta = noobject_mask * predict_scales
            noobject_loss = tf.reduce_mean(
                tf.reduce_sum(tf.square(noobject_delta), axis=[1, 2, 3]),
                name='noobject_loss') * self.noobject_scale

            # coord_loss 边界框坐标损失 shape 为 [batch_size, 7, 7, 2, 1]  原文代价函数公式
1,2项
            #该项表名当格子中有目标时，预测的边界框越接近实际边界框，代价值越小
            coord_mask = tf.expand_dims(object_mask, 4)                 #1ij
            boxes_delta = coord_mask * (predict_boxes - boxes_tran)
            coord_loss = tf.reduce_mean(
                tf.reduce_sum(tf.square(boxes_delta), axis=[1, 2, 3, 4]),
                name='coord_loss') * self.coord_scale

            #将所有损失放在一起
            tf.losses.add_loss(class_loss)
            tf.losses.add_loss(object_loss)
            tf.losses.add_loss(noobject_loss)
            tf.losses.add_loss(coord_loss)

            # 将每个损失添加到日志记录
            tf.summary.scalar('class_loss', class_loss)
            tf.summary.scalar('object_loss', object_loss)
            tf.summary.scalar('noobject_loss', noobject_loss)
            tf.summary.scalar('coord_loss', coord_loss)

            tf.summary.histogram('boxes_delta_x', boxes_delta[..., 0])
            tf.summary.histogram('boxes_delta_y', boxes_delta[..., 1])
            tf.summary.histogram('boxes_delta_w', boxes_delta[..., 2])
            tf.summary.histogram('boxes_delta_h', boxes_delta[..., 3])
            tf.summary.histogram('iou', iou_predict_truth)
```

在计算代价函数时，我们需要计算预测的box与实际边界框之间的IOU。其代码如下：

```
    def calc_iou(self, boxes1, boxes2, scope='iou'):
        """calculate ious
        这个函数的主要作用是计算两个 bounding box 之间的 IoU。输入是两个 5 维的bounding box,输出
的两个 bounding Box 的IoU

        Args:
          boxes1: 5-D tensor [BATCH_SIZE, CELL_SIZE, CELL_SIZE, BOXES_PER_CELL, 4]
====> (x_center, y_center, w, h)
          boxes2: 5-D tensor [BATCH_SIZE, CELL_SIZE, CELL_SIZE, BOXES_PER_CELL, 4] ===>
(x_center, y_center, w, h)
          注意这里的参数x_center, y_center, w, h都是归一到[0,1]之间的，分别表示预测边界框的中心相
对整张图片的坐标，宽和高
        Return:
          iou: 4-D tensor [BATCH_SIZE, CELL_SIZE, CELL_SIZE, BOXES_PER_CELL]
        """
        with tf.variable_scope(scope):
            # transform (x_center, y_center, w, h) to (x1, y1, x2, y2)
            #把以前的中心点坐标和长和宽转换成了左上角和右下角的两个点的坐标
            boxes1_t = tf.stack([boxes1[..., 0] - boxes1[..., 2] / 2.0,   #左上角x
                                 boxes1[..., 1] - boxes1[..., 3] / 2.0,   #左上角y
                                 boxes1[..., 0] + boxes1[..., 2] / 2.0,   #右下角x
                                 boxes1[..., 1] + boxes1[..., 3] / 2.0],  #右下角y
                                axis=-1)

            boxes2_t = tf.stack([boxes2[..., 0] - boxes2[..., 2] / 2.0,
                                 boxes2[..., 1] - boxes2[..., 3] / 2.0,
                                 boxes2[..., 0] + boxes2[..., 2] / 2.0,
                                 boxes2[..., 1] + boxes2[..., 3] / 2.0],
                                axis=-1)
```

```
            # calculate the left up point & right down point
            #lu和rd就是分别求两个框相交的矩形的左上角的坐标和右下角的坐标，因为对于左上角，
            #选择的是x和y较大的，而右下角是选择较小的，可以想想两个矩形框相交是不是这中情况
            lu = tf.maximum(boxes1_t[..., :2], boxes2_t[..., :2])        #两个框相交的矩形的左上角(x1,y1)
            rd = tf.minimum(boxes1_t[..., 2:], boxes2_t[..., 2:])        #两个框相交的矩形的右下角(x2,y2)

            # intersection 这个就是求相交矩形的长和宽，所以有rd-ru，相当于x1-x2和y1-y2，
            #之所以外面还要加一个tf.maximum是因为删除那些不合理的框，比如两个框没交集，
            #就会出现左上角坐标比右下角还大。
            intersection = tf.maximum(0.0, rd - lu)
            #inter_square这个就是求面积了，就是长乘以宽。
            inter_square = intersection[..., 0] * intersection[..., 1]

            # calculate the boxs1 square and boxs2 square
            #square1和square2这个就是求面积了，因为之前是中心点坐标和长和宽，所以这里直接用长和宽
            square1 = boxes1[..., 2] * boxes1[..., 3]
            square2 = boxes2[..., 2] * boxes2[..., 3]

            #union_square就是就两个框的交面积，因为如果两个框的面积相加，那就会重复了相交的部分，
            #所以减去相交的部分，外面有个tf.maximum这个就是保证相交面积不为0,因为后面要做分母。
            union_square = tf.maximum(square1 + square2 - inter_square, 1e-10)

        #最后有一个tf.clip_by_value,这个是将如果你的交并比大于1,那么就让它等于1,如果小于0,那么就
        #让他变为0,因为交并比在0-1之间。
        return tf.clip_by_value(inter_square / union_square, 0.0, 1.0)
```

4、其他函数

漏泄修正线性激活：

```
#def leaky_relu(alpha):
    '''
    激活函数
    '''
    #def op(inputs):
     #    return tf.nn.leaky_relu(inputs, alpha=alpha, name='leaky_relu')
    #return op

def leaky_relu(alpha):
    def op(inputs):
        with tf.variable_scope('leaky_relu'):
         f1 = 0.5 * (1 + alpha)
         f2 = 0.5 * (1 - alpha)
         return f1 * inputs + f2 * tf.abs(inputs)
    return op
```

回到顶部

五 读取数据pascal_voc.py文件解析

我们在YOLENet类中定义了两个占位符，一个是输入图片占位符，一个是图片对应的标签占位符，如下：

```
        #输入图片占位符 [NONE,image_size,image_size,3]
        self.images = tf.placeholder(
            tf.float32, [None, self.image_size, self.image_size, 3],
            name='images')
        #设置标签占位符 [None,S,S,5+C]  即[None,7,7,25]
        self.labels = tf.placeholder(
            tf.float32,
            [None, self.cell_size, self.cell_size, 5 + self.num_class])
```

而pascal_voc.py文件的目的就是为了准备数据，赋值给占位符。在pascal_voc.py文件中定义了一个pascal_voc，该类包含了类初始化函数(__init__())，准备数据函数(prepare()),读取batch大小的图片以及图片对应的标签(get())等函数。

```
import os
import xml.etree.ElementTree as ET
import numpy as np
import cv2
import pickle
import copy
import yolo.config as cfg


'''
VOC2012数据集处理
'''

class pascal_voc(object):
```

1、类初始化函数

```
'''
VOC2012数据集处理
'''

class pascal_voc(object):
    '''
    VOC2012数据集处理的类，主要用来获取训练集图片文件，以及生成对应的标签文件
    '''
    def __init__(self, phase, rebuild=False):
        '''
        准备训练或者测试的数据

        args:
            phase：传入字符串 'train'：表示训练
                              'test'：测试
            rebuild：是否重新创建数据集的标签文件，保存在缓存文件夹下
        '''
        #VOCdevkit文件夹路径
        self.devkil_path = os.path.join(cfg.PASCAL_PATH, 'VOCdevkit')
        #VOC2012文件夹路径
        self.data_path = os.path.join(self.devkil_path, 'VOC2012')
        #catch文件所在路径
        self.cache_path = cfg.CACHE_PATH
        #批大小
        self.batch_size = cfg.BATCH_SIZE
        #图像大小
        self.image_size = cfg.IMAGE_SIZE
        #单元格大小S
        self.cell_size = cfg.CELL_SIZE
        #VOC 2012数据集类别名
        self.classes = cfg.CLASSES
        #类别名->索引的dict
        self.class_to_ind = dict(zip(self.classes, range(len(self.classes))))
        ##图片是否采用水平镜像扩充训练集？
        self.flipped = cfg.FLIPPED
        #训练或测试？
        self.phase = phase
        #是否重新创建数据集标签文件
        self.rebuild = rebuild
        #从gt_labels加载数据，cursor表明当前读取到第几个
        self.cursor = 0
        #存放当前训练的轮数
        self.epoch = 1
        #存放数据集的标签 是一个list 每一个元素都是一个dict，对应一个图片
        #如果我们在配置文件中指定flipped=True，则数据集会扩充一倍，每一张原始图片都有一个水平对称的镜像文件
        #       imname：图片路径
        #       label：图片标签
        #       flipped：图片水平镜像？
        self.gt_labels = None
        #加载数据集标签  初始化gt_labels
        self.prepare()
```

2、prepare()所有数据准备函数

prepare()函数调用load_labels()函数，加载所有数据集的标签，保存在遍历gt_labels集合中，如果在配置文件中指定了水平镜像，则追加一倍的训练数据集。

```
    def prepare(self):
        '''
        初始化数据集的标签，保存在变量gt_labels中

        return:
            gt_labels:返回数据集的标签 是一个list  每一个元素对应一张图片，是一个dict
                                    imname：图片文件路径
                                    label：图片文件对应的标签 [7,7,25]的矩阵
                                    flipped：是否使用水平镜像？ 设置为False
        '''
        #加载数据集的标签
        gt_labels = self.load_labels()
        #如果水平镜像，则追加一倍的训练数据集
        if self.flipped:
            print('Appending horizontally-flipped training examples ...')
            #深度拷贝
            gt_labels_cp = copy.deepcopy(gt_labels)
            #遍历每一个图片标签
            for idx in range(len(gt_labels_cp)):
                #设置flipped属性为True
                gt_labels_cp[idx]['flipped'] = True
                #目标所在格子也进行水平镜像 [7,7,25]
                gt_labels_cp[idx]['label'] =\
                    gt_labels_cp[idx]['label'][:, ::-1, :]
                for i in range(self.cell_size):
                    for j in range(self.cell_size):
                        #置信度==1，表示这个格子有目标
                        if gt_labels_cp[idx]['label'][i, j, 0] == 1:
                            #中心的x坐标水平镜像
                            gt_labels_cp[idx]['label'][i, j, 1] = \
                                self.image_size - 1 -\
                                gt_labels_cp[idx]['label'][i, j, 1]
            #追加数据集的标签    后面的是由原数据集标签扩充的水平镜像数据集标签
            gt_labels += gt_labels_cp
        #打乱数据集的标签
        np.random.shuffle(gt_labels)
        self.gt_labels = gt_labels
        return gt_labels
```

3、get()批量数据读取函数

get()函数用在训练的时候，每次从gt_labels集合随机读取batch大小的图片以及图片对应的标签。

```
    def get(self):
        '''
        加载数据集 每次读取batch大小的图片以及图片对应的标签

        return:
            images:读取到的图片数据 [45,448,448,3]
            labels:对应的图片标签 [45,7,7,25]
        '''
        #[45,448,448,3]
        images = np.zeros(
            (self.batch_size, self.image_size, self.image_size, 3))
        #[45,7,7,25]
        labels = np.zeros(
            (self.batch_size, self.cell_size, self.cell_size, 25))
        count = 0
        #一次加载batch_size个图片数据
        while count < self.batch_size:
            #获取图片路径
            imname = self.gt_labels[self.cursor]['imname']
            #是否使用水平镜像？
            flipped = self.gt_labels[self.cursor]['flipped']
            #读取图片数据
            images[count, :, :, :] = self.image_read(imname, flipped)
```

```
            #读取图片标签
            labels[count, :, :, :] = self.gt_labels[self.cursor]['label']
            count += 1
            self.cursor += 1
            #如果读取完一轮数据，则当前cursor置为0，当前训练轮数+1
            if self.cursor >= len(self.gt_labels):
                #打乱数据集
                np.random.shuffle(self.gt_labels)
                self.cursor = 0
                self.epoch += 1
        return images, labels
```

4、image_read()函数读取图片

图片读取函数，先读取图片，然后缩放，转换为RGB格式，再对数据进行归一化处理。

```
    def image_read(self, imname, flipped=False):
        '''
        读取图片

        args:
            imname：图片路径
            flipped：图片是否水平镜像处理？

        return：
            image：图片数据 [448,448,3]
        '''
        #读取图片数据
        image = cv2.imread(imname)
        #缩放处理
        image = cv2.resize(image, (self.image_size, self.image_size))
        #BGR->RGB  uint->float32
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB).astype(np.float32)
        #归一化处理 [-1.0,1.0]
        image = (image / 255.0) * 2.0 - 1.0
        #宽倒序  即水平镜像
        if flipped:
            image = image[:, ::-1, :]
        return image
```

5、load_labels()加载标签函数

```
    def load_labels(self):
        '''
        加载数据集标签

        return:
            gt_labels：是一个list  每一个元素对应一张图片，是一个dict
                                    imname：图片文件路径
                                    label：图片文件对应的标签 [7,7,25]的矩阵
                                    flipped：是否使用水平镜像？ 设置为False
        '''
        #缓冲文件名：即用来保存数据集标签的文件
        cache_file = os.path.join(
            self.cache_path, 'pascal_' + self.phase + '_gt_labels.pkl')

        #文件存在，且不重新创建则直接读取
        if os.path.isfile(cache_file) and not self.rebuild:
            print('Loading gt_labels from: ' + cache_file)
            with open(cache_file, 'rb') as f:
                gt_labels = pickle.load(f)
            return gt_labels

        print('Processing gt_labels from: ' + self.data_path)

        #如果缓冲文件目录不存在，创建
        if not os.path.exists(self.cache_path):
            os.makedirs(self.cache_path)

        #获取训练测试集的数据文件名
```

```
        if self.phase == 'train':
            txtname = os.path.join(
                self.data_path, 'ImageSets', 'Main', 'trainval.txt')
        #获取测试集的数据文件名
        else:
            txtname = os.path.join(
                self.data_path, 'ImageSets', 'Main', 'test.txt')
        with open(txtname, 'r') as f:
            self.image_index = [x.strip() for x in f.readlines()]

        #存放图片的标签，图片路径，是否使用水平镜像？
        gt_labels = []
        #遍历每一张图片的信息
        for index in self.image_index:
            #读取每一张图片的标签label [7,7,25]
            label, num = self.load_pascal_annotation(index)
            if num == 0:
                continue
            #图片文件路径
            imname = os.path.join(self.data_path, 'JPEGImages', index + '.jpg')
            #保存该图片的信息
            gt_labels.append({'imname': imname,
                              'label': label,
                              'flipped': False})
        print('Saving gt_labels to: ' + cache_file)
        #保存
        with open(cache_file, 'wb') as f:
            pickle.dump(gt_labels, f)
        return gt_labels
```

6、load_pascal_annotation()函数

```
    def load_pascal_annotation(self, index):
        """
        Load image and bounding boxes info from XML file in the PASCAL VOC
        format.

        args:
            index：图片文件的index

        return :
            label：标签 [7,7,25]
                      0:1：置信度，表示这个地方是否有目标
                      1:5：目标边界框  目标中心，宽度和高度(这里是实际值，没有归一化)
                      5:25：目标的类别
            len(objs)：objs对象长度
        """
        #获取图片文件名路径
        imname = os.path.join(self.data_path, 'JPEGImages', index + '.jpg')
        #读取数据
        im = cv2.imread(imname)
        #宽和高缩放比例
        h_ratio = 1.0 * self.image_size / im.shape[0]
        w_ratio = 1.0 * self.image_size / im.shape[1]
        # im = cv2.resize(im, [self.image_size, self.image_size])
        #用于保存图片文件的标签
        label = np.zeros((self.cell_size, self.cell_size, 25))
        #图片文件的标注xml文件
        filename = os.path.join(self.data_path, 'Annotations', index + '.xml')
        tree = ET.parse(filename)
        objs = tree.findall('object')

        for obj in objs:
            bbox = obj.find('bndbox')
            # Make pixel indexes 0-based  当图片缩放到image_size时，边界框也进行同比例缩放
            x1 = max(min((float(bbox.find('xmin').text) - 1) * w_ratio, self.image_size
- 1), 0)
            y1 = max(min((float(bbox.find('ymin').text) - 1) * h_ratio, self.image_size
- 1), 0)
            x2 = max(min((float(bbox.find('xmax').text) - 1) * w_ratio, self.image_size
- 1), 0)
            y2 = max(min((float(bbox.find('ymax').text) - 1) * h_ratio, self.image_size
- 1), 0)
```

```
            #根据图片的分类名 ->类别index 转换
            cls_ind = self.class_to_ind[obj.find('name').text.lower().strip()]
            #计算边框中心点x,y,w,h(没有归一化)
            boxes = [(x2 + x1) / 2.0, (y2 + y1) / 2.0, x2 - x1, y2 - y1]
            #计算当前物体的中心在哪个格子中
            x_ind = int(boxes[0] * self.cell_size / self.image_size)
            y_ind = int(boxes[1] * self.cell_size / self.image_size)
            #表明该图片已经初始化过了
            if label[y_ind, x_ind, 0] == 1:
                continue
            #置信度，表示这个地方有物体
            label[y_ind, x_ind, 0] = 1
            #物体边界框
            label[y_ind, x_ind, 1:5] = boxes
            #物体的类别
            label[y_ind, x_ind, 5 + cls_ind] = 1

        return label, len(objs)
```

回到顶部

六 训练网络

模型训练包含于train.py文件，Solver类的train()方法之中，训练部分只需要看懂了初始化参数，整个结构就很清晰了。

```
import os
import argparse
import datetime
import tensorflow as tf
import yolo.config as cfg
from yolo.yolo_net import YOLONet
from utils.timer import Timer
from utils.pascal_voc import pascal_voc

slim = tf.contrib.slim

'''
用来训练YOLO网络模型
'''

class Solver(object):
    '''
    求解器的类，用于训练YOLO网络
    '''
```

1、类初始化函数

```
   def __init__(self, net, data):
        '''
        构造函数，加载训练参数

        args：
            net：YOLONet对象
            data：pascal_voc对象
        '''
        #yolo网络
        self.net = net
        #voc2012数据处理
        self.data = data
        #检查点文件路径
        self.weights_file = cfg.WEIGHTS_FILE
        #训练最大迭代次数
        self.max_iter = cfg.MAX_ITER
        #初始学习率
        self.initial_learning_rate = cfg.LEARNING_RATE
        ##退化学习率衰减步数
        self.decay_steps = cfg.DECAY_STEPS
        #衰减率
        self.decay_rate = cfg.DECAY_RATE
```

```
        self.staircase = cfg.STAIRCASE
        ##日志文件保存间隔步
        self.summary_iter = cfg.SUMMARY_ITER
        ##模型保存间隔步
        self.save_iter = cfg.SAVE_ITER

        #输出文件夹路径
        self.output_dir = os.path.join(
            cfg.OUTPUT_DIR, datetime.datetime.now().strftime('%Y_%m_%d_%H_%M'))
        if not os.path.exists(self.output_dir):
            os.makedirs(self.output_dir)
        #保存配置信息
        self.save_cfg()
        #指定保存的张量 这里指定所有变量
        self.variable_to_restore = tf.global_variables()
        self.saver = tf.train.Saver(self.variable_to_restore, max_to_keep=None)
        #指定保存的模型名称
        self.ckpt_file = os.path.join(self.output_dir, 'yolo.cpkt')
        #合并所有的summary
        self.summary_op = tf.summary.merge_all()
        #创建writer，指定日志文件路径，用于写日志文件
        self.writer = tf.summary.FileWriter(self.output_dir, flush_secs=60)

        #创建变量，保存当前迭代次数
        self.global_step = tf.train.create_global_step()
        #退化学习率
        self.learning_rate = tf.train.exponential_decay(
            self.initial_learning_rate, self.global_step, self.decay_steps,
            self.decay_rate, self.staircase, name='learning_rate')
        #创建求解器
        self.optimizer = tf.train.GradientDescentOptimizer(
            learning_rate=self.learning_rate)
        # create_train_op that ensures that when we evaluate it to get the loss,
        # the update_ops are done and the gradient updates are computed.
        self.train_op = slim.learning.create_train_op(
            self.net.total_loss, self.optimizer, global_step=self.global_step)

        #设置GPU使用资源
        gpu_options = tf.GPUOptions()
        #按需分配GPU使用的资源
        config = tf.ConfigProto(gpu_options=gpu_options)
        self.sess = tf.Session(config=config)

        #运行图之前，初始化变量
        self.sess.run(tf.global_variables_initializer())

        #恢复模型
        if self.weights_file is not None:
            print('Restoring weights from: ' + self.weights_file)
            self.saver.restore(self.sess, self.weights_file)

        #将图写入日志文件
        self.writer.add_graph(self.sess.graph)
```

2、train()训练函数

```
    def train(self):
        '''
        开始训练
        '''
        #训练时间
        train_timer = Timer()
        #数据集加载时间
        load_timer = Timer()

        #开始迭代
        for step in range(1, self.max_iter + 1):
            #计算每次迭代加载数据的起始时间
            load_timer.tic()
            #加载数据集 每次读取batch大小的图片以及图片对应的标签
            images, labels = self.data.get()
            #计算这次迭代加载数据集所使用的时间
            load_timer.toc()
```

```
            feed_dict = {self.net.images: images,
                         self.net.labels: labels}

            #迭代summary_iter次，保存一次日志文件，迭代summary_iter*10次，输出一次的迭代信息
            if step % self.summary_iter == 0:
                if step % (self.summary_iter * 10) == 0:
                    #计算每次迭代训练的起始时间
                    train_timer.tic()
                    loss = 0.0001
                    #开始迭代训练，每一次迭代后global_step自加1
                    summary_str, loss, _ = self.sess.run(
                        [self.summary_op, self.net.total_loss, self.train_op],
                        feed_dict=feed_dict)
                    #输出信息
                    log_str = '{} Epoch: {}, Step: {}, Learning rate: {}, Loss:
{:5.3f}\nSpeed: {:.3f}s/iter,Load: {:.3f}s/iter, Remain: {}'.format(
                        datetime.datetime.now().strftime('%m-%d %H:%M:%S'),
                        self.data.epoch,
                        int(step),
                        round(self.learning_rate.eval(session=self.sess), 6),
                        loss,
                        train_timer.average_time,
                        load_timer.average_time,
                        train_timer.remain(step, self.max_iter))
                    print(log_str)

                else:
                    #计算每次迭代训练的起始时间
                    train_timer.tic()
                    #开始迭代训练，每一次迭代后global_step自加1
                    summary_str, _ = self.sess.run(
                        [self.summary_op, self.train_op],
                        feed_dict=feed_dict)
                    #计算这次迭代训练所使用的时间
                    train_timer.toc()

                #将summary写入文件
                self.writer.add_summary(summary_str, step)

            else:
                #计算每次迭代训练的起始时间
                train_timer.tic()
                #开始迭代训练，每一次迭代后global_step自加1
                self.sess.run(self.train_op, feed_dict=feed_dict)
                #计算这次迭代训练所使用的时间
                train_timer.toc()

            #没迭代save_iter次，保存一次模型
            if step % self.save_iter == 0:
                print('{} Saving checkpoint file to: {}'.format(
                    datetime.datetime.now().strftime('%m-%d %H:%M:%S'),
                    self.output_dir))
                self.saver.save(
                    self.sess, self.ckpt_file, global_step=self.global_step)
```

3、保存配置参数

```
    def save_cfg(self):
        '''
        保存配置信息
        '''
        with open(os.path.join(self.output_dir, 'config.txt'), 'w') as f:
            cfg_dict = cfg.__dict__
            for key in sorted(cfg_dict.keys()):
                if key[0].isupper():
                    cfg_str = '{}: {}\n'.format(key, cfg_dict[key])
                    f.write(cfg_str)
```

train.py文件除了上面介绍的求解器Solver这个类外，还包含了两个函数，一个是update_config_paths()函数，这个函数主要使用了设定数据集路径，以及检查点文件路径。

```
def update_config_paths(data_dir, weights_file):
    '''
    数据集路径，和模型检查点文件路径

    args:
        data_dir:数据文件夹   数据集放在pascal_voc目录下
        weights_file：检查点文件名  该文件放在数据集目录下的weights文件夹下

    '''
    cfg.DATA_PATH = data_dir                                                    #数据所在
文件夹
    cfg.PASCAL_PATH = os.path.join(data_dir, 'pascal_voc')                      #VOC2012
数据所在文件夹
    cfg.CACHE_PATH = os.path.join(cfg.PASCAL_PATH, 'cache')                     #保存生成
的数据集标签缓冲文件所在文件夹
    cfg.OUTPUT_DIR = os.path.join(cfg.PASCAL_PATH, 'output')                    #保存生成
的网络模型和日志文件所在的文件夹
    cfg.WEIGHTS_DIR = os.path.join(cfg.PASCAL_PATH, 'weights')                  #检查点文
件所在的目录

    cfg.WEIGHTS_FILE = os.path.join(cfg.WEIGHTS_DIR, weights_file)
```

我们主要来说一下另一个函数main()函数，先解析命令行参数，然后创建YOLONet、pascal_voc、Solver对象，最后开始训练。

```
def main():
    #创建一个解析器对象，并告诉它将会有些什么参数。当程序运行时，该解析器就可以用于处理命令行参数。
    #https://www.cnblogs.com/lovemyspring/p/3214598.html
    parser = argparse.ArgumentParser()
    #定义参数
    parser.add_argument('--weights', default="YOLO_small.ckpt", type=str)     #权重文件名
    parser.add_argument('--data_dir', default="data", type=str)               #数据集路径
    parser.add_argument('--threshold', default=0.2, type=float)
    parser.add_argument('--iou_threshold', default=0.5, type=float)
    parser.add_argument('--gpu', default='', type=str)
    #定义了所有参数之后，你就可以给 parse_args() 传递一组参数字符串来解析命令行。默认情况下，参数是从
sys.argv[1:] 中获取
    #parse_args() 的返回值是一个命名空间，包含传递给命令的参数。该对象将参数保存其属性
    args = parser.parse_args()

    #判断是否是使用gpu
    if args.gpu is not None:
        cfg.GPU = args.gpu

    #设定数据集路径，以及检查点文件路径
    if args.data_dir != cfg.DATA_PATH  and args.data_dir is not None:
        update_config_paths(args.data_dir, args.weights)

    #设置环境变量
    os.environ['CUDA_VISIBLE_DEVICES'] = cfg.GPU

    #创建YOLO网络对象
    yolo = YOLONet()
    #数据集对象
    pascal = pascal_voc('train')
    #求解器对象
    solver = Solver(yolo, pascal)

    print('Start training ...')
    #开始训练
    solver.train()
    print('Done training.')
```

我们执行如下代码，开始训练网络：

```
if __name__ == '__main__':
    tf.reset_default_graph()
    # python train.py --weights YOLO_small.ckpt --gpu 0
    main()
```

在控制台会输出以下信息：

```
In [31]: runfile('E:/program/python/yolo_tensorflow/train.py',
wdir='E:/program/python/yolo_tensorflow')
2018-08-25 17:11:21,668 INFO image shape(?, 448, 448, 3)
2018-08-25 17:11:21,673 INFO Layer pad_1  (?, 454, 454, 3)
2018-08-25 17:11:21,702 INFO Layer conv_2 (?, 224, 224, 64)
2018-08-25 17:11:21,706 INFO Layer pool_3 (?, 112, 112, 64)
2018-08-25 17:11:21,735 INFO Layer conv_4 (?, 112, 112, 192)
2018-08-25 17:11:21,738 INFO Layer pool_5 (?, 56, 56, 192)
2018-08-25 17:11:21,764 INFO Layer conv_6 (?, 56, 56, 128)
2018-08-25 17:11:21,790 INFO Layer conv_7 (?, 56, 56, 256)
2018-08-25 17:11:21,816 INFO Layer conv_8 (?, 56, 56, 256)
2018-08-25 17:11:21,842 INFO Layer conv_9 (?, 56, 56, 512)
2018-08-25 17:11:21,846 INFO Layer pool_10 (?, 28, 28, 512)
2018-08-25 17:11:21,871 INFO Layer conv_11 (?, 28, 28, 256)
2018-08-25 17:11:21,897 INFO Layer conv_12 (?, 28, 28, 512)
2018-08-25 17:11:21,921 INFO Layer conv_13 (?, 28, 28, 256)
2018-08-25 17:11:21,946 INFO Layer conv_14 (?, 28, 28, 512)
2018-08-25 17:11:21,975 INFO Layer conv_15 (?, 28, 28, 256)
2018-08-25 17:11:22,001 INFO Layer conv_16 (?, 28, 28, 512)
2018-08-25 17:11:22,030 INFO Layer conv_17 (?, 28, 28, 256)
2018-08-25 17:11:22,058 INFO Layer conv_18 (?, 28, 28, 512)
2018-08-25 17:11:22,085 INFO Layer conv_19 (?, 28, 28, 512)
2018-08-25 17:11:22,181 INFO Layer conv_20 (?, 28, 28, 1024)
2018-08-25 17:11:22,183 INFO Layer pool_21 (?, 14, 14, 1024)
2018-08-25 17:11:22,207 INFO Layer conv_22 (?, 14, 14, 512)
2018-08-25 17:11:22,232 INFO Layer conv_23 (?, 14, 14, 1024)
2018-08-25 17:11:22,257 INFO Layer conv_24 (?, 14, 14, 512)
2018-08-25 17:11:22,283 INFO Layer conv_25 (?, 14, 14, 1024)
2018-08-25 17:11:22,310 INFO Layer conv_26 (?, 14, 14, 1024)
2018-08-25 17:11:22,313 INFO Layer pad_27 (?, 16, 16, 1024)
2018-08-25 17:11:22,339 INFO Layer conv_28 (?, 7, 7, 1024)
2018-08-25 17:11:22,365 INFO Layer conv_29 (?, 7, 7, 1024)
2018-08-25 17:11:22,392 INFO Layer conv_30 (?, 7, 7, 1024)
2018-08-25 17:11:22,396 INFO Layer trans 31 (?, 1024, 7, 7)

2018-08-25 17:11:22,413 INFO Layer flat_32 (?, 50176)
2018-08-25 17:11:22,440 INFO Layer fc_33 (?, 512)
2018-08-25 17:11:22,462 INFO Layer fc_34 (?, 4096)
2018-08-25 17:11:22,473 INFO Layer dropout_35 (?, 4096)
2018-08-25 17:11:22,491 INFO Layer fc_36 (?, 1470)
Loading gt_labels from: data\pascal_voc\cache
\pascal_train_gt_labels.pkl
Appending horizontally-flipped training examples ...
Restoring weights from: data\pascal_voc\weights\YOLO_small.ckpt
INFO:tensorflow:Restoring parameters from data\pascal_voc\weights
\YOLO_small.ckpt
2018-08-25 17:11:29,075 INFO Restoring parameters from data
\pascal_voc\weights\YOLO_small.ckpt
Start training ...
```

回到顶部

七 测试网络

模型测试包含于test.py文件，Detector类的image_detector()函数用于检测目标。

```
import os
import cv2
import argparse
import numpy as np
import tensorflow as tf
import yolo.config as cfg
from yolo.yolo_net import YOLONet
from utils.timer import Timer

'''
用于测试
'''

class Detector(object):
```

1、类初始化函数

```
 def __init__(self, net, weight_file):
        '''
        构造函数
        利用 cfg 文件对网络参数进行初始化，
        其中 offset 的作用应该是一个定长的偏移
```

```
        boundery1和boundery2 作用是在输出中确定每种信息的长度（如类别，置信度等）。
        其中 boundery1 指的是对于所有的 cell 的类别的预测的张量维度，所以是 self.cell_size * self.cell_size * self.num_class
        boundery2 指的是在类别之后每个cell 所对应的 bounding boxes 的数量的总和，所以是 self.boundary1 + self.cell_size * self.cell_size * self.boxes_per_cell


        args：
            net：YOLONet对象
            weight_file：检查点文件路径
        '''
        #yolo网络
        self.net = net
        #检查点文件路径
        self.weights_file = weight_file
        #输出文件夹路径
        self.output_dir = os.path.dirname(self.weights_file)
         #VOC 2012数据集类别名
        self.classes = cfg.CLASSES
        # #VOC 2012数据类别数
        self.num_class = len(self.classes)
        ##图像大小
        self.image_size = cfg.IMAGE_SIZE
        #单元格大小S
        self.cell_size = cfg.CELL_SIZE
        #每个网格边界框的个数B=2
        self.boxes_per_cell = cfg.BOXES_PER_CELL
        #阈值参数
        self.threshold = cfg.THRESHOLD
        #IoU 阈值参数
        self.iou_threshold = cfg.IOU_THRESHOLD
        '''#将网络输出分离为类别和置信度以及边界框的大小，输出维度为7*7*20 + 7*7*2 + 7*7*2*4=1470'''
         #7*7*20
        self.boundary1 = self.cell_size * self.cell_size * self.num_class
         #7*7*20+7*7*2
        self.boundary2 = self.boundary1 +\
            self.cell_size * self.cell_size * self.boxes_per_cell

        #运行图之前，初始化变量
        self.sess = tf.Session()
        self.sess.run(tf.global_variables_initializer())

        #恢复模型
        print('Restoring weights from: ' + self.weights_file)
        self.saver = tf.train.Saver()
        #直接载入最近保存的检查点文件
        ckpt = tf.train.latest_checkpoint(self.output_dir)
        print("ckpt:",ckpt)
        #如果存在检查点文件 则恢复模型
        if ckpt!=None:
            #恢复最近的检查点文件
            self.saver.restore(self.sess, ckpt)
        else:
            #从指定检查点文件恢复
            self.saver.restore(self.sess, self.weights_file)
```

2、draw_result()函数

在原始图像上绘制边界框，并添加一些附件信息，如目标类别，置信度。

```
    def draw_result(self, img, result):
        '''
        在原图上绘制边界框，以及附加信息

        args:
            img：原始图片数据
            result:yolo网络目标检测到的边界框，list类型 每一个元素对应一个目标框
                  包含{类别名,x_center,y_center,w,h,置信度}
        '''
        #遍历每一个边界框
        for i in range(len(result)):
            #x_center
```

```
            x = int(result[i][1])
            #y_center
            y = int(result[i][2])
            #w/2
            w = int(result[i][3] / 2)
            #h/2
            h = int(result[i][4] / 2)
            #绘制矩形框(目标边界框) 矩形左上角，矩形右下角
            cv2.rectangle(img, (x - w, y - h), (x + w, y + h), (0, 255, 0), 2)
            #绘制矩形框，用于存放类别名称，使用灰度填充
            cv2.rectangle(img, (x - w, y - h - 20),
                          (x + w, y - h), (125, 125, 125), -1)
            #线型
            lineType = cv2.LINE_AA if cv2.__version__ > '3' else cv2.CV_AA
            #绘制文本信息 写上类别名和置信度
            cv2.putText(
                img, result[i][0] + ' : %.2f' % result[i][5],
                (x - w + 5, y - h - 7), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
                (0, 0, 0), 1, lineType)
```

3、detect()函数

detect()函数用来对图像进行目标检测。

```
def detect(self, img):
        '''
        图片目标检测

        args:
            img:原始图片数据

        return：
            result：返回检测到的边界框，list类型 每一个元素对应一个目标框
            包含{类别名,x_center,y_center,w,h,置信度}
        '''
        #获取图片的高和宽
        img_h, img_w, _ = img.shape
        #图片缩放 [448,448,3]
        inputs = cv2.resize(img, (self.image_size, self.image_size))
        #BGR->RGB  uint->float32
        inputs = cv2.cvtColor(inputs, cv2.COLOR_BGR2RGB).astype(np.float32)
        #归一化处理 [-1.0,1.0]
        inputs = (inputs / 255.0) * 2.0 - 1.0
        #reshape [1,448,448,3]
        inputs = np.reshape(inputs, (1, self.image_size, self.image_size, 3))

        #获取网络输出第一项(即第一张图片) [1,1470]
        result = self.detect_from_cvmat(inputs)[0]

        #对检测的图片的边界框进行缩放处理，一张图片可以有多个边界框
        for i in range(len(result)):
            #x_center, y_center, w, h都是真实值，分别表示预测边界框的中心坐标，宽和高，都是浮点型
            result[i][1] *= (1.0 * img_w / self.image_size)     #x_center
            result[i][2] *= (1.0 * img_h / self.image_size)     #y_center
            result[i][3] *= (1.0 * img_w / self.image_size)     #w
            result[i][4] *= (1.0 * img_h / self.image_size)     #h

        #<class 'list'> 6 ['person', 405.83171163286482, 161.40340532575334,
166.17623397282193, 298.85661533900668, 0.69636690616607666]
        #Average detecting time: 0.571s
        print(type(result),len(result),result[0])
        return result
```

4、detect_from_cvmat()函数

```
def detect_from_cvmat(self, inputs):
        '''
        运行yolo网络，开始检测
```

```
        args:
            inputs：输入数据  [None,448,448,3]

        return：
            results：返回目标检测的结果，每一个元素对应一个测试图片，每个元素包含着若干个边界框

        '''
        #返回网络最后一层，激活函数处理之前的值  形状[None,1470]
        net_output = self.sess.run(self.net.logits,
                                   feed_dict={self.net.images: inputs})
        results = []

        #对网络输出每一行数据进行处理
        for i in range(net_output.shape[0]):
            results.append(self.interpret_output(net_output[i]))

        #返回处理后的结果
        return results
```

5、interpret_output()函数

该函数对yolo网络输出的结果进行处理，提取出有目标的边界框，方便后续的处理。

```
 def interpret_output(self, output):
        '''
        对yolo网络输出进行处理

        args：
            output：yolo网络输出的每一行数据 大小为[1470,]
                    0：7*7*20：表示预测类别
                    7*7*20:7*7*20 + 7*7*2:表示预测置信度，即预测的边界框与实际边界框之间的IOU
                    7*7*20 + 7*7*2：1470：预测边界框      目标中心是相对于当前格子的，宽度和高度的开根号是相对当前整张图像的(归一化的)

        return:
             result:yolo网络目标检测到的边界框，list类型 每一个元素对应一个目标框
                  包含{类别名,x_center,y_center,w,h,置信度}    实际上这个置信度是yolo网络输出的置信度confidence和预测对应的类别概率的乘积
        '''
        #[7,7,2,20]
        probs = np.zeros((self.cell_size, self.cell_size,
                          self.boxes_per_cell, self.num_class))
        #类别概率 [7,7,20]
        class_probs = np.reshape(
            output[0:self.boundary1],
            (self.cell_size, self.cell_size, self.num_class))
        #置信度 [7,7,2]
        scales = np.reshape(
            output[self.boundary1:self.boundary2],
            (self.cell_size, self.cell_size, self.boxes_per_cell))
        #边界框 [7,7,2,4]
        boxes = np.reshape(
            output[self.boundary2:],
            (self.cell_size, self.cell_size, self.boxes_per_cell, 4))
        #[14,7]  每一行[0,1,2,3,4,5,6]
        offset = np.array(
            [np.arange(self.cell_size)] * self.cell_size * self.boxes_per_cell)
        #[7,7,2] 每一行都是  [[0,0],[1,1],[2,2],[3,3],[4,4],[5,5],[6,6]]
        offset = np.transpose(
            np.reshape(
                offset,
                [self.boxes_per_cell, self.cell_size, self.cell_size]),
            (1, 2, 0))

        #目标中心是相对于整个图片的
        boxes[:, :, :, 0] += offset
        boxes[:, :, :, 1] += np.transpose(offset, (1, 0, 2))
        boxes[:, :, :, :2] = 1.0 * boxes[:, :, :, 0:2] / self.cell_size
        #宽度、高度相对整个图片的
        boxes[:, :, :, 2:] = np.square(boxes[:, :, :, 2:])

        #转换成实际的编辑框(没有归一化的)
        boxes *= self.image_size
```

```
        #遍历每一个边界框的置信度
        for i in range(self.boxes_per_cell):
            #遍历每一个类别
            for j in range(self.num_class):
                #在测试时，乘以条件类概率和单个盒子的置信度预测，这些分数编码了j类出现在框i中的概率以及预测框拟合目标的程度。
                probs[:, :, i, j] = np.multiply(
                    class_probs[:, :, j], scales[:, :, i])

        #[7,7,2,20] 如果第i个边界框检测到类别j 则[:,:,i,j]=1
        filter_mat_probs = np.array(probs >= self.threshold, dtype='bool')
        #返回filter_mat_probs非0值的索引 返回4个List，每个list长度为n  即检测到的边界框的个数
        filter_mat_boxes = np.nonzero(filter_mat_probs)
        #获取检测到目标的边界框 [n,4]  n表示边界框的个数
        boxes_filtered = boxes[filter_mat_boxes[0],
                               filter_mat_boxes[1], filter_mat_boxes[2]]
        #获取检测到目标的边界框的置信度 (n,)
        probs_filtered = probs[filter_mat_probs]
        #获取检测到目标的边界框对应的目标类别 (n,)
        classes_num_filtered = np.argmax(
            filter_mat_probs, axis=3)[
            filter_mat_boxes[0], filter_mat_boxes[1], filter_mat_boxes[2]]
        #按置信度倒序排序，返回对应的索引
        argsort = np.array(np.argsort(probs_filtered))[::-1]
        boxes_filtered = boxes_filtered[argsort]
        probs_filtered = probs_filtered[argsort]
        classes_num_filtered = classes_num_filtered[argsort]

        for i in range(len(boxes_filtered)):
            if probs_filtered[i] == 0:
                continue
            for j in range(i + 1, len(boxes_filtered)):
                #计算n各边界框，两两之间的IoU是否大于阈值，非极大值抑制
                if self.iou(boxes_filtered[i], boxes_filtered[j]) :
                    probs_filtered[j] = 0.0

        #非极大值抑制后的输出
        filter_iou = np.array(probs_filtered > 0.0, dtype='bool')
        boxes_filtered = boxes_filtered[filter_iou]
        probs_filtered = probs_filtered[filter_iou]
        classes_num_filtered = classes_num_filtered[filter_iou]

        result = []
        #遍历每一个边界框
        for i in range(len(boxes_filtered)):
            result.append(
                [self.classes[classes_num_filtered[i]],  #类别名
                 boxes_filtered[i][0],                   #x中心
                 boxes_filtered[i][1],                   #y中心
                 boxes_filtered[i][2],                   #宽度
                 boxes_filtered[i][3],                   #高度
                 probs_filtered[i]])                     #置信度

        return result
```

6、iou()函数

计算两个边界框的IoU值。

```
    def iou(self, box1, box2):
        '''
        计算两个边界框的IoU

        args:
            box1：边界框1  [4,]   真实值
            box2：边界框2  [4,]   真实值
        '''
        tb = min(box1[0] + 0.5 * box1[2], box2[0] + 0.5 * box2[2]) - \
            max(box1[0] - 0.5 * box1[2], box2[0] - 0.5 * box2[2])
        lr = min(box1[1] + 0.5 * box1[3], box2[1] + 0.5 * box2[3]) - \
            max(box1[1] - 0.5 * box1[3], box2[1] - 0.5 * box2[3])
```

```
        inter = 0 if tb < 0 or lr < 0 else tb * lr
        return inter / (box1[2] * box1[3] + box2[2] * box2[3] - inter)
```

7、camera_detector()函数

调用摄像头实现实时目标检测。

```
    def camera_detector(self, cap, wait=10):
        '''
        打开摄像头，实时检测

        '''
        #测试时间
        detect_timer = Timer()
        #读取一帧
        ret, _ = cap.read()

        while ret:
            #读取一帧
            ret, frame = cap.read()
            #测试其实时间
            detect_timer.tic()
            result = self.detect(frame)
            #测试结束时间
            detect_timer.toc()
            print('Average detecting time: {:.3f}s'.format(
                detect_timer.average_time))
            #绘制边界框，以及添加附加信息
            self.draw_result(frame, result)
            #显示
            cv2.imshow('Camera', frame)
            cv2.waitKey(wait)
```

8、image_detector()函数

对图片进行目标检测。

```
    def image_detector(self, imname, wait=0):
        '''
        目标检测

        args：
            imname：测试图片路径
        '''
        #检测时间
        detect_timer = Timer()
        #读取图片
        image = cv2.imread(imname)
        #image = cv2.resize(image,(int(image.shape[1]/2),int(image.shape[0]/2)))
        #检测的起始时间
        detect_timer.tic()
        #开始检测
        result = self.detect(image)
        #检测的结束时间
        detect_timer.toc()
        print('Average detecting time: {:.3f}s'.format(
            detect_timer.average_time))
        #绘制检测结果
        self.draw_result(image, result)
        cv2.imshow('Image', image)
        cv2.waitKey(wait)
```

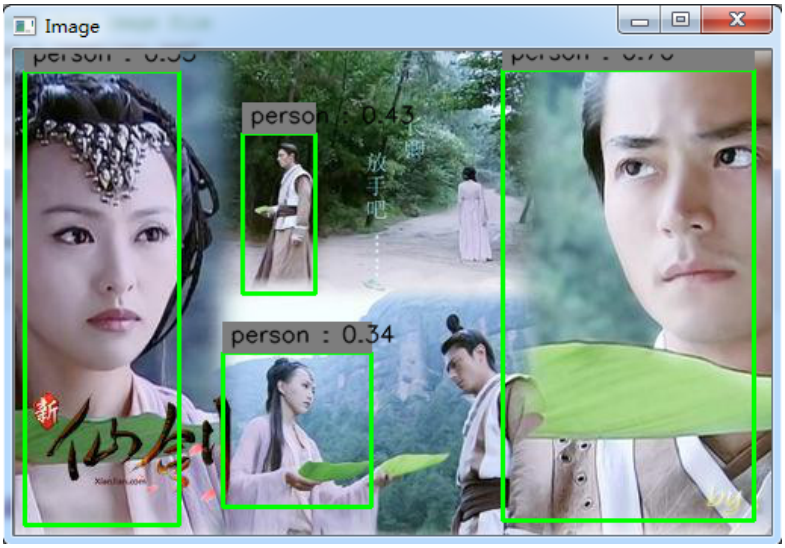
介绍完了Detector这个类，我们来看一下main函数。该函数比较检测，首先解析命令行参数，然后创建yolo网络，以及检测器对象，最后调用image_detector()函数对图片进行目标检测。

```
def main():
    #创建一个解析器对象，并告诉它将会有些什么参数。当程序运行时，该解析器就可以用于处理命令行参数。
```

```
    #https://www.cnblogs.com/lovemyspring/p/3214598.html
    parser = argparse.ArgumentParser()
    #定义参数
    parser.add_argument('--weights', default="YOLO_small.ckpt", type=str)
    parser.add_argument('--weight_dir', default='weights', type=str)
    parser.add_argument('--data_dir', default="data", type=str)
    parser.add_argument('--gpu', default='', type=str)
    #定义了所有参数之后，你就可以给 parse_args() 传递一组参数字符串来解析命令行。默认情况下，参数是从
sys.argv[1:] 中获取
    #parse_args() 的返回值是一个命名空间，包含传递给命令的参数。该对象将参数保存其属性
    args = parser.parse_args()

    #设置环境变量
    os.environ['CUDA_VISIBLE_DEVICES'] = args.gpu

    #创建YOLO网络对象
    yolo = YOLONet(False)
    #加载检查点文件
    weight_file = os.path.join(args.data_dir, args.weight_dir, args.weights)
    weight_file = './data/pascal_voc/weights/YOLO_small.ckpt'
    #weight_file = './data/pascal_voc/output/2018_07_09_17_00/yolo.ckpt-1000'

    #创建测试对象
    detector = Detector(yolo, weight_file)

    # detect from camera
    # cap = cv2.VideoCapture(-1)
    # detector.camera_detector(cap)

    # detect from image file
    imname = 'test/car.jpg'
    detector.image_detector(imname)
```

我们执行如下代码，开始测试网络：

```
if __name__ == '__main__':
    tf.reset_default_graph()
    main()
```



我们可以看到yolo网络对小目标检测效果并不好，漏检了一个目标。这主要与yolo的网络结构以及损失函数有关。除此之外yolo网络还有一些其他缺点，我们总结如下：

- 漏检。每个网格只预测一个类别的边界框，而且最后只取置信度最大的那个边界框。这就导致如果多个不同物体(或者同类物体的不同实体)的中心落在同一个网格中，会造成漏检。yolo对相互靠的很近的物体，还有很小的群体检测效果不好，这是因为一个网格中只预测了两个框，并且只属于一类。
- 位置精准性差。召回率低。由于损失函数的问题，定位误差是影响检测效果的主要原因。尤其是大小物体的处理上，还有待加强。
- 对测试图像中，同一类物体出现的新的不常见的长宽比和其他情况是。泛化能力偏弱。

参考文章：

[1]argparse - 命令行选项与参数解析（转）

[2]Yolo v1详解及相关问题解答

分类： 深度学习

大奥特曼打小怪兽
关注 - 9
粉丝 - 248
+加关注

2 0

« 上一篇： 第十三节、SURF特征提取算法
» 下一篇： 第十四节、FAST角点检测(附源码)

posted @ 2018-08-25 18:12 大奥特曼打小怪兽 阅读(3193) 评论(9) 编辑 收藏

评论列表

#1楼 2018-11-12 14:51 voyage-class

博主，我最近刚开始学。 一般不都是把 xml转换成txt吗。 这里为什么要把 CSV转换成tfrecord形式呢. 另外请指导一下，我换成自己的图片， 需要改哪些东西呢？

支持(0) 反对(0)

#2楼 [楼主] 2018-11-15 16:15 大奥特曼打小怪兽

@ voyage-class
你自己的数据也要和VOC 2012数据格式一样，具体可以看一下数据读取的代码。

支持(0) 反对(0)

#3楼 2018-11-20 22:37 voyage-class

博主你好，我按照你的例子，只取了前1000张图片和标签，进行训练为什么还是不行呢，其他的路径都没改，只是数据量少了

支持(0) 反对(0)

#4楼 [楼主] 2018-11-21 09:37 大奥特曼打小怪兽

@ voyage-class
你可以使用VOC 2012数据集训练测试一下？看看是程序有问题，还是你自己的数据集有问题

支持(0) 反对(0)

#5楼 2018-11-21 09:40 voyage-class

@ 大奥特曼打小怪兽
博主我能加你吗，关于这个问题我困惑好长时间了我想请教你一下，或者我的QQ416347260十分感谢

支持(0) 反对(0)

#6楼 2019-02-09 21:38 大鲨鱼呀

博主，很感谢您！看了很多其他的代码解析都是避重就轻，但我看了你的注释后豁然开朗。谢谢~

支持(0) 反对(0)

#7楼 2019-02-10 15:50 大鲨鱼呀

博主，我今天试了一下train，但是好像不能运行，是因为我版本太高了吗？
python 3.6
tensorflow 1.9.0
上面的内容舍去了，下面运行结果就这样。无法继续运行下去了
Loading gt_labels from: data\pascal_voc\cache\pascal_train_gt_labels.pkl
Appending horizontally-flipped training examples ...
Restoring weights from: data\pascal_voc\weights\YOLO_small.ckpt
INFO:tensorflow:Restoring parameters from data\pascal_voc\weights\YOLO_small.ckpt
2019-02-10 15:40:34,338 INFO Restoring parameters from data\pascal_voc\weights\YOLO_small.ckpt

Process finished with exit code -1073741819 (0xC0000005)

==========

我从github上又下了一遍源码，但github上的源码是可运行的。。是不是博主你哪个地方改错了

支持(0) 反对(0)

#8楼 2019-03-12 10:09 dlb421

博主你好,请问训练数据,需要跑多长时间?

支持(0) 反对(0)

#9楼 [楼主] 2019-03-13 10:32 大奥特曼打小怪兽

@ dlb421
看你电脑配置了

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册， 访问 网站首页。

相关博文：
· yolo源码解析（三）
· yolo源码解析（一）
· 目标检测-yolo
· [目标检测]YOLO原理
· yolo源码解析（二）

最新 IT 新闻:
· AWS 日本出现大规模故障
· 高通获准暂停执行反垄断判决，积极寻求上诉
· 一个有着127年历史的物理学谜题
· 带着机器人的俄罗斯飞船 与国际空间站对接失败
· 用意念打字，中国脑机接口新纪录诞生！
» 更多新闻...