案例九：基于Faster RCNN的目标检测

1 案例目的

• 应用Faster RCNN进行目标识别；

• 掌握Faster RCNN算法。

2 案例内容

对所给的图像进行目标识别

3 案例知识点

• Python语言编程

• VGG-16应用

• Faster RCNN算法

4 案例时长

共2.5学时，具体安排如下：

• 数据预处理（0.5学时）

• 做关联规则分析（1学时）

• 查看分析结果（0.5学时）

• 增加结果可读性（0.5学时）

5 案例实验环境

•操作系统：

1）Windows 10 x64位操作系统

•软件环境：

1）Python 3.7

2）apyori 1.1

•开发环境与工具：

1)Tensorflow r1.0

2)Docker

6 案例分析

本案例主要分为以下6部分：

(1)图片输入；

(2)利用selective search 算法在图像中从上到下提取2000个左右的建议窗口(Region Proposal)；

(3)将整张图片输入CNN，进行特征提取；

(4)把建议窗口映射到CNN的最后一层卷积feature map上；

(5)通过RoI pooling层使每个建议窗口生成固定尺寸的feature map；

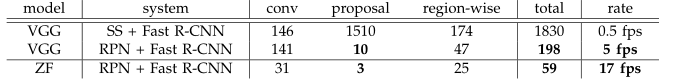
(6)利用Softmax Loss(探测分类概率) 和Smooth L1 Loss(探测边框回归)对分类概率和边框回归(Bounding box regression)联合训练.

1. 案例实验过程

7.1准备实验数据

该数据集由约5k的训练图像和5k的测试图像组成，超过20个目标类别。我们还提供了一些模型的PASCAL VOC 2012基准的结果。对于ImageNet预训练网络，我们使用ZF网络的“快速”版本，它有5个卷积层和3个完全连接层，以及VGG -16模型7，它有13个卷积层和3个完全连接层。我们主要评估检测平均值平均精度(mAP)，因为这是对象检测的实际度量(而不是关注对象提议代理度量)。表2(顶部)显示了使用各种区域建议方法进行训练和测试时的Fast R-CNN结果。这些结果使用了ZF网。对于选择搜索(SS)，我们通过“快速”模式生成了大约2000个建议。对于EdgeBoxes (EB)，我们通过调整为0.7IoU的默认EB设置生成建议。在Fast R-CNN框架下，SS的mAP为58.7%，EB的mAP为58.6%。RPN与Fast R-CNN的竞争结果，与59.9%的地图，而使用多达300个提议8。使用RPN产生一个比使用SS或EB更快的检测系统，因为共享卷积计算;提案的减少也降低了区域级全连接层的成本(表1)。

表 1:K40 GPU上的计时(毫秒)，除非SS提议是在CPU中评估的。“区域明智”包括NMS，池，全连接和softmax层。



7.2RPN的烧蚀实验

为了研究rpn的行为作为一种建议方法，我们进行了几项消融研究。首先，我们展示了RPN和Fast R-CNN检测网络之间共享卷积层的效果。为了做到这一点，我们在四步训练过程的第二步之后停止。使用单独的网络将结果略微降低到58.7%。我们观察到，这是因为在第三步中，当使用检测器调谐特征来微调RPN时，提议质量得到了改善。

接下来，我们解开RPN对训练Fast R-CNN检测网络的影响。为此，我们使用2000 SS建议和ZF网络训练一个Fast R-CNN模型。我们修正了这个检测器，并通过改变测试时使用的建议区域来评估检测映射。

在测试时间用300个RPN提议替换SS会导致56.8%的mAP。mAP中的损失是由于培训/测试提议之间的不一致。这个结果作为下面比较的基准。在测试时使用排名前100的建议时，RPN仍然会导致竞争结果(55.1%)，这表明排名前的RPN建议是准确的。在另一个极端，使用排名最高的6000个RPN提议(没有NMS)有一个可比较的mAP(55.2%)，这表明NMS不会损害检测mAP，并可能减少假警报。

接下来，我们将通过在测试时关闭它们中的任何一个来分别研究RPN的cls和reg输出的角色。当cls层在测试时被删除时(因此不使用NMS/排名)，我们从未评分的区域中随机抽样N个建议。当N = 1000时，mAP基本不变(55.8%)，但当N = 100时，mAP明显下降至44.6%。这表明cls得分决定了排名最高的提案的准确性。

另一方面，当reg层在测试时被删除时(因此建议成为锚框)，mAP下降到52.1%。这表明，高质量的建议主要是由于回归框边界。锚盒虽然具有多个尺度和高宽比，但不足以进行精确检测。

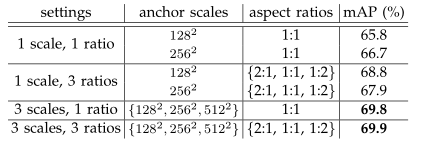
7.3研究VGG-16的性能

表1总结了整个目标检测系统的运行时间。SS需要1-2秒，这取决于内容(平均约1.5秒)，快速R-CNN与vg -16在2000 SS提议需要320ms(或223ms，如果使用SVD全连接层)。我们的VGG-16系统在提案和检测总共需要198毫秒。由于共享了卷积特性，仅RPN计算额外的层只需要10ms。我们的区域计算也更低，这要感谢更少的建议(每幅图像300个)。我们的系统在ZF网络下的帧率是17fps。

7.4超参数的敏感度

在表2中，我们调查了锚点的设置。默认情况下，我们使用3个比例尺和3个纵横比(表8中69.9%的地图)。如果在每个位置只使用一个锚点，地图将下降3-4%。如果使用3个比例尺(1个纵横比)或3个纵横比(1个比例尺)，地图会更高，这表明使用多个尺寸的锚点作为回归参考是一种有效的解决方案。在这个数据集中，使用3个尺度和1个高深比(69.8%)和使用3个尺度和3个高深比是一样的，这表明对于检测精度来说，尺度和高深比并不是分离的维度。但我们仍然在设计中采用这两个维度，以保持系统的灵活性。

表 2使用不同的锚点设置，Faster R-CNN在PASCAL VOC 2007测试集上的检测结果。



7.5结果分析

在图4中，我们展示了使用300个、1000个和2000个建议的结果。我们与SS和EB方法进行了比较，根据这些方法生成的置信度，N个建议是排名前N的。这些图表明，当建议的数量从2000个下降到300个时，RPN方法表现得很优雅。这就解释了为什么RPN在使用300个提案时具有良好的最终检测mAP。正如我们之前分析的，这个性质主要归因于RPN的cls项。当提案较少时，SS和EB的召回率比RPN下降得更快。

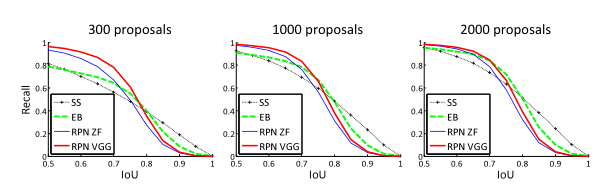


图 4在PASCAL VOC 2007测试集上的召回与欠条重叠比率

我们提出了rpn，用于高效和准确的区域建议生成。通过与下游检测网络共享卷积特征，区域建议步骤几乎是免费的。我们的方法使一个统一的，基于深度学习的目标检测系统运行在接近实时的帧率。学习到的RPN还提高了区域提议质量，从而提高了整体目标检测精度。

8案例代码

# --------------------------------------------------------

# Fast R-CNN

# --------------------------------------------------------

from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

import numpy as np

import tensorflow as tf

def bbox\_transform(ex\_rois, gt\_rois):

ex\_widths = ex\_rois[:, 2] - ex\_rois[:, 0] + 1.0

ex\_heights = ex\_rois[:, 3] - ex\_rois[:, 1] + 1.0

ex\_ctr\_x = ex\_rois[:, 0] + 0.5 \* ex\_widths

ex\_ctr\_y = ex\_rois[:, 1] + 0.5 \* ex\_heights

gt\_widths = gt\_rois[:, 2] - gt\_rois[:, 0] + 1.0

gt\_heights = gt\_rois[:, 3] - gt\_rois[:, 1] + 1.0

gt\_ctr\_x = gt\_rois[:, 0] + 0.5 \* gt\_widths

gt\_ctr\_y = gt\_rois[:, 1] + 0.5 \* gt\_heights

targets\_dx = (gt\_ctr\_x - ex\_ctr\_x) / ex\_widths

targets\_dy = (gt\_ctr\_y - ex\_ctr\_y) / ex\_heights

targets\_dw = np.log(gt\_widths / ex\_widths)

targets\_dh = np.log(gt\_heights / ex\_heights)

targets = np.vstack(

(targets\_dx, targets\_dy, targets\_dw, targets\_dh)).transpose()

return targets

def bbox\_transform\_inv(boxes, deltas):

if boxes.shape[0] == 0:

return np.zeros((0, deltas.shape[1]), dtype=deltas.dtype)

boxes = boxes.astype(deltas.dtype, copy=False)

widths = boxes[:, 2] - boxes[:, 0] + 1.0

heights = boxes[:, 3] - boxes[:, 1] + 1.0

ctr\_x = boxes[:, 0] + 0.5 \* widths

ctr\_y = boxes[:, 1] + 0.5 \* heights

dx = deltas[:, 0::4]

dy = deltas[:, 1::4]

dw = deltas[:, 2::4]

dh = deltas[:, 3::4]

pred\_ctr\_x = dx \* widths[:, np.newaxis] + ctr\_x[:, np.newaxis]

pred\_ctr\_y = dy \* heights[:, np.newaxis] + ctr\_y[:, np.newaxis]

pred\_w = np.exp(dw) \* widths[:, np.newaxis]

pred\_h = np.exp(dh) \* heights[:, np.newaxis]

pred\_boxes = np.zeros(deltas.shape, dtype=deltas.dtype)

# x1

pred\_boxes[:, 0::4] = pred\_ctr\_x - 0.5 \* pred\_w

# y1

pred\_boxes[:, 1::4] = pred\_ctr\_y - 0.5 \* pred\_h

# x2

pred\_boxes[:, 2::4] = pred\_ctr\_x + 0.5 \* pred\_w

# y2

pred\_boxes[:, 3::4] = pred\_ctr\_y + 0.5 \* pred\_h

return pred\_boxes

def clip\_boxes(boxes, im\_shape):

"""

Clip boxes to image boundaries.

"""

# x1 >= 0

boxes[:, 0::4] = np.maximum(np.minimum(boxes[:, 0::4], im\_shape[1] - 1), 0)

# y1 >= 0

boxes[:, 1::4] = np.maximum(np.minimum(boxes[:, 1::4], im\_shape[0] - 1), 0)

# x2 < im\_shape[1]

boxes[:, 2::4] = np.maximum(np.minimum(boxes[:, 2::4], im\_shape[1] - 1), 0)

# y2 < im\_shape[0]

boxes[:, 3::4] = np.maximum(np.minimum(boxes[:, 3::4], im\_shape[0] - 1), 0)

return boxes

def bbox\_transform\_inv\_tf(boxes, deltas):

boxes = tf.cast(boxes, deltas.dtype)

widths = tf.subtract(boxes[:, 2], boxes[:, 0]) + 1.0

heights = tf.subtract(boxes[:, 3], boxes[:, 1]) + 1.0

ctr\_x = tf.add(boxes[:, 0], widths \* 0.5)

ctr\_y = tf.add(boxes[:, 1], heights \* 0.5)

dx = deltas[:, 0]

dy = deltas[:, 1]

dw = deltas[:, 2]

dh = deltas[:, 3]

pred\_ctr\_x = tf.add(tf.multiply(dx, widths), ctr\_x)

pred\_ctr\_y = tf.add(tf.multiply(dy, heights), ctr\_y)

pred\_w = tf.multiply(tf.exp(dw), widths)

pred\_h = tf.multiply(tf.exp(dh), heights)

pred\_boxes0 = tf.subtract(pred\_ctr\_x, pred\_w \* 0.5)

pred\_boxes1 = tf.subtract(pred\_ctr\_y, pred\_h \* 0.5)

pred\_boxes2 = tf.add(pred\_ctr\_x, pred\_w \* 0.5)

pred\_boxes3 = tf.add(pred\_ctr\_y, pred\_h \* 0.5)

return tf.stack([pred\_boxes0, pred\_boxes1, pred\_boxes2, pred\_boxes3], axis=1)

def clip\_boxes\_tf(boxes, im\_info):

b0 = tf.maximum(tf.minimum(boxes[:, 0], im\_info[1] - 1), 0)

b1 = tf.maximum(tf.minimum(boxes[:, 1], im\_info[0] - 1), 0)

b2 = tf.maximum(tf.minimum(boxes[:, 2], im\_info[1] - 1), 0)

b3 = tf.maximum(tf.minimum(boxes[:, 3], im\_info[0] - 1), 0)

return tf.stack([b0, b1, b2, b3], axis=1)

#train-val.py

# --------------------------------------------------------

# Tensorflow Faster R-CNN

# --------------------------------------------------------

from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

from model.config import cfg

import roi\_data\_layer.roidb as rdl\_roidb

from roi\_data\_layer.layer import RoIDataLayer

from utils.timer import Timer

try:

import cPickle as pickle

except ImportError:

import pickle

import numpy as np

import os

import sys

import glob

import time

import tensorflow as tf

from tensorflow.python import pywrap\_tensorflow

class SolverWrapper(object):

"""

A wrapper class for the training process

"""

def \_\_init\_\_(self, sess, network, imdb, roidb, valroidb, output\_dir, tbdir, pretrained\_model=None):

self.net = network

self.imdb = imdb

self.roidb = roidb

self.valroidb = valroidb

self.output\_dir = output\_dir

self.tbdir = tbdir

# Simply put '\_val' at the end to save the summaries from the validation set

self.tbvaldir = tbdir + '\_val'

if not os.path.exists(self.tbvaldir):

os.makedirs(self.tbvaldir)

self.pretrained\_model = pretrained\_model

def snapshot(self, sess, iter):

net = self.net

if not os.path.exists(self.output\_dir):

os.makedirs(self.output\_dir)

# Store the model snapshot

filename = cfg.TRAIN.SNAPSHOT\_PREFIX + '\_iter\_{:d}'.format(iter) + '.ckpt'

filename = os.path.join(self.output\_dir, filename)

self.saver.save(sess, filename)

print('Wrote snapshot to: {:s}'.format(filename))

# Also store some meta information, random state, etc.

nfilename = cfg.TRAIN.SNAPSHOT\_PREFIX + '\_iter\_{:d}'.format(iter) + '.pkl'

nfilename = os.path.join(self.output\_dir, nfilename)

# current state of numpy random

st0 = np.random.get\_state()

# current position in the database

cur = self.data\_layer.\_cur

# current shuffled indexes of the database

perm = self.data\_layer.\_perm

# current position in the validation database

cur\_val = self.data\_layer\_val.\_cur

# current shuffled indexes of the validation database

perm\_val = self.data\_layer\_val.\_perm

# Dump the meta info

with open(nfilename, 'wb') as fid:

pickle.dump(st0, fid, pickle.HIGHEST\_PROTOCOL)

pickle.dump(cur, fid, pickle.HIGHEST\_PROTOCOL)

pickle.dump(perm, fid, pickle.HIGHEST\_PROTOCOL)

pickle.dump(cur\_val, fid, pickle.HIGHEST\_PROTOCOL)

pickle.dump(perm\_val, fid, pickle.HIGHEST\_PROTOCOL)

pickle.dump(iter, fid, pickle.HIGHEST\_PROTOCOL)

return filename, nfilename

def from\_snapshot(self, sess, sfile, nfile):

print('Restoring model snapshots from {:s}'.format(sfile))

self.saver.restore(sess, sfile)

print('Restored.')

# Needs to restore the other hyper-parameters/states for training, (TODO xinlei) I have

# tried my best to find the random states so that it can be recovered exactly

# However the Tensorflow state is currently not available

with open(nfile, 'rb') as fid:

st0 = pickle.load(fid)

cur = pickle.load(fid)

perm = pickle.load(fid)

cur\_val = pickle.load(fid)

perm\_val = pickle.load(fid)

last\_snapshot\_iter = pickle.load(fid)

np.random.set\_state(st0)

self.data\_layer.\_cur = cur

self.data\_layer.\_perm = perm

self.data\_layer\_val.\_cur = cur\_val

self.data\_layer\_val.\_perm = perm\_val

return last\_snapshot\_iter

def get\_variables\_in\_checkpoint\_file(self, file\_name):

try:

reader = pywrap\_tensorflow.NewCheckpointReader(file\_name)

var\_to\_shape\_map = reader.get\_variable\_to\_shape\_map()

return var\_to\_shape\_map

except Exception as e: # pylint: disable=broad-except

print(str(e))

if "corrupted compressed block contents" in str(e):

print("It's likely that your checkpoint file has been compressed "

"with SNAPPY.")

def construct\_graph(self, sess):

with sess.graph.as\_default():

# Set the random seed for tensorflow

tf.set\_random\_seed(cfg.RNG\_SEED)

# Build the main computation graph

layers = self.net.create\_architecture('TRAIN', self.imdb.num\_classes, tag='default',anchor\_scales=cfg.ANCHOR\_SCALES,

anchor\_ratios=cfg.ANCHOR\_RATIOS)

# Define the loss

loss = layers['total\_loss']

# Set learning rate and momentum

lr = tf.Variable(cfg.TRAIN.LEARNING\_RATE, trainable=False)

self.optimizer = tf.train.MomentumOptimizer(lr, cfg.TRAIN.MOMENTUM)

# Compute the gradients with regard to the loss

gvs = self.optimizer.compute\_gradients(loss)

# Double the gradient of the bias if set

if cfg.TRAIN.DOUBLE\_BIAS:

final\_gvs = []

with tf.variable\_scope('Gradient\_Mult') as scope:

for grad, var in gvs:

scale = 1.

if cfg.TRAIN.DOUBLE\_BIAS and '/biases:' in var.name:

scale \*= 2.

if not np.allclose(scale, 1.0):

grad = tf.multiply(grad, scale)

final\_gvs.append((grad, var))

train\_op = self.optimizer.apply\_gradients(final\_gvs)

else:

train\_op = self.optimizer.apply\_gradients(gvs)

# We will handle the snapshots ourselves

self.saver = tf.train.Saver(max\_to\_keep=100000)

# Write the train and validation information to tensorboard

self.writer = tf.summary.FileWriter(self.tbdir, sess.graph)

self.valwriter = tf.summary.FileWriter(self.tbvaldir)

return lr, train\_op

def find\_previous(self):

sfiles = os.path.join(self.output\_dir, cfg.TRAIN.SNAPSHOT\_PREFIX + '\_iter\_\*.ckpt.meta')

sfiles = glob.glob(sfiles)

sfiles.sort(key=os.path.getmtime)

# Get the snapshot name in TensorFlow

redfiles = []

for stepsize in cfg.TRAIN.STEPSIZE:

redfiles.append(os.path.join(self.output\_dir,

cfg.TRAIN.SNAPSHOT\_PREFIX + '\_iter\_{:d}.ckpt.meta'.format(stepsize+1)))

sfiles = [ss.replace('.meta', '') for ss in sfiles if ss not in redfiles]

nfiles = os.path.join(self.output\_dir, cfg.TRAIN.SNAPSHOT\_PREFIX + '\_iter\_\*.pkl')

nfiles = glob.glob(nfiles)

nfiles.sort(key=os.path.getmtime)

redfiles = [redfile.replace('.ckpt.meta', '.pkl') for redfile in redfiles]

nfiles = [nn for nn in nfiles if nn not in redfiles]

lsf = len(sfiles)

assert len(nfiles) == lsf

return lsf, nfiles, sfiles

def initialize(self, sess):

# Initial file lists are empty

np\_paths = []

ss\_paths = []

# Fresh train directly from ImageNet weights

print('Loading initial model weights from {:s}'.format(self.pretrained\_model))

variables = tf.global\_variables()

# Initialize all variables first

sess.run(tf.variables\_initializer(variables, name='init'))

var\_keep\_dic = self.get\_variables\_in\_checkpoint\_file(self.pretrained\_model)

# Get the variables to restore, ignoring the variables to fix

variables\_to\_restore = self.net.get\_variables\_to\_restore(variables, var\_keep\_dic)

restorer = tf.train.Saver(variables\_to\_restore)

restorer.restore(sess, self.pretrained\_model)

print('Loaded.')

# Need to fix the variables before loading, so that the RGB weights are changed to BGR

# For VGG16 it also changes the convolutional weights fc6 and fc7 to

# fully connected weights

self.net.fix\_variables(sess, self.pretrained\_model)

print('Fixed.')

last\_snapshot\_iter = 0

rate = cfg.TRAIN.LEARNING\_RATE

stepsizes = list(cfg.TRAIN.STEPSIZE)

return rate, last\_snapshot\_iter, stepsizes, np\_paths, ss\_paths

def restore(self, sess, sfile, nfile):

# Get the most recent snapshot and restore

np\_paths = [nfile]

ss\_paths = [sfile]

# Restore model from snapshots

last\_snapshot\_iter = self.from\_snapshot(sess, sfile, nfile)

# Set the learning rate

rate = cfg.TRAIN.LEARNING\_RATE

stepsizes = []

for stepsize in cfg.TRAIN.STEPSIZE:

if last\_snapshot\_iter > stepsize:

rate \*= cfg.TRAIN.GAMMA

else:

stepsizes.append(stepsize)

return rate, last\_snapshot\_iter, stepsizes, np\_paths, ss\_paths

def remove\_snapshot(self, np\_paths, ss\_paths):

to\_remove = len(np\_paths) - cfg.TRAIN.SNAPSHOT\_KEPT

for c in range(to\_remove):

nfile = np\_paths[0]

os.remove(str(nfile))

np\_paths.remove(nfile)

to\_remove = len(ss\_paths) - cfg.TRAIN.SNAPSHOT\_KEPT

for c in range(to\_remove):

sfile = ss\_paths[0]

# To make the code compatible to earlier versions of Tensorflow,

# where the naming tradition for checkpoints are different

if os.path.exists(str(sfile)):

os.remove(str(sfile))

else:

os.remove(str(sfile + '.data-00000-of-00001'))

os.remove(str(sfile + '.index'))

sfile\_meta = sfile + '.meta'

os.remove(str(sfile\_meta))

ss\_paths.remove(sfile)

def train\_model(self, sess, max\_iters):

# Build data layers for both training and validation set

self.data\_layer = RoIDataLayer(self.roidb, self.imdb.num\_classes)

self.data\_layer\_val = RoIDataLayer(self.valroidb, self.imdb.num\_classes, random=True)

# Construct the computation graph

lr, train\_op = self.construct\_graph(sess)

# Find previous snapshots if there is any to restore from

lsf, nfiles, sfiles = self.find\_previous()

# Initialize the variables or restore them from the last snapshot

if lsf == 0:

rate, last\_snapshot\_iter, stepsizes, np\_paths, ss\_paths = self.initialize(sess)

else:

rate, last\_snapshot\_iter, stepsizes, np\_paths, ss\_paths = self.restore(sess,

str(sfiles[-1]),

str(nfiles[-1]))

timer = Timer()

iter = last\_snapshot\_iter + 1

last\_summary\_time = time.time()

# Make sure the lists are not empty

stepsizes.append(max\_iters)

stepsizes.reverse()

next\_stepsize = stepsizes.pop()

while iter < max\_iters + 1:

# Learning rate

if iter == next\_stepsize + 1:

# Add snapshot here before reducing the learning rate

self.snapshot(sess, iter)

rate \*= cfg.TRAIN.GAMMA

sess.run(tf.assign(lr, rate))

next\_stepsize = stepsizes.pop()

timer.tic()

# Get training data, one batch at a time

blobs = self.data\_layer.forward()

now = time.time()

if iter == 1 or now - last\_summary\_time > cfg.TRAIN.SUMMARY\_INTERVAL:

# Compute the graph with summary

rpn\_loss\_cls, rpn\_loss\_box, loss\_cls, loss\_box, total\_loss, summary = \

self.net.train\_step\_with\_summary(sess, blobs, train\_op)

self.writer.add\_summary(summary, float(iter))

# Also check the summary on the validation set

blobs\_val = self.data\_layer\_val.forward()

summary\_val = self.net.get\_summary(sess, blobs\_val)

self.valwriter.add\_summary(summary\_val, float(iter))

last\_summary\_time = now

else:

# Compute the graph without summary

rpn\_loss\_cls, rpn\_loss\_box, loss\_cls, loss\_box, total\_loss = \

self.net.train\_step(sess, blobs, train\_op)

timer.toc()

# Display training information

if iter % (cfg.TRAIN.DISPLAY) == 0:

print('iter: %d / %d, total loss: %.6f\n >>> rpn\_loss\_cls: %.6f\n '

'>>> rpn\_loss\_box: %.6f\n >>> loss\_cls: %.6f\n >>> loss\_box: %.6f\n >>> lr: %f' % \(iter, max\_iters, total\_loss, rpn\_loss\_cls, rpn\_loss\_box, loss\_cls, loss\_box, lr.eval()))

print('speed: {:.3f}s / iter'.format(timer.average\_time))

# Snapshotting

if iter % cfg.TRAIN.SNAPSHOT\_ITERS == 0:

last\_snapshot\_iter = iter

ss\_path, np\_path = self.snapshot(sess, iter)

np\_paths.append(np\_path)

ss\_paths.append(ss\_path)

# Remove the old snapshots if there are too many

if len(np\_paths) > cfg.TRAIN.SNAPSHOT\_KEPT:

self.remove\_snapshot(np\_paths, ss\_paths)

iter += 1

if last\_snapshot\_iter != iter - 1:

self.snapshot(sess, iter - 1)

self.writer.close()

self.valwriter.close()

"""Returns a roidb (Region of Interest database) for use in training."""

if cfg.TRAIN.USE\_FLIPPED:

print('Appending horizontally-flipped training examples...')

imdb.append\_flipped\_images()

print('done')

print('Preparing training data...')

rdl\_roidb.prepare\_roidb(imdb)

print('done')

return imdb.roidb

def filter\_roidb(roidb):

"""Remove roidb entries that have no usable RoIs."""

def is\_valid(entry):

# Valid images have:

# (1) At least one foreground RoI OR

# (2) At least one background RoI

overlaps = entry['max\_overlaps']

# find boxes with sufficient overlap

fg\_inds = np.where(overlaps >= cfg.TRAIN.FG\_THRESH)[0]

# Select background RoIs as those within [BG\_THRESH\_LO, BG\_THRESH\_HI)

bg\_inds = np.where((overlaps < cfg.TRAIN.BG\_THRESH\_HI) &

(overlaps >= cfg.TRAIN.BG\_THRESH\_LO))[0]

# image is only valid if such boxes exist

valid = len(fg\_inds) > 0 or len(bg\_inds) > 0

return valid

num = len(roidb)

filtered\_roidb = [entry for entry in roidb if is\_valid(entry)]

num\_after = len(filtered\_roidb)

print('Filtered {} roidb entries: {} -> {}'.format(num - num\_after,

num, num\_after))

return filtered\_roidb

def train\_net(network, imdb, roidb, valroidb, output\_dir, tb\_dir,

pretrained\_model=None,

max\_iters=40000):

"""Train a Faster R-CNN network."""

roidb = filter\_roidb(roidb)

valroidb = filter\_roidb(valroidb)

tfconfig = tf.ConfigProto(allow\_soft\_placement=True)

tfconfig.gpu\_options.allow\_growth = True

with tf.Session(config=tfconfig) as sess:

sw = SolverWrapper(sess, network, imdb, roidb, valroidb, output\_dir, tb\_dir,

pretrained\_model=pretrained\_model)

print('Solving...')

sw.train\_model(sess, max\_iters)

print('done solving')