Applying Faster R-CNN for Object Detection on Malaria Images

https://data.broadinstitute.org/bbbc/BBBC041/ (https://data.broadinstitute.org/bbbc/BBBC041/)

https://arxiv.org/abs/1804.09548 (https://arxiv.org/abs/1804.09548)

Deep learning based models have had great success in object detection, butthe state of the art models have not yet been widely applied to biologicalimage data. We apply for the first time an object detection model previouslyused on natural images to identify cells and recognize their stages inbrightfield microscopy images of malaria-infected blood. Many micro-organismslike malaria parasites are still studied by expert manual inspection and handcounting. This type of object detection task is challenging due to factors likevariations in cell shape, density, and color, and uncertainty of some cellclasses. In addition, annotated data useful for training is scarce, and theclass distribution is inherently highly imbalanced due to the dominance ofuninfected red blood cells. We use Faster Region-based Convolutional NeuralNetwork (Faster R-CNN), one of the top performing object detection models inrecent years, pre-trained on ImageNet but fine tuned with our data, and compareit to a baseline, which is based on a traditional approach consisting of cellsegmentation, extraction of several single-cell features, and classificationusing random forests. To conduct our initial study, we collect and label adataset of 1300 fields of view consisting of around 100,000 individual cells.We demonstrate that Faster R-CNN outperforms our baseline and put the resultsin context of human performance.

Metric

Mean Absolute Precision

Losses

RPN(Region Proposal Network) Classification & Regression Losses

R-CNN classification & Regression Losses

Import libs

In [1]:

```
from __future__ import division
from __future__ import print_function
from __future__ import absolute_import
import random
import pprint
import sys
import time
import numpy as np
from optparse import OptionParser
import pickle
import math
import cv2
import copy
from matplotlib import pyplot as plt
import tensorflow as tf
import pandas as pd
import os

from sklearn.metrics import average_precision_score

from keras import backend as K
from keras.optimizers import Adam, SGD, RMSprop
from keras.layers import Flatten, Dense, Input, Conv2D, MaxPooling2D, Dropout
from keras.layers import GlobalAveragePooling2D, GlobalMaxPooling2D, TimeDistributed
from keras.engine.topology import get_source_inputs
from keras.utils import layer_utils
from keras.utils.data_utils import get_file
from keras.objectives import categorical_crossentropy

from keras.models import Model
from keras.utils import generic_utils
from keras.engine import Layer, InputSpec
from keras import initializers, regularizers
```

```
Using TensorFlow backend.
```

Reference Links:

https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/ (https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/)

https://towardsdatascience.com/faster-r-cnn-object-detection-implemented-by-keras-for-custom-data-from-googles-open-images-125f62b9141a (https://towardsdatascience.com/faster-r-cnn-object-detection-implemented-by-keras-for-custom-data-from-googles-open-images-125f62b9141a)

https://www.analyticsvidhya.com/blog/2018/11/implementation-faster-r-cnn-python-object-detection/ (https://www.analyticsvidhya.com/blog/2018/11/implementation-faster-r-cnn-python-object-detection/)

https://github.com/RockyXu66/Faster_RCNN_for_Open_Images_Dataset_Keras (https://github.com/RockyXu66/Faster_RCNN_for_Open_Images_Dataset_Keras)

https://towardsdatascience.com/faster-r-cnn-object-detection-implemented-by-keras-for-custom-data-from-googles-open-images-125f62b9141a (https://towardsdatascience.com/faster-r-cnn-object-detection-implemented-by-keras-for-custom-data-from-googles-open-images-125f62b9141a)

Config setting

This is a config class where we initiate the base model to use, anchor_box_scales-adjusted as per the bounding boxes of the objects in the image, number of rois(region of interests) to return at once, image augmentation and various other parameters.

In [2]:

```
class Config:

        def __init__(self):

                # Print the process or not
                self.verbose = True

                # Name of base network
                self.network = 'vgg'

                # Setting for data augmentation
                self.use_horizontal_flips = False
                self.use_vertical_flips = False
                self.rot_90 = False

                # Anchor box scales
    # Note that if im_size is smaller, anchor_box_scales should be scaled
    # Original anchor_box_scales in the paper is [128, 256, 512]
                self.anchor_box_scales = [8,16,32]

                # Anchor box ratios
                self.anchor_box_ratios = [[1, 1], [1./math.sqrt(2), 2./math.sqrt(2)], [2./math.sqrt(2), 1./math.s
qrt(2)]]

                # Size to resize the smallest side of the image
                # Original setting in paper is 600. Set to 300 in here to save training time
                self.im_size = 300

                # image channel-wise mean to subtract
                self.img_channel_mean = [103.939, 116.779, 123.68]
                self.img_scaling_factor = 1.0

                # number of ROIs at once
                self.num_rois = 4

                # stride at the RPN (this depends on the network configuration)
                self.rpn_stride = 16

                self.balanced_classes = False

                # scaling the stdev
                self.std_scaling = 4.0
                self.classifier_regr_std = [8.0, 8.0, 4.0, 4.0]

                # overlaps for RPN
                self.rpn_min_overlap = 0.3
                self.rpn_max_overlap = 0.7

                # overlaps for classifier ROIs
                self.classifier_min_overlap = 0.1
                self.classifier_max_overlap = 0.5

                # placeholder for the class mapping, automatically generated by the parser
                self.class_mapping = None

                self.model_path = None
```

Parse the data from annotation file

Based on the annotation file input, number of classes and the bounding box information will be extracted.

In [3]:

```
def get_data(input_path):
        """Parse the data from annotation file

        Args:
                input_path: annotation file path

        Returns:
                all_data: list(filepath, width, height, list(bboxes))
                classes_count: dict{key:class_name, value:count_num}
                        e.g. {'red blood cell': 77420, 'trophozoite': 1473, 'difficult': 441, 'ring': 353,'schizo
nt':179, 'gametocyte':144,
            'leukocyte':103}
                class_mapping: dict{key:class_name, value: idx}
                        e.g. {'red blood cell': 0, 'trophozoite': 1, 'difficult': 2, 'ring': 3,'schizont':4, 'gam
etocyte':5,
            'leukocyte':6}
        """
```

```
	"""
	found_bg = False
	all_imgs = {}

	classes_count = {}

	class_mapping = {}

	visualise = True

	i = 1

	with open(input_path,'r') as f:

		print('Parsing annotation files')

		for line in f:

			# Print process
			sys.stdout.write('\r'+'idx=' + str(i))
			i += 1

			line_split = line.strip().split(',')

			# Make sure the info saved in annotation file matching the format (path_filename, x1, y1, 
x2, y2, class_name)
			# Note:
			#	One path_filename might has several classes (class_name)
			#	x1, y1, x2, y2 are the pixel value of the origial image, not the ratio value
			#	(x1, y1) top left coordinates; (x2, y2) bottom right coordinates
			#   x1,y1-------------------
			#	|						|
			#	|						|
			#	|						|
			#	|						|
			#	---------------------x2,y2

			(filename,x1,y1,x2,y2,class_name) = line_split

			if class_name not in classes_count:
				classes_count[class_name] = 1
			else:
				classes_count[class_name] += 1

			if class_name not in class_mapping:
				if class_name == 'bg' and found_bg == False:
					print('Found class name with special name bg. Will be treated as a backgr
ound region (this is usually for hard negative mining).')
					found_bg = True
				class_mapping[class_name] = len(class_mapping)

			if filename not in all_imgs:
				all_imgs[filename] = {}

				img = cv2.imread(filename)
				(rows,cols) = img.shape[:2]
				all_imgs[filename]['filepath'] = filename
				all_imgs[filename]['width'] = cols
				all_imgs[filename]['height'] = rows
				all_imgs[filename]['bboxes'] = []
				# if np.random.randint(0,6) > 0:
				#	all_imgs[filename]['imageset'] = 'trainval'
				# else:
				#	all_imgs[filename]['imageset'] = 'test'

			all_imgs[filename]['bboxes'].append({'class': class_name, 'x1': int(x1), 'x2': int(x2), '
y1': int(y1), 'y2': int(y2)})


		all_data = []
		for key in all_imgs:
			all_data.append(all_imgs[key])

		# make sure the bg class is last in the list
		if found_bg:
			if class_mapping['bg'] != len(class_mapping) - 1:
				key_to_switch = [key for key in class_mapping.keys() if class_mapping[key] == len
(class_mapping)-1][0]
				val_to_switch = class_mapping['bg']
				class_mapping['bg'] = len(class_mapping) - 1
				class_mapping[key_to_switch] = val_to_switch

		return all_data, classes_count, class_mapping
```

Define ROI Pooling Convolutional Layer

Region-based convolutional neural network (R-CNN) is the final step in Faster R-CNN's pipeline. After getting a convolutional feature map from the image, using it to get object proposals with the RPN and finally extracting features for each of those proposals (via RoI Pooling), we finally need to use these features for classification. R-CNN tries to mimic the final stages of classification CNNs where a fully-connected layer is used to output a score for each possible object class.

In [4]:

```
class RoiPoolingConv(Layer):
    '''ROI pooling layer for 2D inputs.
    See Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition,
    K. He, X. Zhang, S. Ren, J. Sun
    # Arguments
        pool_size: int
            Size of pooling region to use. pool_size = 7 will result in a 7x7 region.
        num_rois: number of regions of interest to be used
    # Input shape
        list of two 4D tensors [X_img,X_roi] with shape:
        X_img:
        `(1, rows, cols, channels)`
        X_roi:
        `(1,num_rois,4)` list of rois, with ordering (x,y,w,h)
    # Output shape
        3D tensor with shape:
        `(1, num_rois, channels, pool_size, pool_size)`
    '''
    def __init__(self, pool_size, num_rois, **kwargs):

        self.dim_ordering = K.image_data_format()
        self.pool_size = pool_size
        self.num_rois = num_rois

        super(RoiPoolingConv, self).__init__(**kwargs)

    def build(self, input_shape):
        self.nb_channels = input_shape[0][3]

    def compute_output_shape(self, input_shape):
        return None, self.num_rois, self.pool_size, self.pool_size, self.nb_channels

    def call(self, x, mask=None):

        assert(len(x) == 2)

        # x[0] is image with shape (rows, cols, channels)
        img = x[0]

        # x[1] is roi with shape (num_rois,4) with ordering (x,y,w,h)
        rois = x[1]

        input_shape = K.shape(img)

        outputs = []

        for roi_idx in range(self.num_rois):

            x = rois[0, roi_idx, 0]
            y = rois[0, roi_idx, 1]
            w = rois[0, roi_idx, 2]
            h = rois[0, roi_idx, 3]

            x = K.cast(x, 'int32')
            y = K.cast(y, 'int32')
            w = K.cast(w, 'int32')
            h = K.cast(h, 'int32')

            # Resized roi of the image to pooling size (7x7)
            rs = tf.image.resize_images(img[:, y:y+h, x:x+w, :], (self.pool_size, self.pool_size))
            outputs.append(rs)


        final_output = K.concatenate(outputs, axis=0)

        # Reshape to (1, num_rois, pool_size, pool_size, nb_channels)
        # Might be (1, 4, 7, 7, 3)
        final_output = K.reshape(final_output, (1, self.num_rois, self.pool_size, self.pool_size, self.nb_channel
s))

        # permute_dimensions is similar to transpose
        final_output = K.permute_dimensions(final_output, (0, 1, 2, 3, 4))
        return final_output


    def get_config(self):
        config = {'pool_size': self.pool_size,
                  'num_rois': self.num_rois}
        base_config = super(RoiPoolingConv, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))
```

Vgg-16 model

VGG-16 trained on imagenet is used as base model for detecting objects using Faster R-CNN

In [5]:

```
def get_img_output_length(width, height):
    def get_output_length(input_length):
        return input_length//16

    return get_output_length(width), get_output_length(height)

def nn_base(input_tensor=None, trainable=False):


    input_shape = (None, None, 3)

    if input_tensor is None:
        img_input = Input(shape=input_shape)
    else:
        if not K.is_keras_tensor(input_tensor):
            img_input = Input(tensor=input_tensor, shape=input_shape)
        else:
            img_input = input_tensor

    bn_axis = 3

    # Block 1
    x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv1')(img_input)
    x = Conv2D(64, (3, 3), activation='relu', padding='same', name='block1_conv2')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block1_pool')(x)

    # Block 2
    x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv1')(x)
    x = Conv2D(128, (3, 3), activation='relu', padding='same', name='block2_conv2')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block2_pool')(x)

    # Block 3
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv1')(x)
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv2')(x)
    x = Conv2D(256, (3, 3), activation='relu', padding='same', name='block3_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block3_pool')(x)

    # Block 4
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block4_conv3')(x)
    x = MaxPooling2D((2, 2), strides=(2, 2), name='block4_pool')(x)

    # Block 5
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv1')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv2')(x)
    x = Conv2D(512, (3, 3), activation='relu', padding='same', name='block5_conv3')(x)
    # x = MaxPooling2D((2, 2), strides=(2, 2), name='block5_pool')(x)

    return x
```

RPN layer

The RPN takes all the reference boxes (anchors) and outputs a set of good proposals for objects. It does this by having two different outputs for each of the anchors.

The first one is the probability that an anchor is an object. An “objectness score”, if you will. Note that the RPN doesn’t care what class of object it is, only that it does in fact look like an object (and not background). We are going to use this objectness score to filter out the bad predictions for the second stage. The second output is the bounding box regression for adjusting the anchors to better fit the object it’s predicting.

The RPN is implemented efficiently in a fully convolutional way, using the convolutional feature map returned by the base network as an input. First, we use a convolutional layer with 512 channels and 3x3 kernel size and then we have two parallel convolutional layers using a 1x11x1 kernel, whose number of channels depends on the number of anchors per point.

In [6]:

```
def rpn_layer(base_layers, num_anchors):
    """Create a rpn layer
        Step1: Pass through the feature map from base layer to a 3x3 512 channels convolutional layer
                Keep the padding 'same' to preserve the feature map's size
        Step2: Pass the step1 to two (1,1) convolutional layer to replace the fully connected layer
                classification layer: num_anchors (9 in here) channels for 0, 1 sigmoid activation output
                regression layer: num_anchors*4 (36 in here) channels for computing the regression of bboxes with
linear activation
    Args:
        base_layers: vgg in here
        num_anchors: 9 in here

    Returns:
        [x_class, x_regr, base_layers]
        x_class: classification for whether it's an object
        x_regr: bboxes regression
        base_layers: vgg in here
    """
    x = Conv2D(512, (3, 3), padding='same', activation='relu', kernel_initializer='normal', name='rpn_conv1')(bas
e_layers)

    x_class = Conv2D(num_anchors, (1, 1), activation='sigmoid', kernel_initializer='uniform', name='rpn_out_class
')(x)
    x_regr = Conv2D(num_anchors * 4, (1, 1), activation='linear', kernel_initializer='zero', name='rpn_out_regres
s')(x)

    return [x_class, x_regr, base_layers]
```

Classifier layer

For the classification layer, we output two predictions per anchor: the score of it being background (not an object) and the score of it being foreground (an actual object).

For the regression, or bounding box adjustment layer, we output 4 predictions: the deltas \Delta*{x*{center}}, \Delta*{y*{center}}, \Delta*{width}, \Delta*{height}Δxcenter,Δycenter,Δwidth,Δheight which we will apply to the anchors to get the final proposals.

Using the final proposal coordinates and their “objectness” score we then have a good set of proposals for objects.

In [7]:

```
def classifier_layer(base_layers, input_rois, num_rois, nb_classes = 4):
    """Create a classifier layer

    Args:
        base_layers: vgg
        input_rois: `(1,num_rois,4)` list of rois, with ordering (x,y,w,h)
        num_rois: number of rois to be processed in one time (4 in here)

    Returns:
        list(out_class, out_regr)
        out_class: classifier layer output
        out_regr: regression layer output
    """

    input_shape = (num_rois,7,7,512)

    pooling_regions = 7

    # out_roi_pool.shape = (1, num_rois, channels, pool_size, pool_size)
    # num_rois (4) 7x7 roi pooling
    out_roi_pool = RoiPoolingConv(pooling_regions, num_rois)([base_layers, input_rois])

    # Flatten the convlutional layer and connected to 2 FC and 2 dropout
    out = TimeDistributed(Flatten(name='flatten'))(out_roi_pool)
    out = TimeDistributed(Dense(4096, activation='relu', name='fc1'))(out)
    out = TimeDistributed(Dropout(0.5))(out)
    out = TimeDistributed(Dense(4096, activation='relu', name='fc2'))(out)
    out = TimeDistributed(Dropout(0.5))(out)

    # There are two output layer
    # out_class: softmax acivation function for classify the class name of the object
    # out_regr: linear activation function for bboxes coordinates regression
    out_class = TimeDistributed(Dense(nb_classes, activation='softmax', kernel_initializer='zero'), name='dense_c
lass_{}'.format(nb_classes))(out)
    # note: no regression target for bg class
    out_regr = TimeDistributed(Dense(4 * (nb_classes-1), activation='linear', kernel_initializer='zero'), name='d
ense_regress_{}'.format(nb_classes))(out)

    return [out_class, out_regr]
```

Calculate IoU (Intersection of Union)

IoU is to find the overlap score between anchors and ground truth bounding boxes.

In [8]:

```
def union(au, bu, area_intersection):
        area_a = (au[2] - au[0]) * (au[3] - au[1])
        area_b = (bu[2] - bu[0]) * (bu[3] - bu[1])
        area_union = area_a + area_b - area_intersection
        return area_union


def intersection(ai, bi):
        x = max(ai[0], bi[0])
        y = max(ai[1], bi[1])
        w = min(ai[2], bi[2]) - x
        h = min(ai[3], bi[3]) - y
        if w < 0 or h < 0:
                return 0
        return w*h


def iou(a, b):
        # a and b should be (x1,y1,x2,y2)

        if a[0] >= a[2] or a[1] >= a[3] or b[0] >= b[2] or b[1] >= b[3]:
                return 0.0

        area_i = intersection(a, b)
        area_u = union(a, b, area_i)

        return float(area_i) / float(area_u + 1e-6)
```

Calculate the rpn for all anchors of all images

In [9]:

```python
def calc_rpn(C, img_data, width, height, resized_width, resized_height, img_length_calc_function):

        """(Important part!) Calculate the rpn for all anchors
                If feature map has shape 38x50=1900, there are 1900x9=17100 potential anchors

        Args:
                C: config
                img_data: augmented image data
                width: original image width (e.g. 600)
                height: original image height (e.g. 800)
                resized_width: resized image width according to C.im_size (e.g. 300)
                resized_height: resized image height according to C.im_size (e.g. 400)
                img_length_calc_function: function to calculate final layer's feature map (of base model) size ac
cording to input image size

        Returns:
                y_rpn_cls: list(num_bboxes, y_is_box_valid + y_rpn_overlap)
                        y_is_box_valid: 0 or 1 (0 means the box is invalid, 1 means the box is valid)
                        y_rpn_overlap: 0 or 1 (0 means the box is not an object, 1 means the box is an object)
                y_rpn_regr: list(num_bboxes, 4*y_rpn_overlap + y_rpn_regr)
                        y_rpn_regr: x1,y1,x2,y2 bunding boxes coordinates
        """
        downscale = float(C.rpn_stride)
        anchor_sizes = C.anchor_box_scales   # 128, 256, 512
        anchor_ratios = C.anchor_box_ratios  # 1:1, 1:2*sqrt(2), 2*sqrt(2):1
        num_anchors = len(anchor_sizes) * len(anchor_ratios) # 3x3=9

        # calculate the output map size based on the network architecture
        (output_width, output_height) = img_length_calc_function(resized_width, resized_height)

        n_anchratios = len(anchor_ratios)    # 3

        # initialise empty output objectives
        y_rpn_overlap = np.zeros((output_height, output_width, num_anchors))
        y_is_box_valid = np.zeros((output_height, output_width, num_anchors))
        y_rpn_regr = np.zeros((output_height, output_width, num_anchors * 4))

        num_bboxes = len(img_data['bboxes'])

        num_anchors_for_bbox = np.zeros(num_bboxes).astype(int)
        best_anchor_for_bbox = -1*np.ones((num_bboxes, 4)).astype(int)
        best_iou_for_bbox = np.zeros(num_bboxes).astype(np.float32)
        best_x_for_bbox = np.zeros((num_bboxes, 4)).astype(int)
        best_dx_for_bbox = np.zeros((num_bboxes, 4)).astype(np.float32)

        # get the GT box coordinates, and resize to account for image resizing
        gta = np.zeros((num_bboxes, 4))
        for bbox_num, bbox in enumerate(img_data['bboxes']):
                # get the GT box coordinates, and resize to account for image resizing
                gta[bbox_num, 0] = bbox['x1'] * (resized_width / float(width))
                gta[bbox_num, 1] = bbox['x2'] * (resized_width / float(width))
                gta[bbox_num, 2] = bbox['y1'] * (resized_height / float(height))
                gta[bbox_num, 3] = bbox['y2'] * (resized_height / float(height))

        # rpn ground truth

        for anchor_size_idx in range(len(anchor_sizes)):
                for anchor_ratio_idx in range(n_anchratios):
                        anchor_x = anchor_sizes[anchor_size_idx] * anchor_ratios[anchor_ratio_idx][0]
                        anchor_y = anchor_sizes[anchor_size_idx] * anchor_ratios[anchor_ratio_idx][1]

                        for ix in range(output_width):
                                # x-coordinates of the current anchor box
                                x1_anc = downscale * (ix + 0.5) - anchor_x / 2
                                x2_anc = downscale * (ix + 0.5) + anchor_x / 2

                                # ignore boxes that go across image boundaries
                                if x1_anc < 0 or x2_anc > resized_width:
                                        continue

                                for jy in range(output_height):

                                        # y-coordinates of the current anchor box
                                        y1_anc = downscale * (jy + 0.5) - anchor_y / 2
                                        y2_anc = downscale * (jy + 0.5) + anchor_y / 2

                                        # ignore boxes that go across image boundaries
                                        if y1_anc < 0 or y2_anc > resized_height:
                                                continue

                                        # bbox_type indicates whether an anchor should be a target
                                        # Initialize with 'negative'
                                        bbox_type = 'neg'
                                        # this is the best IOU for the (x,y) coord and the current anchor

                                        # note that this is different from the best IOU for a GT bbox
                                        best_iou_for_loc = 0.0

                                        for bbox_num in range(num_bboxes):

                                                # get IOU of the current GT box and the current anchor box
                                                curr_iou = iou([gta[bbox_num, 0], gta[bbox_num, 2], gta[bbox_num,
1], gta[bbox_num, 3]], [x1_anc, y1_anc, x2_anc, y2_anc])
                                                # calculate the regression targets if they will be needed
                                                if curr_iou > best_iou_for_bbox[bbox_num] or curr_iou > C.rpn_max
_overlap:
                                                        cx = (gta[bbox_num, 0] + gta[bbox_num, 1]) / 2.0
                                                        cy = (gta[bbox_num, 2] + gta[bbox_num, 3]) / 2.0
                                                        cxa = (x1_anc + x2_anc)/2.0
                                                        cya = (y1_anc + y2_anc)/2.0

                                                        # x,y are the center point of ground-truth bbox
                                                        # xa,ya are the center point of anchor bbox (xa=downscale
* (ix + 0.5); ya=downscale * (iy+0.5))
                                                        # w,h are the width and height of ground-truth bbox
                                                        # wa,ha are the width and height of anchor bboxe
                                                        # tx = (x - xa) / wa
                                                        # ty = (y - ya) / ha
                                                        # tw = log(w / wa)
                                                        # th = log(h / ha)
                                                        tx = (cx - cxa) / (x2_anc - x1_anc)
                                                        ty = (cy - cya) / (y2_anc - y1_anc)
                                                        tw = np.log((gta[bbox_num, 1] - gta[bbox_num, 0]) / (x2_a
nc - x1_anc))
                                                        th = np.log((gta[bbox_num, 3] - gta[bbox_num, 2]) / (y2_a
nc - y1_anc))

                                                if img_data['bboxes'][bbox_num]['class'] != 'bg':

                                                        # all GT boxes should be mapped to an anchor box, so we k
eep track of which anchor box was best
                                                        if curr_iou > best_iou_for_bbox[bbox_num]:
                                                                best_anchor_for_bbox[bbox_num] = [jy, ix, anchor_
ratio_idx, anchor_size_idx]
                                                                best_iou_for_bbox[bbox_num] = curr_iou
                                                                best_x_for_bbox[bbox_num,:] = [x1_anc, x2_anc, y1
_anc, y2_anc]
                                                                best_dx_for_bbox[bbox_num,:] = [tx, ty, tw, th]

                                                        # we set the anchor to positive if the IOU is >0.7 (it do
es not matter if there was another better box, it just indicates overlap)
                                                        if curr_iou > C.rpn_max_overlap:
                                                                bbox_type = 'pos'
                                                                num_anchors_for_bbox[bbox_num] += 1
                                                                # we update the regression layer target if this I
OU is the best for the current (x,y) and anchor position
                                                                if curr_iou > best_iou_for_loc:
                                                                        best_iou_for_loc = curr_iou
                                                                        best_regr = (tx, ty, tw, th)

                                                        # if the IOU is >0.3 and <0.7, it is ambiguous and no inc
luded in the objective
                                                        if C.rpn_min_overlap < curr_iou < C.rpn_max_overlap:
                                                                # gray zone between neg and pos
                                                                if bbox_type != 'pos':
                                                                        bbox_type = 'neutral'

                                        # turn on or off outputs depending on IOUs
                                        if bbox_type == 'neg':
                                                y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_s
ize_idx] = 1
                                                y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_si
ze_idx] = 0
                                        elif bbox_type == 'neutral':
                                                y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_s
ize_idx] = 0
                                                y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_si
ze_idx] = 0
                                        elif bbox_type == 'pos':
                                                y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_s
ize_idx] = 1
                                                y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_si
ze_idx] = 1
                                                start = 4 * (anchor_ratio_idx + n_anchratios * anchor_size_idx)
                                                y_rpn_regr[jy, ix, start:start+4] = best_regr

        # we ensure that every bbox has at least one positive RPN region
        for idx in range(num_anchors_for_bbox.shape[0]):
                if num_anchors_for_bbox[idx] == 0:
                        # no box with an IOU greater than zero ...
                        if best_anchor_for_bbox[idx, 0] == -1:
                                continue
                        y_is_box_valid[
                                best_anchor_for_bbox[idx,0], best_anchor_for_bbox[idx,1], best_anchor_for_bbox[id
x,2] + n_anchratios *
                                best_anchor_for_bbox[idx,3]] = 1
                        y_rpn_overlap[
                                best_anchor_for_bbox[idx,0], best_anchor_for_bbox[idx,1], best_anchor_for_bbox[id
x,2] + n_anchratios *
                                best_anchor_for_bbox[idx,3]] = 1
                        start = 4 * (best_anchor_for_bbox[idx,2] + n_anchratios * best_anchor_for_bbox[idx,3])
                        y_rpn_regr[
                                best_anchor_for_bbox[idx,0], best_anchor_for_bbox[idx,1], start:start+4] = best_d
x_for_bbox[idx, :]

        y_rpn_overlap = np.transpose(y_rpn_overlap, (2, 0, 1))
        y_rpn_overlap = np.expand_dims(y_rpn_overlap, axis=0)

        y_is_box_valid = np.transpose(y_is_box_valid, (2, 0, 1))
        y_is_box_valid = np.expand_dims(y_is_box_valid, axis=0)

        y_rpn_regr = np.transpose(y_rpn_regr, (2, 0, 1))
        y_rpn_regr = np.expand_dims(y_rpn_regr, axis=0)

        pos_locs = np.where(np.logical_and(y_rpn_overlap[0, :, :, :] == 1, y_is_box_valid[0, :, :, :] == 1))
        neg_locs = np.where(np.logical_and(y_rpn_overlap[0, :, :, :] == 0, y_is_box_valid[0, :, :, :] == 1))

        num_pos = len(pos_locs[0])

        # one issue is that the RPN has many more negative than positive regions, so we turn off some of the nega
tive
        # regions. We also limit it to 256 regions.
        num_regions = 256

        if len(pos_locs[0]) > num_regions/2:
                val_locs = random.sample(range(len(pos_locs[0])), len(pos_locs[0]) - num_regions/2)
                y_is_box_valid[0, pos_locs[0][val_locs], pos_locs[1][val_locs], pos_locs[2][val_locs]] = 0
                num_pos = num_regions/2

        if len(neg_locs[0]) + num_pos > num_regions:
                val_locs = random.sample(range(len(neg_locs[0])), len(neg_locs[0]) - num_pos)
                y_is_box_valid[0, neg_locs[0][val_locs], neg_locs[1][val_locs], neg_locs[2][val_locs]] = 0

        y_rpn_cls = np.concatenate([y_is_box_valid, y_rpn_overlap], axis=1)
        y_rpn_regr = np.concatenate([np.repeat(y_rpn_overlap, 4, axis=1), y_rpn_regr], axis=1)

        return np.copy(y_rpn_cls), np.copy(y_rpn_regr), num_pos
```

Get new image size and augment the image

In [10]:

```
def get_new_img_size(width, height, img_min_side=300):
        if width <= height:
                f = float(img_min_side) / width
                resized_height = int(f * height)
                resized_width = img_min_side
        else:
                f = float(img_min_side) / height
                resized_width = int(f * width)
                resized_height = img_min_side

        return resized_width, resized_height

def augment(img_data, config, augment=True):
        assert 'filepath' in img_data
        assert 'bboxes' in img_data
        assert 'width' in img_data
        assert 'height' in img_data

        img_data_aug = copy.deepcopy(img_data)

        img = cv2.imread(img_data_aug['filepath'])

        if augment:
                rows, cols = img.shape[:2]

                if config.use_horizontal_flips and np.random.randint(0, 2) == 0:
                        img = cv2.flip(img, 1)
                        for bbox in img_data_aug['bboxes']:
                                x1 = bbox['x1']
                                x2 = bbox['x2']
                                bbox['x2'] = cols - x1
                                bbox['x1'] = cols - x2

                if config.use_vertical_flips and np.random.randint(0, 2) == 0:
                        img = cv2.flip(img, 0)
                        for bbox in img_data_aug['bboxes']:
                                y1 = bbox['y1']
                                y2 = bbox['y2']
                                bbox['y2'] = rows - y1
                                bbox['y1'] = rows - y2

                if config.rot_90:
                        angle = np.random.choice([0,90,180,270],1)[0]
                        if angle == 270:
                                img = np.transpose(img, (1,0,2))
                                img = cv2.flip(img, 0)
                        elif angle == 180:
                                img = cv2.flip(img, -1)
                        elif angle == 90:
                                img = np.transpose(img, (1,0,2))
                                img = cv2.flip(img, 1)
                        elif angle == 0:
                                pass

                        for bbox in img_data_aug['bboxes']:
                                x1 = bbox['x1']
                                x2 = bbox['x2']
                                y1 = bbox['y1']
                                y2 = bbox['y2']
                                if angle == 270:
                                        bbox['x1'] = y1
                                        bbox['x2'] = y2
                                        bbox['y1'] = cols - x2
                                        bbox['y2'] = cols - x1
                                elif angle == 180:
                                        bbox['x2'] = cols - x1
                                        bbox['x1'] = cols - x2
                                        bbox['y2'] = rows - y1
                                        bbox['y1'] = rows - y2
                                elif angle == 90:
                                        bbox['x1'] = rows - y2
                                        bbox['x2'] = rows - y1
                                        bbox['y1'] = x1
                                        bbox['y2'] = x2
                                elif angle == 0:
                                        pass

        img_data_aug['width'] = img.shape[1]
        img_data_aug['height'] = img.shape[0]
        return img_data_aug, img
```

Generate the ground_truth anchors

In [11]:

```
def get_anchor_gt(all_img_data, C, img_length_calc_function, mode='train'):
        """ Yield the ground-truth anchors as Y (labels)

        Args:
                all_img_data: list(filepath, width, height, list(bboxes))
                C: config
                img_length_calc_function: function to calculate final layer's feature map (of base model) size ac
cording to input image size
                mode: 'train' or 'test'; 'train' mode need augmentation

        Returns:
                x_img: image data after resized and scaling (smallest size = 300px)
                Y: [y_rpn_cls, y_rpn_regr]
                img_data_aug: augmented image data (original image with augmentation)
                debug_img: show image for debug
                num_pos: show number of positive anchors for debug
        """
        while True:

                for img_data in all_img_data:
                        try:

                                # read in image, and optionally add augmentation

                                if mode == 'train':
                                        img_data_aug, x_img = augment(img_data, C, augment=True)
                                else:
                                        img_data_aug, x_img = augment(img_data, C, augment=False)

                                (width, height) = (img_data_aug['width'], img_data_aug['height'])
                                (rows, cols, _) = x_img.shape

                                assert cols == width
                                assert rows == height

                                # get image dimensions for resizing
                                (resized_width, resized_height) = get_new_img_size(width, height, C.im_size)

                                # resize the image so that smalles side is length = 300px
                                x_img = cv2.resize(x_img, (resized_width, resized_height), interpolation=cv2.INTE
R_CUBIC)
                                debug_img = x_img.copy()

                                try:
                                        y_rpn_cls, y_rpn_regr, num_pos = calc_rpn(C, img_data_aug, width, height,
resized_width, resized_height, img_length_calc_function)
                                except:
                                        continue

                                # Zero-center by mean pixel, and preprocess image

                                x_img = x_img[:,:, (2, 1, 0)]  # BGR -> RGB
                                x_img = x_img.astype(np.float32)
                                x_img[:, :, 0] -= C.img_channel_mean[0]
                                x_img[:, :, 1] -= C.img_channel_mean[1]
                                x_img[:, :, 2] -= C.img_channel_mean[2]
                                x_img /= C.img_scaling_factor

                                x_img = np.transpose(x_img, (2, 0, 1))
                                x_img = np.expand_dims(x_img, axis=0)

                                y_rpn_regr[:, y_rpn_regr.shape[1]//2:, :, :] *= C.std_scaling

                                x_img = np.transpose(x_img, (0, 2, 3, 1))
                                y_rpn_cls = np.transpose(y_rpn_cls, (0, 2, 3, 1))
                                y_rpn_regr = np.transpose(y_rpn_regr, (0, 2, 3, 1))

                                yield np.copy(x_img), [np.copy(y_rpn_cls), np.copy(y_rpn_regr)], img_data_aug, de
bug_img, num_pos

                        except Exception as e:
                                print(e)
                                continue
```

Define loss functions for all four outputs

In [12]:

```
lambda_rpn_regr = 1.0
lambda_rpn_class = 1.0

lambda_cls_regr = 1.0
lambda_cls_class = 1.0

epsilon = 1e-4
```

In [13]:

```
def rpn_loss_regr(num_anchors):
    """Loss function for rpn regression
    Args:
        num_anchors: number of anchors (9 in here)
    Returns:
        Smooth L1 loss function
                           0.5*x*x (if x_abs < 1)
                           x_abx - 0.5 (otherwise)
    """
    def rpn_loss_regr_fixed_num(y_true, y_pred):

        # x is the difference between true value and predicted vaue
        x = y_true[:, :, :, 4 * num_anchors:] - y_pred

        # absolute value of x
        x_abs = K.abs(x)

        # If x_abs <= 1.0, x_bool = 1
        x_bool = K.cast(K.less_equal(x_abs, 1.0), tf.float32)

        return lambda_rpn_regr * K.sum(
            y_true[:, :, :, :4 * num_anchors] * (x_bool * (0.5 * x * x) + (1 - x_bool) * (x_abs - 0.5))) / K.sum(
epsilon + y_true[:, :, :, :4 * num_anchors])

    return rpn_loss_regr_fixed_num


def rpn_loss_cls(num_anchors):
    """Loss function for rpn classification
    Args:
        num_anchors: number of anchors (9 in here)
        y_true[:, :, :, :9]: [0,1,0,0,0,0,0,1,0] means only the second and the eighth box is valid which contains
pos or neg anchor => isValid
        y_true[:, :, :, 9:]: [0,1,0,0,0,0,0,0,0] means the second box is pos and eighth box is negative
    Returns:
        lambda * sum((binary_crossentropy(isValid*y_pred,y_true))) / N
    """
    def rpn_loss_cls_fixed_num(y_true, y_pred):

            return lambda_rpn_class * K.sum(y_true[:, :, :, :num_anchors] * K.binary_crossentropy(y_pred[:, :, :,
:], y_true[:, :, :, num_anchors:])) / K.sum(epsilon + y_true[:, :, :, :num_anchors])

    return rpn_loss_cls_fixed_num


def class_loss_regr(num_classes):
    """Loss function for rpn regression
    Args:
        num_anchors: number of anchors (9 in here)
    Returns:
        Smooth L1 loss function
                           0.5*x*x (if x_abs < 1)
                           x_abx - 0.5 (otherwise)
    """
    def class_loss_regr_fixed_num(y_true, y_pred):
        x = y_true[:, :, 4*num_classes:] - y_pred
        x_abs = K.abs(x)
        x_bool = K.cast(K.less_equal(x_abs, 1.0), 'float32')
        return lambda_cls_regr * K.sum(y_true[:, :, :4*num_classes] * (x_bool * (0.5 * x * x) + (1 - x_bool) * (x
_abs - 0.5))) / K.sum(epsilon + y_true[:, :, :4*num_classes])
    return class_loss_regr_fixed_num


def class_loss_cls(y_true, y_pred):
    return lambda_cls_class * K.mean(categorical_crossentropy(y_true[0, :, :], y_pred[0, :, :]))
```

In [14]:

```
def non_max_suppression_fast(boxes, probs, overlap_thresh=0.9, max_boxes=300):
    # code used from here: http://www.pyimagesearch.com/2015/02/16/faster-non-maximum-suppression-python/
    # if there are no boxes, return an empty list
```

```python
    # Process explanation:
    #   Step 1: Sort the probs list
    #   Step 2: Find the larget prob 'Last' in the list and save it to the pick list
    #   Step 3: Calculate the IoU with 'Last' box and other boxes in the list. If the IoU is larger than overlap_
threshold, delete the box from list
    #   Step 4: Repeat step 2 and step 3 until there is no item in the probs list
    if len(boxes) == 0:
        return []

    # grab the coordinates of the bounding boxes
    x1 = boxes[:, 0]
    y1 = boxes[:, 1]
    x2 = boxes[:, 2]
    y2 = boxes[:, 3]

    np.testing.assert_array_less(x1, x2)
    np.testing.assert_array_less(y1, y2)

    # if the bounding boxes integers, convert them to floats --
    # this is important since we'll be doing a bunch of divisions
    if boxes.dtype.kind == "i":
        boxes = boxes.astype("float")

    # initialize the list of picked indexes
    pick = []

    # calculate the areas
    area = (x2 - x1) * (y2 - y1)

    # sort the bounding boxes
    idxs = np.argsort(probs)

    # keep looping while some indexes still remain in the indexes
    # list
    while len(idxs) > 0:
        # grab the last index in the indexes list and add the
        # index value to the list of picked indexes
        last = len(idxs) - 1
        i = idxs[last]
        pick.append(i)

        # find the intersection

        xx1_int = np.maximum(x1[i], x1[idxs[:last]])
        yy1_int = np.maximum(y1[i], y1[idxs[:last]])
        xx2_int = np.minimum(x2[i], x2[idxs[:last]])
        yy2_int = np.minimum(y2[i], y2[idxs[:last]])

        ww_int = np.maximum(0, xx2_int - xx1_int)
        hh_int = np.maximum(0, yy2_int - yy1_int)

        area_int = ww_int * hh_int

        # find the union
        area_union = area[i] + area[idxs[:last]] - area_int

        # compute the ratio of overlap
        overlap = area_int/(area_union + 1e-6)

        # delete all indexes from the index list that have
        idxs = np.delete(idxs, np.concatenate(([last],
            np.where(overlap > overlap_thresh)[0])))

        if len(pick) >= max_boxes:
            break

    # return only the bounding boxes that were picked using the integer data type
    boxes = boxes[pick].astype("int")
    probs = probs[pick]
    return boxes, probs

def apply_regr_np(X, T):
    """Apply regression layer to all anchors in one feature map

    Args:
        X: shape=(4, 18, 25) the current anchor type for all points in the feature map
        T: regression layer shape=(4, 18, 25)

    Returns:
        X: regressed position and size for current anchor
    """
    try:
        x = X[0, :, :]

        y = X[1, :, :]
        w = X[2, :, :]
        h = X[3, :, :]

        tx = T[0, :, :]
        ty = T[1, :, :]
        tw = T[2, :, :]
        th = T[3, :, :]

        cx = x + w/2.
        cy = y + h/2.
        cx1 = tx * w + cx
        cy1 = ty * h + cy

        w1 = np.exp(tw.astype(np.float64)) * w
        h1 = np.exp(th.astype(np.float64)) * h
        x1 = cx1 - w1/2.
        y1 = cy1 - h1/2.

        x1 = np.round(x1)
        y1 = np.round(y1)
        w1 = np.round(w1)
        h1 = np.round(h1)
        return np.stack([x1, y1, w1, h1])
    except Exception as e:
        print(e)
        return X

def apply_regr(x, y, w, h, tx, ty, tw, th):
    # Apply regression to x, y, w and h
    try:
        cx = x + w/2.
        cy = y + h/2.
        cx1 = tx * w + cx
        cy1 = ty * h + cy
        w1 = math.exp(tw) * w
        h1 = math.exp(th) * h
        x1 = cx1 - w1/2.
        y1 = cy1 - h1/2.
        x1 = int(round(x1))
        y1 = int(round(y1))
        w1 = int(round(w1))
        h1 = int(round(h1))

        return x1, y1, w1, h1

    except ValueError:
        return x, y, w, h
    except OverflowError:
        return x, y, w, h
    except Exception as e:
        print(e)
        return x, y, w, h

def calc_iou(R, img_data, C, class_mapping):
    """Converts from (x1,y1,x2,y2) to (x,y,w,h) format

    Args:
        R: bboxes, probs
    """
    bboxes = img_data['bboxes']
    (width, height) = (img_data['width'], img_data['height'])
    # get image dimensions for resizing
    (resized_width, resized_height) = get_new_img_size(width, height, C.im_size)

    gta = np.zeros((len(bboxes), 4))

    for bbox_num, bbox in enumerate(bboxes):
        # get the GT box coordinates, and resize to account for image resizing
        # gta[bbox_num, 0] = (40 * (600 / 800)) / 16 = int(round(1.875)) = 2 (x in feature map)
        gta[bbox_num, 0] = int(round(bbox['x1'] * (resized_width / float(width))/C.rpn_stride))
        gta[bbox_num, 1] = int(round(bbox['x2'] * (resized_width / float(width))/C.rpn_stride))
        gta[bbox_num, 2] = int(round(bbox['y1'] * (resized_height / float(height))/C.rpn_stride))
        gta[bbox_num, 3] = int(round(bbox['y2'] * (resized_height / float(height))/C.rpn_stride))

    x_roi = []
    y_class_num = []
    y_class_regr_coords = []
    y_class_regr_label = []
    IoUs = [] # for debugging only

    # R.shape[0]: number of bboxes (=300 from non_max_suppression)
    for ix in range(R.shape[0]):

        (x1, y1, x2, y2) = R[ix, :]
        x1 = int(round(x1))
        y1 = int(round(y1))
        x2 = int(round(x2))
        y2 = int(round(y2))

        best_iou = 0.0
        best_bbox = -1
        # Iterate through all the ground-truth bboxes to calculate the iou
        for bbox_num in range(len(bboxes)):
            curr_iou = iou([gta[bbox_num, 0], gta[bbox_num, 2], gta[bbox_num, 1], gta[bbox_num, 3]], [x1, y1, x2,
y2])

            # Find out the corresponding ground-truth bbox_num with larget iou
            if curr_iou > best_iou:
                best_iou = curr_iou
                best_bbox = bbox_num

        if best_iou < C.classifier_min_overlap:
                continue
        else:
            w = x2 - x1
            h = y2 - y1
            x_roi.append([x1, y1, w, h])
            IoUs.append(best_iou)

            if C.classifier_min_overlap <= best_iou < C.classifier_max_overlap:
                # hard negative example
                cls_name = 'bg'
            elif C.classifier_max_overlap <= best_iou:
                cls_name = bboxes[best_bbox]['class']
                cxg = (gta[best_bbox, 0] + gta[best_bbox, 1]) / 2.0
                cyg = (gta[best_bbox, 2] + gta[best_bbox, 3]) / 2.0

                cx = x1 + w / 2.0
                cy = y1 + h / 2.0

                tx = (cxg - cx) / float(w)
                ty = (cyg - cy) / float(h)
                tw = np.log((gta[best_bbox, 1] - gta[best_bbox, 0]) / float(w))
                th = np.log((gta[best_bbox, 3] - gta[best_bbox, 2]) / float(h))
            else:
                print('roi = {}'.format(best_iou))
                raise RuntimeError

        class_num = class_mapping[cls_name]
        class_label = len(class_mapping) * [0]
        class_label[class_num] = 1
        y_class_num.append(copy.deepcopy(class_label))
        coords = [0] * 4 * (len(class_mapping) - 1)
        labels = [0] * 4 * (len(class_mapping) - 1)
        if cls_name != 'bg':
            label_pos = 4 * class_num
            sx, sy, sw, sh = C.classifier_regr_std
            coords[label_pos:4+label_pos] = [sx*tx, sy*ty, sw*tw, sh*th]
            labels[label_pos:4+label_pos] = [1, 1, 1, 1]
            y_class_regr_coords.append(copy.deepcopy(coords))
            y_class_regr_label.append(copy.deepcopy(labels))
        else:
            y_class_regr_coords.append(copy.deepcopy(coords))
            y_class_regr_label.append(copy.deepcopy(labels))

    if len(x_roi) == 0:
        return None, None, None, None

    # bboxes that iou > C.classifier_min_overlap for all gt bboxes in 300 non_max_suppression bboxes
    X = np.array(x_roi)
    # one hot code for bboxes from above => x_roi (X)
    Y1 = np.array(y_class_num)
    # corresponding labels and corresponding gt bboxes
    Y2 = np.concatenate([np.array(y_class_regr_label),np.array(y_class_regr_coords)],axis=1)

    return np.expand_dims(X, axis=0), np.expand_dims(Y1, axis=0), np.expand_dims(Y2, axis=0), IoUs
```

In [15]:

```
def rpn_to_roi(rpn_layer, regr_layer, C, dim_ordering, use_regr=True, max_boxes=300,overlap_thresh=0.9):
        """Convert rpn layer to roi bboxes

        Args: (num_anchors = 9)
                rpn_layer: output layer for rpn classification
                        shape (1, feature_map.height, feature_map.width, num_anchors)
```

```
                    Might be (1, 18, 25, 18) if resized image is 400 width and 300

            regr_layer: output layer for rpn regression
                    shape (1, feature_map.height, feature_map.width, num_anchors)
                    Might be (1, 18, 25, 72) if resized image is 400 width and 300
            C: config
            use_regr: Wether to use bboxes regression in rpn
            max_boxes: max bboxes number for non-max-suppression (NMS)
            overlap_thresh: If iou in NMS is larger than this threshold, drop the box

    Returns:
            result: boxes from non-max-suppression (shape=(300, 4))
                    boxes: coordinates for bboxes (on the feature map)
    """
    regr_layer = regr_layer / C.std_scaling

    anchor_sizes = C.anchor_box_scales   # (3 in here)
    anchor_ratios = C.anchor_box_ratios  # (3 in here)

    assert rpn_layer.shape[0] == 1

    (rows, cols) = rpn_layer.shape[1:3]

    curr_layer = 0

    # A.shape = (4, feature_map.height, feature_map.width, num_anchors)
    # Might be (4, 18, 25, 18) if resized image is 400 width and 300
    # A is the coordinates for 9 anchors for every point in the feature map
    # => all 18x25x9=4050 anchors cooridnates
    A = np.zeros((4, rpn_layer.shape[1], rpn_layer.shape[2], rpn_layer.shape[3]))

    for anchor_size in anchor_sizes:
            for anchor_ratio in anchor_ratios:
                    # anchor_x = (128 * 1) / 16 = 8  => width of current anchor
                    # anchor_y = (128 * 2) / 16 = 16 => height of current anchor
                    anchor_x = (anchor_size * anchor_ratio[0])/C.rpn_stride
                    anchor_y = (anchor_size * anchor_ratio[1])/C.rpn_stride

                    # curr_layer: 0~8 (9 anchors)
                    # the Kth anchor of all position in the feature map (9th in total)
                    regr = regr_layer[0, :, :, 4 * curr_layer:4 * curr_layer + 4] # shape => (18, 25, 4)
                    regr = np.transpose(regr, (2, 0, 1)) # shape => (4, 18, 25)

                    # Create 18x25 mesh grid
                    # For every point in x, there are all the y points and vice versa
                    # X.shape = (18, 25)
                    # Y.shape = (18, 25)
                    X, Y = np.meshgrid(np.arange(cols),np. arange(rows))

                    # Calculate anchor position and size for each feature map point
                    A[0, :, :, curr_layer] = X - anchor_x/2 # Top left x coordinate
                    A[1, :, :, curr_layer] = Y - anchor_y/2 # Top left y coordinate
                    A[2, :, :, curr_layer] = anchor_x       # width of current anchor
                    A[3, :, :, curr_layer] = anchor_y       # height of current anchor

                    # Apply regression to x, y, w and h if there is rpn regression layer
                    if use_regr:
                            A[:, :, :, curr_layer] = apply_regr_np(A[:, :, :, curr_layer], regr)

                    # Avoid width and height exceeding 1
                    A[2, :, :, curr_layer] = np.maximum(1, A[2, :, :, curr_layer])
                    A[3, :, :, curr_layer] = np.maximum(1, A[3, :, :, curr_layer])

                    # Convert (x, y , w, h) to (x1, y1, x2, y2)
                    # x1, y1 is top left coordinate
                    # x2, y2 is bottom right coordinate
                    A[2, :, :, curr_layer] += A[0, :, :, curr_layer]
                    A[3, :, :, curr_layer] += A[1, :, :, curr_layer]

                    # Avoid bboxes drawn outside the feature map
                    A[0, :, :, curr_layer] = np.maximum(0, A[0, :, :, curr_layer])
                    A[1, :, :, curr_layer] = np.maximum(0, A[1, :, :, curr_layer])
                    A[2, :, :, curr_layer] = np.minimum(cols-1, A[2, :, :, curr_layer])
                    A[3, :, :, curr_layer] = np.minimum(rows-1, A[3, :, :, curr_layer])

                    curr_layer += 1

    all_boxes = np.reshape(A.transpose((0, 3, 1, 2)), (4, -1)).transpose((1, 0))  # shape=(4050, 4)
    all_probs = rpn_layer.transpose((0, 3, 1, 2)).reshape((-1))                   # shape=(4050,)

    x1 = all_boxes[:, 0]
    y1 = all_boxes[:, 1]
    x2 = all_boxes[:, 2]
    y2 = all_boxes[:, 3]
```

```
        # Find out the bboxes which is illegal and delete them from bboxes list
        idxs = np.where((x1 - x2 >= 0) | (y1 - y2 >= 0))

        all_boxes = np.delete(all_boxes, idxs, 0)
        all_probs = np.delete(all_probs, idxs, 0)

        # Apply non_max_suppression
        # Only extract the bboxes. Don't need rpn probs in the later process
        result = non_max_suppression_fast(all_boxes, all_probs, overlap_thresh=overlap_thresh, max_boxes=max_boxe
s)[0]

        return result
```

Start training

In [16]:

```
base_path = 'Dataset/Open Images Dataset v4 (Bounding Boxes)/'

train_path =  'Dataset/Open Images Dataset v4 (Bounding Boxes)/annotation.txt' # Training data (annotation file)

num_rois = 4 # Number of RoIs to process at once.

# Augmentation flag
horizontal_flips = True # Augment with horizontal flips in training.
vertical_flips = True   # Augment with vertical flips in training.
rot_90 = True           # Augment with 90 degree rotations in training.

output_weight_path = os.path.join(base_path, 'model/model_frcnn_vgg.hdf5')

record_path = os.path.join(base_path, 'model/record.csv') # Record data (used to save the losses, classification
accuracy and mean average precision)

base_weight_path = os.path.join(base_path, 'model/vgg16_weights_tf_dim_ordering_tf_kernels.h5')

config_output_filename = os.path.join(base_path, 'model_vgg_config.pickle')
```

In [17]:

```
# Create the config
C = Config()

C.use_horizontal_flips = horizontal_flips
C.use_vertical_flips = vertical_flips
C.rot_90 = rot_90

C.record_path = record_path
C.model_path = output_weight_path
C.num_rois = num_rois

C.base_net_weights = base_weight_path
```

In [18]:

```
#--------------------------------------------------------#
# This step will spend some time to load the data        #
#--------------------------------------------------------#
st = time.time()
train_imgs, classes_count, class_mapping = get_data(train_path)
print()
print('Spend %0.2f mins to load the data' % ((time.time()-st)/60) )
```

```
Parsing annotation files
idx=80113
Spend 1.29 mins to load the data
```

In [19]:

```
if 'bg' not in classes_count:
        classes_count['bg'] = 0
        class_mapping['bg'] = len(class_mapping)
# e.g.
#    classes_count: {'Car': 2383, 'Mobile phone': 1108, 'Person': 3745, 'bg': 0}
#    class_mapping: {'Person': 0, 'Car': 1, 'Mobile phone': 2, 'bg': 3}
C.class_mapping = class_mapping

print('Training images per class:')
pprint.pprint(classes_count)
print('Num classes (including bg) = {}'.format(len(classes_count)))
print(class_mapping)

# Save the configuration
with open(config_output_filename, 'wb') as config_f:
        pickle.dump(C,config_f)
        print('Config has been written to {}, and can be loaded when testing to ensure correct results'.format(co
nfig_output_filename))
```

```
Training images per class:
{'bg': 0,
 'difficult': 441,
 'gametocyte': 144,
 'leukocyte': 103,
 'red blood cell': 77420,
 'ring': 353,
 'schizont': 179,
 'trophozoite': 1473}
Num classes (including bg) = 8
{'red blood cell': 0, 'trophozoite': 1, 'schizont': 2, 'difficult': 3, 'ring': 4, 'leukocyte': 5, 'g
ametocyte': 6, 'bg': 7}
Config has been written to Dataset/Open Images Dataset v4 (Bounding Boxes)/model_vgg_config.pickle,
and can be loaded when testing to ensure correct results
```

In [20]:

```
# Shuffle the images with seed
random.seed(1)
random.shuffle(train_imgs)

print('Num train samples (images) {}'.format(len(train_imgs)))
```

```
Num train samples (images) 1208
```

In [21]:

```
# Get train data generator which generate X, Y, image_data
data_gen_train = get_anchor_gt(train_imgs, C, get_img_output_length, mode='train')
```

Explore 'data_gen_train'

data_gen_train is an **generator**, so we get the data by calling **next(data_gen_train)**

In [22]:

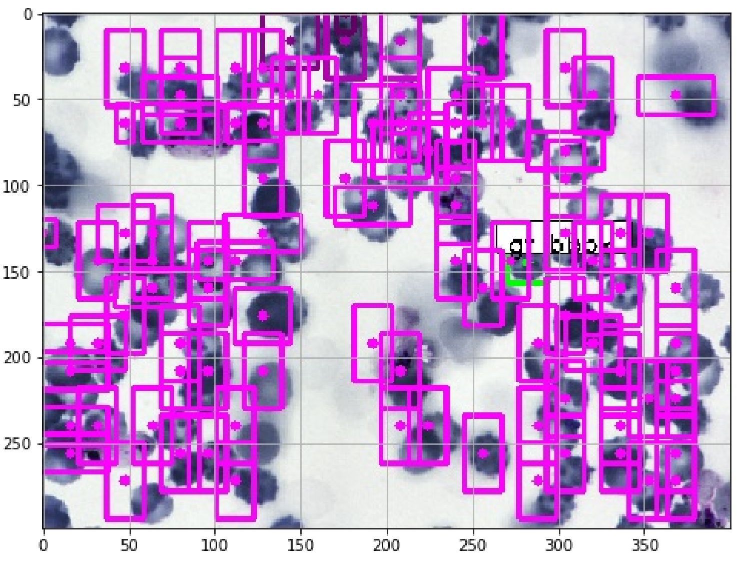
```
X, Y, image_data, debug_img, debug_num_pos = next(data_gen_train)
```

The below code will create all possible positive anchors and a ground truth box

In [23]:

```
print('Original image: height=%d width=%d'%(image_data['height'], image_data['width']))
print('Resized image:  height=%d width=%d C.im_size=%d'%(X.shape[1], X.shape[2], C.im_size))
print('Feature map size: height=%d width=%d C.rpn_stride=%d'%(Y[0].shape[1], Y[0].shape[2], C.rpn_stride))
print(X.shape)
print(str(len(Y))+" includes 'y_rpn_cls' and 'y_rpn_regr'")
print('Shape of y_rpn_cls {}'.format(Y[0].shape))
print('Shape of y_rpn_regr {}'.format(Y[1].shape))
print(image_data)

print('Number of positive anchors for this image: %d' % (debug_num_pos))
if debug_num_pos==0:
    gt_x1, gt_x2 = image_data['bboxes'][0]['x1']*(X.shape[2]/image_data['height']), image_data['bboxes'][0]['x2']
*(X.shape[2]/image_data['height'])
    gt_y1, gt_y2 = image_data['bboxes'][0]['y1']*(X.shape[1]/image_data['width']), image_data['bboxes'][0]['y2']*
(X.shape[1]/image_data['width'])
    gt_x1, gt_y1, gt_x2, gt_y2 = int(gt_x1), int(gt_y1), int(gt_x2), int(gt_y2)

    img = debug_img.copy()
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    color = (0, 255, 0)
    cv2.putText(img, 'gt bbox', (gt_x1, gt_y1-5), cv2.FONT_HERSHEY_DUPLEX, 0.7, color, 1)
    cv2.rectangle(img, (gt_x1, gt_y1), (gt_x2, gt_y2), color, 2)
    cv2.circle(img, (int((gt_x1+gt_x2)/2), int((gt_y1+gt_y2)/2)), 3, color, -1)

    plt.grid()
    plt.imshow(img)
    plt.show()
else:
    cls = Y[0][0]
    pos_cls = np.where(cls==1)
    print(pos_cls)
    regr = Y[1][0]
    pos_regr = np.where(regr==1)
    print(pos_regr)
    print('y_rpn_cls for possible pos anchor: {}'.format(cls[pos_cls[0][0],pos_cls[1][0],:]))
    print('y_rpn_regr for positive anchor: {}'.format(regr[pos_regr[0][0],pos_regr[1][0],:]))

    gt_x1, gt_x2 = image_data['bboxes'][0]['x1']*(X.shape[2]/image_data['width']), image_data['bboxes'][0]['x2']*
(X.shape[2]/image_data['width'])
    gt_y1, gt_y2 = image_data['bboxes'][0]['y1']*(X.shape[1]/image_data['height']), image_data['bboxes'][0]['y2']
*(X.shape[1]/image_data['height'])
    gt_x1, gt_y1, gt_x2, gt_y2 = int(gt_x1), int(gt_y1), int(gt_x2), int(gt_y2)

    img = debug_img.copy()
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    color = (0, 255, 0)
    #   cv2.putText(img, 'gt bbox', (gt_x1, gt_y1-5), cv2.FONT_HERSHEY_DUPLEX, 0.7, color, 1)
    cv2.rectangle(img, (gt_x1, gt_y1), (gt_x2, gt_y2), color, 2)
    cv2.circle(img, (int((gt_x1+gt_x2)/2), int((gt_y1+gt_y2)/2)), 3, color, -1)

    # Add text
    textLabel = 'gt bbox'
    (retval,baseLine) = cv2.getTextSize(textLabel,cv2.FONT_HERSHEY_COMPLEX,0.5,1)
    textOrg = (gt_x1, gt_y1+5)
    cv2.rectangle(img, (textOrg[0] - 5, textOrg[1]+baseLine - 5), (textOrg[0]+retval[0] + 5, textOrg[1]-retval[1]
- 5), (0, 0, 0), 2)
    cv2.rectangle(img, (textOrg[0] - 5,textOrg[1]+baseLine - 5), (textOrg[0]+retval[0] + 5, textOrg[1]-retval[1]
- 5), (255, 255, 255), -1)
    cv2.putText(img, textLabel, textOrg, cv2.FONT_HERSHEY_DUPLEX, 0.5, (0, 0, 0), 1)

    # Draw positive anchors according to the y_rpn_regr
    for i in range(debug_num_pos):

        color = (100+i*(155/4), 0, 100+i*(155/4))

        idx = pos_regr[2][i*4]/4
        anchor_size = C.anchor_box_scales[int(idx/3)]
        anchor_ratio = C.anchor_box_ratios[2-int((idx+1)%3)]

        center = (pos_regr[1][i*4]*C.rpn_stride, pos_regr[0][i*4]*C.rpn_stride)
        print('Center position of positive anchor: ', center)
        cv2.circle(img, center, 3, color, -1)
        anc_w, anc_h = anchor_size*anchor_ratio[0], anchor_size*anchor_ratio[1]
        cv2.rectangle(img, (center[0]-int(anc_w/2), center[1]-int(anc_h/2)), (center[0]+int(anc_w/2), center[1]+i
nt(anc_h/2)), color, 2)
#         cv2.putText(img, 'pos anchor bbox '+str(i+1), (center[0]-int(anc_w/2), center[1]-int(anc_h/2)-5), cv2.F
ONT_HERSHEY_DUPLEX, 0.5, color, 1)

print('Green bboxes is ground-truth bbox. Others are positive anchors')
plt.figure(figsize=(8,8))
plt.grid()
plt.imshow(img)
plt.show()
```

```
Original image: height=1200 width=1600
Resized image:  height=300 width=400 C.im_size=300
Feature map size: height=18 width=25 C.rpn_stride=16
(1, 300, 400, 3)
2 includes 'y_rpn_cls' and 'y_rpn_regr'
Shape of y_rpn_cls (1, 18, 25, 18)
Shape of y_rpn_regr (1, 18, 25, 72)
{'filepath': 'Dataset/Open Images Dataset v4 (Bounding Boxes)/train\\151f3af5-5e6e-4bcd-9389-4974fc6
8813e.png', 'width': 1600, 'height': 1200, 'bboxes': [{'class': 'red blood cell', 'x1': 1082, 'x2':
1177, 'y1': 539, 'y2': 631}, {'class': 'red blood cell', 'x1': 483, 'x2': 605, 'y1': 385, 'y2': 502}
, {'class': 'red blood cell', 'x1': 192, 'x2': 272, 'y1': 248, 'y2': 340}, {'class': 'red blood cell
', 'x1': 942, 'x2': 1048, 'y1': 143, 'y2': 245}, {'class': 'red blood cell', 'x1': 1219, 'x2': 1331,
'y1': 404, 'y2': 490}, {'class': 'red blood cell', 'x1': 648, 'x2': 754, 'y1': 153, 'y2': 251}, {'cl
ass': 'red blood cell', 'x1': 554, 'x2': 655, 'y1': 199, 'y2': 285}, {'class': 'red blood cell', 'x1
': 505, 'x2': 603, 'y1': 90, 'y2': 192}, {'class': 'red blood cell', 'x1': 1121, 'x2': 1226, 'y1': 7
64, 'y2': 864}, {'class': 'red blood cell', 'x1': 1076, 'x2': 1186, 'y1': 244, 'y2': 349}, {'class':
'red blood cell', 'x1': 1222, 'x2': 1326, 'y1': 122, 'y2': 233}, {'class': 'red blood cell', 'x1': 1
173, 'x2': 1271, 'y1': 486, 'y2': 599}, {'class': 'red blood cell', 'x1': 1006, 'x2': 1123, 'y1': 63
8, 'y2': 757}, {'class': 'red blood cell', 'x1': 800, 'x2': 908, 'y1': 39, 'y2': 145}, {'class': 're
d blood cell', 'x1': 8, 'x2': 103, 'y1': 943, 'y2': 1039}, {'class': 'red blood cell', 'x1': 261, 'x
2': 361, 'y1': 924, 'y2': 1020}, {'class': 'red blood cell', 'x1': 440, 'x2': 554, 'y1': 1052, 'y2':
1153}, {'class': 'red blood cell', 'x1': 1187, 'x2': 1297, 'y1': 662, 'y2': 783}, {'class': 'red blo
od cell', 'x1': 1482, 'x2': 1585, 'y1': 599, 'y2': 701}, {'class': 'red blood cell', 'x1': 744, 'x2'
: 834, 'y1': 216, 'y2': 310}, {'class': 'red blood cell', 'x1': 75, 'x2': 188, 'y1': 548, 'y2': 661}
, {'class': 'red blood cell', 'x1': 941, 'x2': 1048, 'y1': 449, 'y2': 545}, {'class': 'red blood cel
l', 'x1': 479, 'x2': 582, 'y1': 811, 'y2': 933}, {'class': 'red blood cell', 'x1': 1191, 'x2': 1283,
'y1': 1021, 'y2': 1115}, {'class': 'red blood cell', 'x1': 1234, 'x2': 1341, 'y1': 211, 'y2': 300},
{'class': 'red blood cell', 'x1': 1179, 'x2': 1303, 'y1': 304, 'y2': 396}, {'class': 'red blood cell
', 'x1': 1299, 'x2': 1408, 'y1': 601, 'y2': 694}, {'class': 'red blood cell', 'x1': 983, 'x2': 1101,
'y1': 71, 'y2': 164}, {'class': 'red blood cell', 'x1': 1232, 'x2': 1342, 'y1': 527, 'y2': 628}, {'c
lass': 'red blood cell', 'x1': 171, 'x2': 289, 'y1': 120, 'y2': 215}, {'class': 'red blood cell', 'x
1': 454, 'x2': 558, 'y1': 948, 'y2': 1052}, {'class': 'red blood cell', 'x1': 289, 'x2': 388, 'y1':
1017, 'y2': 1111}, {'class': 'red blood cell', 'x1': 796, 'x2': 894, 'y1': 156, 'y2': 255}, {'class'
: 'red blood cell', 'x1': 413, 'x2': 510, 'y1': 135, 'y2': 229}, {'class': 'red blood cell', 'x1': 1
435, 'x2': 1540, 'y1': 990, 'y2': 1089}, {'class': 'red blood cell', 'x1': 700, 'x2': 802, 'y1': 379
, 'y2': 496}, {'class': 'red blood cell', 'x1': 988, 'x2': 1089, 'y1': 977, 'y2': 1082}, {'class': '
red blood cell', 'x1': 200, 'x2': 320, 'y1': 633, 'y2': 736}, {'class': 'red blood cell', 'x1': 1361
, 'x2': 1478, 'y1': 1063, 'y2': 1168}, {'class': 'red blood cell', 'x1': 116, 'x2': 203, 'y1': 760,
'y2': 875}, {'class': 'red blood cell', 'x1': 203, 'x2': 308, 'y1': 1093, 'y2': 1189}, {'class': 're
d blood cell', 'x1': 323, 'x2': 424, 'y1': 717, 'y2': 830}, {'class': 'red blood cell', 'x1': 355, '
x2': 453, 'y1': 543, 'y2': 646}, {'class': 'red blood cell', 'x1': 689, 'x2': 792, 'y1': 58, 'y2': 1
56}, {'class': 'red blood cell', 'x1': 1450, 'x2': 1562, 'y1': 740, 'y2': 847}, {'class': 'red blood
cell', 'x1': 165, 'x2': 278, 'y1': 684, 'y2': 787}, {'class': 'red blood cell', 'x1': 448, 'x2': 565
, 'y1': 549, 'y2': 655}, {'class': 'red blood cell', 'x1': 832, 'x2': 931, 'y1': 963, 'y2': 1078}, {
'class': 'red blood cell', 'x1': 449, 'x2': 536, 'y1': 229, 'y2': 317}, {'class': 'red blood cell',
'x1': 1265, 'x2': 1355, 'y1': 699, 'y2': 790}, {'class': 'red blood cell', 'x1': 8, 'x2': 116, 'y1':
772, 'y2': 885}, {'class': 'red blood cell', 'x1': 792, 'x2': 911, 'y1': 813, 'y2': 925}, {'class':
'red blood cell', 'x1': 279, 'x2': 381, 'y1': 124, 'y2': 227}, {'class': 'red blood cell', 'x1': 146
6, 'x2': 1553, 'y1': 884, 'y2': 984}, {'class': 'red blood cell', 'x1': 1147, 'x2': 1249, 'y1': 909,
'y2': 1025}, {'class': 'red blood cell', 'x1': 1305, 'x2': 1399, 'y1': 1008, 'y2': 1125}, {'class':
'red blood cell', 'x1': 1298, 'x2': 1414, 'y1': 984, 'y2': 1061}, {'class': 'red blood cell', 'x1':
817, 'x2': 927, 'y1': 257, 'y2': 355}, {'class': 'red blood cell', 'x1': 952, 'x2': 1029, 'y1': 270,
'y2': 358}, {'class': 'red blood cell', 'x1': 367, 'x2': 454, 'y1': 994, 'y2': 1101}, {'class': 'red
blood cell', 'x1': 1314, 'x2': 1415, 'y1': 464, 'y2': 566}, {'class': 'red blood cell', 'x1': 340, '
x2': 453, 'y1': 613, 'y2': 731}, {'class': 'red blood cell', 'x1': 235, 'x2': 349, 'y1': 466, 'y2':
578}, {'class': 'red blood cell', 'x1': 464, 'x2': 563, 'y1': 261, 'y2': 376}, {'class': 'red blood
cell', 'x1': 862, 'x2': 977, 'y1': 930, 'y2': 1035}, {'class': 'red blood cell', 'x1': 475, 'x2': 59
3, 'y1': 648, 'y2': 761}, {'class': 'red blood cell', 'x1': 1363, 'x2': 1476, 'y1': 497, 'y2': 613},
{'class': 'red blood cell', 'x1': 572, 'x2': 679, 'y1': 6, 'y2': 127}, {'class': 'red blood cell', '
x1': 1273, 'x2': 1396, 'y1': 771, 'y2': 892}, {'class': 'red blood cell', 'x1': 175, 'x2': 263, 'y1'
: 535, 'y2': 622}, {'class': 'red blood cell', 'x1': 936, 'x2': 1036, 'y1': 578, 'y2': 662}, {'class
': 'red blood cell', 'x1': 506, 'x2': 627, 'y1': 490, 'y2': 586}, {'class': 'red blood cell', 'x1':
802, 'x2': 909, 'y1': 323, 'y2': 434}, {'class': 'red blood cell', 'x1': 1170, 'x2': 1273, 'y1': 851
, 'y2': 946}, {'class': 'red blood cell', 'x1': 719, 'x2': 832, 'y1': 431, 'y2': 533}, {'class': 're
d blood cell', 'x1': 249, 'x2': 352, 'y1': 566, 'y2': 674}, {'class': 'red blood cell', 'x1': 1481,
'x2': 1583, 'y1': 780, 'y2': 888}, {'class': 'red blood cell', 'x1': 1305, 'x2': 1419, 'y1': 796, 'y
2': 907}, {'class': 'red blood cell', 'x1': 1391, 'x2': 1489, 'y1': 864, 'y2': 970}, {'class': 'red
blood cell', 'x1': 1, 'x2': 100, 'y1': 997, 'y2': 1099}, {'class': 'red blood cell', 'x1': 319, 'x2'
: 436, 'y1': 830, 'y2': 935}, {'class': 'trophozoite', 'x1': 100, 'x2': 259, 'y1': 937, 'y2': 1059},
{'class': 'trophozoite', 'x1': 770, 'x2': 877, 'y1': 685, 'y2': 858}, {'class': 'trophozoite', 'x1':
1487, 'x2': 1587, 'y1': 1065, 'y2': 1200}, {'class': 'trophozoite', 'x1': 903, 'x2': 1033, 'y1': 355
, 'y2': 473}, {'class': 'trophozoite', 'x1': 280, 'x2': 447, 'y1': 238, 'y2': 338}, {'class': 'red b
lood cell', 'x1': 325, 'x2': 430, 'y1': 195, 'y2': 287}, {'class': 'red blood cell', 'x1': 667, 'x2'
: 759, 'y1': 2, 'y2': 98}, {'class': 'red blood cell', 'x1': 1480, 'x2': 1589, 'y1': 180, 'y2': 265}
, {'class': 'red blood cell', 'x1': 986, 'x2': 1108, 'y1': 216, 'y2': 323}, {'class': 'red blood cel
l', 'x1': 877, 'x2': 962, 'y1': 285, 'y2': 387}, {'class': 'red blood cell', 'x1': 325, 'x2': 447, '
y1': 789, 'y2': 885}, {'class': 'red blood cell', 'x1': 1, 'x2': 85, 'y1': 486, 'y2': 591}, {'class'
: 'red blood cell', 'x1': 9, 'x2': 131, 'y1': 828, 'y2': 939}, {'class': 'red blood cell', 'x1': 265
, 'x2': 381, 'y1': 163, 'y2': 272}, {'class': 'red blood cell', 'x1': 1146, 'x2': 1241, 'y1': 1089,
'y2': 1196}]}
Number of positive anchors for this image: 96
(array([ 0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  1,
         1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,
         3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,
         3,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,
         4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  5,  5,  5,
         5,  5,  5,  5,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,
         7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  8,  8,  8,  8,  8,
         8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,
         9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10,
        10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11, 11, 11,
        11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
        12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13,
        13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14,
        14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15,
        15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 16, 16, 16,
        16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 17, 17,
        17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17]),
       dtype=int64), array([ 2,  4,  6,  7,  8, 11, 11, 11,  2,  6,  9,  9, 11, 11, 13, 13, 16,
        16, 19, 22, 24,  3,  3,  5,  5,  7,  7,  8,  8, 17, 19, 19, 20, 24,
        2,  3,  5,  5,  5,  5,  9,  9, 10, 10, 12, 13, 13, 14, 15, 15, 15,
       20, 20, 21, 22, 23, 23, 23,  1,  3,  3,  5,  5,  6,  7,  7,  8,  8,
       10, 10, 12, 12, 13, 13, 15, 15, 16, 16, 17, 17, 18, 19, 10, 13, 13,
       14, 14, 19, 19,  1,  8,  8, 11, 11, 12, 14, 15, 15, 17, 18, 19, 19,
        4,  5, 11, 12, 12, 13, 13, 13, 15, 15, 18, 23,  0,  0,  3,  3,  4,
        4,  6,  8,  8, 15, 16, 19, 19, 19, 21, 21, 22, 22,  2,  2,  4,  4,
        6,  6,  7,  7, 12, 15, 15, 16, 17, 17, 19, 20, 20, 21,  1,  1,  4,
        4,  6,  6, 10, 16, 16, 21, 21, 23, 23, 24,  3,  3,  5,  7,  8,  8,
       10, 12, 13, 19, 19, 20, 20,  1,  1,  1,  2,  2,  4,  5,  5, 11, 12,
       12, 18, 18, 19, 20, 20, 21, 21, 22, 23, 23, 24,  0,  1,  1,  1,  5,
        5,  6,  6,  7,  8,  8, 13, 13, 14, 18, 21, 21, 23, 23, 23,  2,  3,
       10, 12, 19, 19, 20, 22, 22, 23, 23, 23, 24,  1,  1,  2,  2,  4,  4,
        6,  7,  7, 10, 12, 13, 13, 14, 14, 18, 18, 20, 21, 21,  1,  1,  2,
        3,  5,  5,  6,  6, 12, 12, 16, 16, 19, 19, 21, 21, 23, 23,  3,  3,
        3,  6,  7,  7,  9, 10, 13, 15, 18, 18, 19, 20, 22, 22, 23, 23],
      dtype=int64), array([ 1,  5,  0,  1,  1,  0,  3, 12,  6,  0,  7, 16,  6, 15,  6, 15,  6,
       15,  1,  0,  3,  6, 15,  6, 15,  6, 15,  6, 15,  0,  6, 15,  5,  3,
        5,  5,  6,  8, 15, 17,  6, 15,  6, 15,  2,  6, 15,  6,  1,  7, 16,
        6, 15,  3,  3,  0,  8, 17,  3,  3, 12,  8, 17,  0,  3, 12,  6, 15,
        5,  6,  6, 15,  6, 15,  3, 12,  6, 15,  6, 15,  0,  6,  7,  7, 16,
        6, 15,  8, 17,  2,  6, 15,  6, 15,  4,  4,  6, 15,  0,  5,  6, 15,
        5,  6,  0,  8, 17,  2,  3,  6,  6, 15,  3,  6,  4, 13,  7, 16,  6,
       15,  0,  8, 17,  3,  4,  5,  6, 15,  6, 15,  6, 15,  6, 15,  6, 15,
        6, 15,  8, 17,  8,  6, 15,  5,  6, 15,  4,  6, 15,  4,  0,  5,  8,
       17,  6, 15,  4,  6, 15,  6, 15,  6, 15,  4,  6, 15,  3,  8,  7, 16,
        2,  0,  2,  6, 15,  3, 12,  4,  8, 17,  7, 16,  6,  6, 15,  6,  6,
       15,  6, 15,  3,  7, 16,  1,  2,  1,  6, 15,  1,  1,  2,  8, 17,  6,
       15,  6, 15,  1,  6, 15,  6, 15,  6,  8,  6, 15,  5,  6, 15,  5,  4,
        4,  8,  6, 15,  0,  6, 15,  2,  6, 15,  3,  8, 17,  6, 15,  6, 15,
        0,  6, 15,  6,  5,  6, 15,  6, 15,  6, 15,  3,  6, 15,  8, 17,  2,
        7,  6, 15,  6, 15,  2,  3,  6, 15,  6, 15,  6, 15,  6, 15,  2,  6,
       15,  5,  6, 15,  4,  5,  0,  8,  6, 15,  5,  6,  6, 15,  6, 15],
      dtype=int64))
(array([ 0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,
        2,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,
        3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,
        3,  3,  3,  3,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,
        4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,
        4,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,
        5,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,
        7,  7,  7,  7,  7,  7,  7,  7,  8,  8,  8,  8,  8,  8,  8,  8,  8,
        8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,
        8,  8,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,
        9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10,
       10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 11,
       11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 12, 12,
       12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
       12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13,
       13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
       13, 13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 15, 15,
       15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
       15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 16, 16, 16, 16,
       16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16,
       16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17,
       17, 17, 17, 17, 17, 17, 17, 17, 17, 17], dtype=int64), array([11, 11, 11, 11,  9,  9,  9,  9,
11, 11, 11, 11, 13, 13, 13, 13, 16,
       16, 16, 16,  3,  3,  3,  3,  5,  5,  5,  5,  7,  7,  7,  7,  8,  8,
        8,  8, 19, 19, 19, 19,  5,  5,  5,  5,  5,  5,  5,  5,  9,  9,  9,
        9, 10, 10, 10, 10, 13, 13, 13, 13, 15, 15, 15, 15, 20, 20, 20, 20,
       23, 23, 23, 23,  3,  3,  3,  3,  5,  5,  5,  5,  7,  7,  7,  7,  8,
        8,  8,  8, 12, 12, 12, 12, 13, 13, 13, 13, 15, 15, 15, 15, 16, 16,
       16, 16, 17, 17, 17, 17, 13, 13, 13, 13, 14, 14, 14, 14, 19, 19, 19,
       19,  8,  8,  8,  8, 11, 11, 11, 11, 15, 15, 15, 15, 19, 19, 19, 19,
       12, 12, 12, 12, 15, 15, 15, 15,  0,  0,  0,  0,  3,  3,  3,  3,  4,
        4,  4,  4,  8,  8,  8,  8, 19, 19, 19, 19, 21, 21, 21, 21, 22, 22,
       22, 22,  2,  2,  2,  2,  4,  4,  4,  4,  6,  6,  6,  6,  7,  7,  7,
        7, 15, 15, 15, 15, 17, 17, 17, 17, 20, 20, 20, 20,  4,  4,  4,  4,
        6,  6,  6,  6, 16, 16, 16, 16, 21, 21, 21, 21, 23, 23, 23, 23,  3,
        3,  3,  3,  8,  8,  8,  8, 19, 19, 19, 19, 20, 20, 20, 20,  1,  1,
        1,  1,  2,  2,  2,  2,  5,  5,  5,  5, 12, 12, 12, 12, 18, 18, 18,
       18, 20, 20, 20, 20, 23, 23, 23, 23,  1,  1,  1,  1,  5,  5,  5,  5,
        6,  6,  6,  6,  8,  8,  8,  8, 13, 13, 13, 13, 21, 21, 21, 21, 23,
       23, 23, 23, 19, 19, 19, 19, 22, 22, 22, 22, 23, 23, 23, 23,  1,  1,
        1,  1,  2,  2,  2,  2,  4,  4,  4,  4,  7,  7,  7,  7, 13, 13, 13,
       13, 14, 14, 14, 14, 18, 18, 18, 18, 21, 21, 21, 21,  1,  1,  1,  1,
        5,  5,  5,  5,  6,  6,  6,  6, 16, 16, 16, 16, 19, 19, 19, 19, 21,
       21, 21, 21, 23, 23, 23, 23,  3,  3,  3,  3,  7,  7,  7,  7, 18, 18,
       18, 18, 22, 22, 22, 22, 23, 23, 23, 23], dtype=int64), array([12, 13, 14, 15, 28, 29, 30, 31,
24, 25, 26, 27, 24, 25, 26, 27, 24,
       25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25,
       26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 32, 33, 34, 35, 24, 25, 26,
       27, 24, 25, 26, 27, 24, 25, 26, 27, 28, 29, 30, 31, 24, 25, 26, 27,
       32, 33, 34, 35, 12, 13, 14, 15, 32, 33, 34, 35, 12, 13, 14, 15, 24,
       25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 12, 13, 14, 15, 24, 25,
       26, 27, 24, 25, 26, 27, 28, 29, 30, 31, 24, 25, 26, 27, 32, 33, 34,
       35, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27,
       32, 33, 34, 35, 24, 25, 26, 27, 16, 17, 18, 19, 28, 29, 30, 31, 24,
       25, 26, 27, 32, 33, 34, 35, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25,
       26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 32, 33, 34,
       35, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 32, 33, 34, 35,
       24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24,
       25, 26, 27, 28, 29, 30, 31, 24, 25, 26, 27, 12, 13, 14, 15, 32, 33,
       34, 35, 28, 29, 30, 31, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26,
       27, 28, 29, 30, 31, 24, 25, 26, 27, 32, 33, 34, 35, 24, 25, 26, 27,
       24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24,
       25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 32, 33,
       34, 35, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26,
       27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 32, 33, 34, 35,
       24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24,
       25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25, 26, 27, 24, 25,
       26, 27, 24, 25, 26, 27, 24, 25, 26, 27], dtype=int64))
y_rpn_cls for possible pos anchor: [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
y_rpn_regr for positive anchor: [ 0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
  1.          1.          1.          1.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
 -1.4375      1.125       1.45162201  1.62186038  0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.        ]
Center position of positive anchor:  (176, 0)
Center position of positive anchor:  (144, 16)
Center position of positive anchor:  (176, 16)
Center position of positive anchor:  (208, 16)
Center position of positive anchor:  (256, 16)
Center position of positive anchor:  (48, 32)
Center position of positive anchor:  (80, 32)
Center position of positive anchor:  (112, 32)
Center position of positive anchor:  (128, 32)
Center position of positive anchor:  (304, 32)
Center position of positive anchor:  (80, 48)
Center position of positive anchor:  (80, 48)
Center position of positive anchor:  (144, 48)
Center position of positive anchor:  (160, 48)
Center position of positive anchor:  (208, 48)
Center position of positive anchor:  (240, 48)
Center position of positive anchor:  (320, 48)
Center position of positive anchor:  (368, 48)
Center position of positive anchor:  (48, 64)
Center position of positive anchor:  (80, 64)
Center position of positive anchor:  (112, 64)
Center position of positive anchor:  (128, 64)
Center position of positive anchor:  (192, 64)
Center position of positive anchor:  (208, 64)
Center position of positive anchor:  (240, 64)
Center position of positive anchor:  (256, 64)
Center position of positive anchor:  (272, 64)
Center position of positive anchor:  (208, 80)
Center position of positive anchor:  (224, 80)
Center position of positive anchor:  (304, 80)
Center position of positive anchor:  (128, 96)
Center position of positive anchor:  (176, 96)
Center position of positive anchor:  (240, 96)
Center position of positive anchor:  (304, 96)
Center position of positive anchor:  (192, 112)
Center position of positive anchor:  (240, 112)
Center position of positive anchor:  (0, 128)
Center position of positive anchor:  (48, 128)
Center position of positive anchor:  (64, 128)
Center position of positive anchor:  (128, 128)
Center position of positive anchor:  (304, 128)
Center position of positive anchor:  (336, 128)
Center position of positive anchor:  (352, 128)
Center position of positive anchor:  (32, 144)
Center position of positive anchor:  (64, 144)
Center position of positive anchor:  (96, 144)
Center position of positive anchor:  (112, 144)
Center position of positive anchor:  (240, 144)
Center position of positive anchor:  (272, 144)
Center position of positive anchor:  (320, 144)
Center position of positive anchor:  (64, 160)
Center position of positive anchor:  (96, 160)
Center position of positive anchor:  (256, 160)
Center position of positive anchor:  (336, 160)
Center position of positive anchor:  (368, 160)
Center position of positive anchor:  (48, 176)
Center position of positive anchor:  (128, 176)
Center position of positive anchor:  (304, 176)
Center position of positive anchor:  (320, 176)
Center position of positive anchor:  (16, 192)
Center position of positive anchor:  (32, 192)
Center position of positive anchor:  (80, 192)
Center position of positive anchor:  (192, 192)
Center position of positive anchor:  (288, 192)
Center position of positive anchor:  (320, 192)
Center position of positive anchor:  (368, 192)
Center position of positive anchor:  (16, 208)
Center position of positive anchor:  (80, 208)
Center position of positive anchor:  (96, 208)
Center position of positive anchor:  (128, 208)
Center position of positive anchor:  (208, 208)
Center position of positive anchor:  (336, 208)
Center position of positive anchor:  (368, 208)
Center position of positive anchor:  (304, 224)
Center position of positive anchor:  (352, 224)
Center position of positive anchor:  (368, 224)
Center position of positive anchor:  (16, 240)
Center position of positive anchor:  (32, 240)
Center position of positive anchor:  (64, 240)
Center position of positive anchor:  (112, 240)
Center position of positive anchor:  (208, 240)
Center position of positive anchor:  (224, 240)
Center position of positive anchor:  (288, 240)
Center position of positive anchor:  (336, 240)
Center position of positive anchor:  (16, 256)
Center position of positive anchor:  (80, 256)
Center position of positive anchor:  (96, 256)
Center position of positive anchor:  (256, 256)
Center position of positive anchor:  (304, 256)
Center position of positive anchor:  (336, 256)
Center position of positive anchor:  (368, 256)
Center position of positive anchor:  (48, 272)
Center position of positive anchor:  (112, 272)
Center position of positive anchor:  (288, 272)
Center position of positive anchor:  (352, 272)
Center position of positive anchor:  (368, 272)
Green bboxes is ground-truth bbox. Others are positive anchors
```



Build the model

Building the model using VGG-16 as the base layer and getting the feature map to input it to RPN

In [24]:

```
input_shape_img = (None, None, 3)

img_input = Input(shape=input_shape_img)
roi_input = Input(shape=(None, 4))

# define the base network (VGG here, can be Resnet50, Inception, etc)
shared_layers = nn_base(img_input, trainable=True)
```

```
WARNING:tensorflow:From c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\tens
orflow\python\framework\op_def_library.py:263: colocate_with (from tensorflow.python.framework.ops)
is deprecated and will be removed in a future version.
Instructions for updating:
Colocations handled automatically by placer.
```

In [25]:

```
# define the RPN, built on the base layers
num_anchors = len(C.anchor_box_scales) * len(C.anchor_box_ratios) # 9
rpn = rpn_layer(shared_layers, num_anchors)

classifier = classifier_layer(shared_layers, roi_input, C.num_rois, nb_classes=len(classes_count))

model_rpn = Model(img_input, rpn[:2])
model_classifier = Model([img_input, roi_input], classifier)

# this is a model that holds both the RPN and the classifier, used to load/save weights for the models
model_all = Model([img_input, roi_input], rpn[:2] + classifier)

# Because the google colab can only run the session several hours one time (then you need to connect again),
# we need to save the model and load the model to continue training
if not os.path.isfile(C.model_path):
    #If this is the begin of the training, load the pre-traind base network such as vgg-16
    try:
        print('This is the first time of your training')
        print('loading weights from {}'.format(C.base_net_weights))
        model_rpn.load_weights(C.base_net_weights, by_name=True)
        model_classifier.load_weights(C.base_net_weights, by_name=True)
    except:
        print('Could not load pretrained model weights. Weights can be found in the keras application folder \
            https://github.com/fchollet/keras/tree/master/keras/applications')

    # Create the record.csv file to record losses, acc and mAP
    record_df = pd.DataFrame(columns=['mean_overlapping_bboxes', 'class_acc', 'loss_rpn_cls', 'loss_rpn_regr', 'l
oss_class_cls', 'loss_class_regr', 'curr_loss', 'elapsed_time', 'mAP'])
else:
    # If this is a continued training, load the trained model from before
    print('Continue training based on previous trained model')
    print('Loading weights from {}'.format(C.model_path))
    model_rpn.load_weights(C.model_path, by_name=True)
    model_classifier.load_weights(C.model_path, by_name=True)

    # Load the records
    record_df = pd.read_csv(record_path)

    r_mean_overlapping_bboxes = record_df['mean_overlapping_bboxes']
    r_class_acc = record_df['class_acc']
    r_loss_rpn_cls = record_df['loss_rpn_cls']
    r_loss_rpn_regr = record_df['loss_rpn_regr']
    r_loss_class_cls = record_df['loss_class_cls']
    r_loss_class_regr = record_df['loss_class_regr']
    r_curr_loss = record_df['curr_loss']
    r_elapsed_time = record_df['elapsed_time']
    r_mAP = record_df['mAP']

    print('Already train %dK batches'% (len(record_df)))
```

```
WARNING:tensorflow:From c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\kera
s\backend\tensorflow_backend.py:3445: calling dropout (from tensorflow.python.ops.nn_ops) with keep_
prob is deprecated and will be removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 - keep_prob`.
Continue training based on previous trained model
Loading weights from Dataset/Open Images Dataset v4 (Bounding Boxes)/model/model_frcnn_vgg.hdf5
Already train 20K batches
```

In [26]:

```
optimizer = Adam(lr=1e-5)
optimizer_classifier = Adam(lr=1e-5)
model_rpn.compile(optimizer=optimizer, loss=[rpn_loss_cls(num_anchors), rpn_loss_regr(num_anchors)])
model_classifier.compile(optimizer=optimizer_classifier, loss=[class_loss_cls, class_loss_regr(len(classes_count)
-1)], metrics={'dense_class_{}'.format(len(classes_count)): 'accuracy'})
model_all.compile(optimizer='sgd', loss='mae')
```

In [27]:

```
# Training setting
total_epochs = len(record_df)
r_epochs = len(record_df)

epoch_length = 1000
num_epochs = 40
iter_num = 0

total_epochs += num_epochs

losses = np.zeros((epoch_length, 5))
rpn_accuracy_rpn_monitor = []
rpn_accuracy_for_epoch = []

if len(record_df)==0:
    best_loss = np.Inf
else:
    best_loss = np.min(r_curr_loss)
```

In [28]:

```
print(len(record_df))
```

```
20
```

Training a model

2 RPN loss - classifier and regressor 2 R-CNN loss - classifier and regressor

In [29]:

```
start_time = time.time()
for epoch_num in range(num_epochs):

    progbar = generic_utils.Progbar(epoch_length)
    print('Epoch {}/{}'.format(r_epochs + 1, total_epochs))

    r_epochs += 1

    while True:
        try:

            if len(rpn_accuracy_rpn_monitor) == epoch_length and C.verbose:
                mean_overlapping_bboxes = float(sum(rpn_accuracy_rpn_monitor))/len(rpn_accuracy_rpn_monitor)
                rpn_accuracy_rpn_monitor = []
#                 print('Average number of overlapping bounding boxes from RPN = {} for {} previous iterations'.f
ormat(mean_overlapping_bboxes, epoch_length))
                if mean_overlapping_bboxes == 0:
                    print('RPN is not producing bounding boxes that overlap the ground truth boxes. Check RPN set
tings or keep training.')

            # Generate X (x_img) and label Y ([y_rpn_cls, y_rpn_regr])
            X, Y, img_data, debug_img, debug_num_pos = next(data_gen_train)

            # Train rpn model and get loss value [_, loss_rpn_cls, loss_rpn_regr]
            loss_rpn = model_rpn.train_on_batch(X, Y)

            # Get predicted rpn from rpn model [rpn_cls, rpn_regr]
            P_rpn = model_rpn.predict_on_batch(X)

            # R: bboxes (shape=(300,4))
            # Convert rpn layer to roi bboxes
            R = rpn_to_roi(P_rpn[0], P_rpn[1], C, K.image_data_format(), use_regr=True, overlap_thresh=0.7, max_b
oxes=300)

            # note: calc_iou converts from (x1,y1,x2,y2) to (x,y,w,h) format
            # X2: bboxes that iou > C.classifier_min_overlap for all gt bboxes in 300 non_max_suppression bboxes
            # Y1: one hot code for bboxes from above => x_roi (X)
            # Y2: corresponding labels and corresponding gt bboxes
            X2, Y1, Y2, IouS = calc_iou(R, img_data, C, class_mapping)
            # If X2 is None means there are no matching bboxes
            if X2 is None:
                rpn_accuracy_rpn_monitor.append(0)
                rpn_accuracy_for_epoch.append(0)
                continue

            # Find out the positive anchors and negative anchors
            neg_samples = np.where(Y1[0, :, -1] == 1)
            pos_samples = np.where(Y1[0, :, -1] == 0)

            if len(neg_samples) > 0:
                neg_samples = neg_samples[0]
            else:
                neg_samples = []

            if len(pos_samples) > 0:
                pos_samples = pos_samples[0]
            else:
                pos_samples = []

            rpn_accuracy_rpn_monitor.append(len(pos_samples))
            rpn_accuracy_for_epoch.append((len(pos_samples)))

            if C.num_rois > 1:
                # If number of positive anchors is larger than 4//2 = 2, randomly choose 2 pos samples
                if len(pos_samples) < C.num_rois//2:
                    selected_pos_samples = pos_samples.tolist()
                else:
                    selected_pos_samples = np.random.choice(pos_samples, C.num_rois//2, replace=False).tolist()

                # Randomly choose (num_rois - num_pos) neg samples
                try:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois - len(selected_pos_samples),
replace=False).tolist()
                except:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois - len(selected_pos_samples),
replace=True).tolist()

                # Save all the pos and neg samples in sel_samples
                sel_samples = selected_pos_samples + selected_neg_samples
            else:
                # in the extreme case where num_rois = 1, we pick a random pos or neg sample
                selected_pos_samples = pos_samples.tolist()
                selected_neg_samples = neg_samples.tolist()
                if np.random.randint(0, 2):
                    sel_samples = random.choice(neg_samples)
                else:
                    sel_samples = random.choice(pos_samples)

            # training_data: [X, X2[:, sel_samples, :]]
            # labels: [Y1[:, sel_samples, :], Y2[:, sel_samples, :]]
            #  X                     => img_data resized image
            #  X2[:, sel_samples, :] => num_rois (4 in here) bboxes which contains selected neg and pos
            #  Y1[:, sel_samples, :] => one hot encode for num_rois bboxes which contains selected neg and pos
            #  Y2[:, sel_samples, :] => labels and gt bboxes for num_rois bboxes which contains selected neg and
pos
            loss_class = model_classifier.train_on_batch([X, X2[:, sel_samples, :]], [Y1[:, sel_samples, :], Y2[:
, sel_samples, :]])

            losses[iter_num, 0] = loss_rpn[1]
            losses[iter_num, 1] = loss_rpn[2]

            losses[iter_num, 2] = loss_class[1]
            losses[iter_num, 3] = loss_class[2]
            losses[iter_num, 4] = loss_class[3]

            iter_num += 1

            progbar.update(iter_num, [('rpn_cls', np.mean(losses[:iter_num, 0])), ('rpn_regr', np.mean(losses[:it
er_num, 1])),
                                      ('final_cls', np.mean(losses[:iter_num, 2])), ('final_regr', np.mean(losses
[:iter_num, 3]))])

            if iter_num == epoch_length:
                loss_rpn_cls = np.mean(losses[:, 0])
                loss_rpn_regr = np.mean(losses[:, 1])
                loss_class_cls = np.mean(losses[:, 2])
                loss_class_regr = np.mean(losses[:, 3])
                class_acc = np.mean(losses[:, 4])

                mean_overlapping_bboxes = float(sum(rpn_accuracy_for_epoch)) / len(rpn_accuracy_for_epoch)
                rpn_accuracy_for_epoch = []
                if C.verbose:

                    print('Mean number of bounding boxes from RPN overlapping ground truth boxes: {}'.format(mean
_overlapping_bboxes))
                    print('Classifier accuracy for bounding boxes from RPN: {}'.format(class_acc))
                    print('Loss RPN classifier: {}'.format(loss_rpn_cls))
                    print('Loss RPN regression: {}'.format(loss_rpn_regr))
                    print('Loss Detector classifier: {}'.format(loss_class_cls))
                    print('Loss Detector regression: {}'.format(loss_class_regr))
                    print('Total loss: {}'.format(loss_rpn_cls + loss_rpn_regr + loss_class_cls + loss_class_regr
))
                    print('Elapsed time: {}'.format(time.time() - start_time))
                    elapsed_time = (time.time()-start_time)/60

                curr_loss = loss_rpn_cls + loss_rpn_regr + loss_class_cls + loss_class_regr
                iter_num = 0
                start_time = time.time()

                if curr_loss < best_loss:
                    if C.verbose:
                        print('Total loss decreased from {} to {}, saving weights'.format(best_loss,curr_loss))
                    best_loss = curr_loss
                    model_all.save_weights(C.model_path)

                new_row = {'mean_overlapping_bboxes':round(mean_overlapping_bboxes, 3),
                           'class_acc':round(class_acc, 3),
                           'loss_rpn_cls':round(loss_rpn_cls, 3),
                           'loss_rpn_regr':round(loss_rpn_regr, 3),
                           'loss_class_cls':round(loss_class_cls, 3),
                           'loss_class_regr':round(loss_class_regr, 3),
                           'curr_loss':round(curr_loss, 3),
                           'elapsed_time':round(elapsed_time, 3),
                           'mAP': 0}

                record_df = record_df.append(new_row, ignore_index=True)
                record_df.to_csv(record_path, index=0)

                break

        except Exception as e:
            print('Exception: {}'.format(e))
            continue

print('Training complete, exiting.')
```

```
Epoch 1/40
WARNING:tensorflow:From C:\Users\srila\AppData\Local\Programs\Python\Python36\Lib\site-packages\tens
orflow\python\ops\math_ops.py:3066: to_int32 (from tensorflow.python.ops.math_ops) is deprecated and
will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
1000/1000 [==============================] - 6723s 7s/step - rpn_cls: 2.3347 - rpn_regr: 0.1594 - fi
nal_cls: 0.9081 - final_regr: 0.0536
Mean number of bounding boxes from RPN overlapping ground truth boxes: 28.844
Classifier accuracy for bounding boxes from RPN: 0.65125
Loss RPN classifier: 1.2809414826540533
Loss RPN regression: 0.13392785605043173
Loss Detector classifier: 0.7541338799484074
Loss Detector regression: 0.058351556689473
Total loss: 2.227354775342365
Elapsed time: 6722.938431739807
Total loss decreased from inf to 2.227354775342365, saving weights
Epoch 2/40
1000/1000 [==============================] - 6912s 7s/step - rpn_cls: 0.5581 - rpn_regr: 0.1012 - fi
nal_cls: 0.5400 - final_regr: 0.0594
Mean number of bounding boxes from RPN overlapping ground truth boxes: 31.025
Classifier accuracy for bounding boxes from RPN: 0.78
Loss RPN classifier: 0.5186480625785657
Loss RPN regression: 0.09682097958028317
Loss Detector classifier: 0.5322709212508052
Loss Detector regression: 0.057457299676927504
Total loss: 1.2051972630865815
Elapsed time: 6919.993641138077
Total loss decreased from 2.227354775342365 to 1.2051972630865815, saving weights
Epoch 3/40
1000/1000 [==============================] - 6733s 7s/step - rpn_cls: 0.4175 - rpn_regr: 0.0876 - fi
nal_cls: 0.4950 - final_regr: 0.0520
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.697
Classifier accuracy for bounding boxes from RPN: 0.8045
Loss RPN classifier: 0.40441063979233355
Loss RPN regression: 0.08608768356405198
Loss Detector classifier: 0.486815283554839
Loss Detector regression: 0.050786536802363114
Total loss: 1.0281001437135875
```

```
Elapsed time: 6740.037739992142
Total loss decreased from 1.2051972630865815 to 1.0281001437135875, saving weights
Epoch 4/40
1000/1000 [==============================] - 6848s 7s/step - rpn_cls: 0.3653 - rpn_regr: 0.0829 - fi
nal_cls: 0.4594 - final_regr: 0.0632
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.348
Classifier accuracy for bounding boxes from RPN: 0.82525
Loss RPN classifier: 0.3576161533523732
Loss RPN regression: 0.07896673729643226
Loss Detector classifier: 0.4301767678941833
Loss Detector regression: 0.0590315410499461
Total loss: 0.9257911995929349
Elapsed time: 6856.268742322922
Total loss decreased from 1.0281001437135875 to 0.9257911995929349, saving weights
Epoch 5/40
1000/1000 [==============================] - 6792s 7s/step - rpn_cls: 0.3294 - rpn_regr: 0.0736 - fi
nal_cls: 0.3984 - final_regr: 0.0601
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.266
Classifier accuracy for bounding boxes from RPN: 0.834
Loss RPN classifier: 0.3382360807969335
Loss RPN regression: 0.0720826659463346
Loss Detector classifier: 0.39526259384304285
Loss Detector regression: 0.0566410052155843
Total loss: 0.8622223458018954
Elapsed time: 6799.486665010452
Total loss decreased from 0.9257911995929349 to 0.8622223458018954, saving weights
Epoch 6/40
1000/1000 [==============================] - 6778s 7s/step - rpn_cls: 0.2983 - rpn_regr: 0.0682 - fi
nal_cls: 0.3794 - final_regr: 0.0546
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.194
Classifier accuracy for bounding boxes from RPN: 0.856
Loss RPN classifier: 0.30552511191919224
Loss RPN regression: 0.06851604136452079
Loss Detector classifier: 0.37373029634344856
Loss Detector regression: 0.055159976175636984
Total loss: 0.8029314258027985
Elapsed time: 6784.520849943161
Total loss decreased from 0.8622223458018954 to 0.8029314258027985, saving weights
Epoch 7/40
1000/1000 [==============================] - 6819s 7s/step - rpn_cls: 0.3012 - rpn_regr: 0.0639 - fi
nal_cls: 0.3684 - final_regr: 0.0521
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.349
Classifier accuracy for bounding boxes from RPN: 0.857
Loss RPN classifier: 0.2956202124182932
Loss RPN regression: 0.06282897458598018
Loss Detector classifier: 0.35236990158469417
Loss Detector regression: 0.04997506797182723
Total loss: 0.7607941565607949
Elapsed time: 6827.975492954254
Total loss decreased from 0.8029314258027985 to 0.7607941565607949, saving weights
Epoch 8/40
1000/1000 [==============================] - 6819s 7s/step - rpn_cls: 0.3109 - rpn_regr: 0.0630 - fi
nal_cls: 0.3574 - final_regr: 0.0622
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.307
Classifier accuracy for bounding boxes from RPN: 0.863
Loss RPN classifier: 0.3010300902604573
Loss RPN regression: 0.06091257355548441
Loss Detector classifier: 0.34619856309657915
Loss Detector regression: 0.06271925096402993
Total loss: 0.7708604778765509
Elapsed time: 6827.912655591965
Epoch 9/40
1000/1000 [==============================] - 6799s 7s/step - rpn_cls: 0.2763 - rpn_regr: 0.0575 - fi
nal_cls: 0.3498 - final_regr: 0.0595
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.18
Classifier accuracy for bounding boxes from RPN: 0.876
Loss RPN classifier: 0.27863984458670654
Loss RPN regression: 0.05689155087061226
Loss Detector classifier: 0.33211702109285396
Loss Detector regression: 0.05888072609592927
Total loss: 0.7265291426461021
Elapsed time: 6799.409912347794
Total loss decreased from 0.7607941565607949 to 0.7265291426461021, saving weights
Epoch 10/40
1000/1000 [==============================] - 6755s 7s/step - rpn_cls: 0.2601 - rpn_regr: 0.0554 - fi
nal_cls: 0.3129 - final_regr: 0.0589
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.487
Classifier accuracy for bounding boxes from RPN: 0.87175
Loss RPN classifier: 0.273140078156474
Loss RPN regression: 0.05508372028451413
Loss Detector classifier: 0.32331595669177476
Loss Detector regression: 0.05929984462395078
Total loss: 0.7108395997567136
```

```
Elapsed time: 6762.121208190918
Total loss decreased from 0.7265291426461021 to 0.7108395997567136, saving weights
Epoch 11/40
1000/1000 [==============================] - 6760s 7s/step - rpn_cls: 0.2640 - rpn_regr: 0.0535 - fi
nal_cls: 0.2984 - final_regr: 0.0617
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.127
Classifier accuracy for bounding boxes from RPN: 0.882
Loss RPN classifier: 0.2544038438291425
Loss RPN regression: 0.05280188094638288
Loss Detector classifier: 0.29937939195611396
Loss Detector regression: 0.05547748419397976
Total loss: 0.6620626009256192
Elapsed time: 6768.176007270813
Total loss decreased from 0.7108395997567136 to 0.6620626009256192, saving weights
Epoch 12/40
1000/1000 [==============================] - 6775s 7s/step - rpn_cls: 0.1764 - rpn_regr: 0.0527 - fi
nal_cls: 0.3498 - final_regr: 0.0663
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.266
Classifier accuracy for bounding boxes from RPN: 0.869
Loss RPN classifier: 0.167566536823391
Loss RPN regression: 0.05138052842393517
Loss Detector classifier: 0.3291536918994388
Loss Detector regression: 0.06250066769952536
Total loss: 0.6106014248462903
Elapsed time: 6784.039059638977
Total loss decreased from 0.6620626009256192 to 0.6106014248462903, saving weights
Epoch 13/40
1000/1000 [==============================] - 6691s 7s/step - rpn_cls: 0.1556 - rpn_regr: 0.0501 - fi
nal_cls: 0.3123 - final_regr: 0.0506
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.617
Classifier accuracy for bounding boxes from RPN: 0.8855
Loss RPN classifier: 0.15106525567805637
Loss RPN regression: 0.049594349475577476
Loss Detector classifier: 0.3088491088832088
Loss Detector regression: 0.05053162077692105
Total loss: 0.5600403348137637
Elapsed time: 6698.214406013489
Total loss decreased from 0.6106014248462903 to 0.5600403348137637, saving weights
Epoch 14/40
1000/1000 [==============================] - 6794s 7s/step - rpn_cls: 0.1490 - rpn_regr: 0.0488 - fi
nal_cls: 0.2757 - final_regr: 0.0497
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.457
Classifier accuracy for bounding boxes from RPN: 0.889
Loss RPN classifier: 0.14907303395690943
Loss RPN regression: 0.04860098019428551
Loss Detector classifier: 0.2861004366655543
Loss Detector regression: 0.051454101542389254
Total loss: 0.5352285523591385
Elapsed time: 6803.002883434296
Total loss decreased from 0.5600403348137637 to 0.5352285523591385, saving weights
Epoch 15/40
1000/1000 [==============================] - 6835s 7s/step - rpn_cls: 0.1356 - rpn_regr: 0.0465 - fi
nal_cls: 0.2882 - final_regr: 0.0519
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.788
Classifier accuracy for bounding boxes from RPN: 0.89175
Loss RPN classifier: 0.1344173876410495
Loss RPN regression: 0.04594066179636866
Loss Detector classifier: 0.2882424144114193
Loss Detector regression: 0.058885432395152745
Total loss: 0.5274858962439902
Elapsed time: 6843.156196832657
Total loss decreased from 0.5352285523591385 to 0.5274858962439902, saving weights
Epoch 16/40
1000/1000 [==============================] - 6880s 7s/step - rpn_cls: 0.1351 - rpn_regr: 0.0469 - fi
nal_cls: 0.2834 - final_regr: 0.0544
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.947
Classifier accuracy for bounding boxes from RPN: 0.89175
Loss RPN classifier: 0.1353646372853834
Loss RPN regression: 0.04651028857287019
Loss Detector classifier: 0.2909264440734405
Loss Detector regression: 0.05624676379573066
Total loss: 0.5290481337274248
Elapsed time: 6889.203699827194
Epoch 17/40
1000/1000 [==============================] - 6893s 7s/step - rpn_cls: 0.1163 - rpn_regr: 0.0438 - fi
nal_cls: 0.2789 - final_regr: 0.0526
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.164
Classifier accuracy for bounding boxes from RPN: 0.89575
Loss RPN classifier: 0.12304012068540172
Loss RPN regression: 0.04438275943975896
Loss Detector classifier: 0.28191207587004463
Loss Detector regression: 0.05424246825012961
Total loss: 0.5035774242453349
```

```
Elapsed time: 6892.637051582336
Total loss decreased from 0.5274858962439902 to 0.5035774242453349, saving weights
Epoch 18/40
1000/1000 [==============================] - 6804s 7s/step - rpn_cls: 0.1225 - rpn_regr: 0.0439 - fi
nal_cls: 0.2568 - final_regr: 0.0510
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.267
Classifier accuracy for bounding boxes from RPN: 0.894
Loss RPN classifier: 0.1245105185448358
Loss RPN regression: 0.04447207772079855
Loss Detector classifier: 0.2744269459225325
Loss Detector regression: 0.054044814167544246
Total loss: 0.4974543563557111
Elapsed time: 6812.214331626892
Total loss decreased from 0.5035774242453349 to 0.4974543563557111, saving weights
Epoch 19/40
1000/1000 [==============================] - 6794s 7s/step - rpn_cls: 0.1155 - rpn_regr: 0.0429 - fi
nal_cls: 0.2814 - final_regr: 0.0606
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.198
Classifier accuracy for bounding boxes from RPN: 0.8935
Loss RPN classifier: 0.1253600315896196
Loss RPN regression: 0.04354816838214174
Loss Detector classifier: 0.2809509732370207
Loss Detector regression: 0.0532449726027844
Total loss: 0.5031041458115664
Elapsed time: 6800.694343328476
Epoch 20/40
1000/1000 [==============================] - 6929s 7s/step - rpn_cls: 0.1271 - rpn_regr: 0.0432 - fi
nal_cls: 0.2774 - final_regr: 0.0603
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.594
Classifier accuracy for bounding boxes from RPN: 0.8975
Loss RPN classifier: 0.12787452696147908
Loss RPN regression: 0.04281209096033126
Loss Detector classifier: 0.2626244281702384
Loss Detector regression: 0.047496749625366645
Total loss: 0.4808077957174154
Elapsed time: 6928.8898758888245
Total loss decreased from 0.4974543563557111 to 0.4808077957174154, saving weights
Epoch 21/40
   2/1000 [..............................] - ETA: 1:49:53 - rpn_cls: 0.0609 - rpn_regr: 0.0564 - fin
al_cls: 0.6726 - final_regr: 0.2739

---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-29-493fa9600e77> in <module>()
     21 
     22             # Train rpn model and get loss value [_, loss_rpn_cls, loss_rpn_regr]
---> 23             loss_rpn = model_rpn.train_on_batch(X, Y)
     24 
     25             # Get predicted rpn from rpn model [rpn_cls, rpn_regr]

~\AppData\Local\Programs\Python\Python36\Lib\site-packages\keras\engine\training.py in train_on_batc
h(self, x, y, sample_weight, class_weight, reset_metrics)
   1512             ins = x + y + sample_weights
   1513         self._make_train_function()
-> 1514         outputs = self.train_function(ins)
   1515 
   1516         if reset_metrics:

~\AppData\Local\Programs\Python\Python36\Lib\site-packages\tensorflow\python\keras\backend.py in __c
all__(self, inputs)
   3074 
   3075     fetched = self._callable_fn(*array_vals,
-> 3076                                 run_metadata=self.run_metadata)
   3077     self._call_fetch_callbacks(fetched[-len(self._fetches):])
   3078     return nest.pack_sequence_as(self._outputs_structure,

~\AppData\Local\Programs\Python\Python36\Lib\site-packages\tensorflow\python\client\session.py in __
call__(self, *args, **kwargs)
   1437           ret = tf_session.TF_SessionRunCallable(
   1438               self._session._session, self._handle, args, status,
-> 1439               run_metadata_ptr)
   1440         if run_metadata:
   1441           proto_data = tf_session.TF_GetBuffer(run_metadata_ptr)

KeyboardInterrupt: 
```

In [29]:

```
start_time = time.time()
for epoch_num in range(num_epochs):

    progbar = generic_utils.Progbar(epoch_length)
    print('Epoch {}/{}'.format(r_epochs + 1, total_epochs))
    r_epochs += 1


    while True:
        try:

            if len(rpn_accuracy_rpn_monitor) == epoch_length and C.verbose:
                mean_overlapping_bboxes = float(sum(rpn_accuracy_rpn_monitor))/len(rpn_accuracy_rpn_monitor)
                rpn_accuracy_rpn_monitor = []
#                 print('Average number of overlapping bounding boxes from RPN = {} for {} previous iterations'.f
ormat(mean_overlapping_bboxes, epoch_length))
                if mean_overlapping_bboxes == 0:
                    print('RPN is not producing bounding boxes that overlap the ground truth boxes. Check RPN set
tings or keep training.')

            # Generate X (x_img) and label Y ([y_rpn_cls, y_rpn_regr])
            X, Y, img_data, debug_img, debug_num_pos = next(data_gen_train)

            # Train rpn model and get loss value [_, loss_rpn_cls, loss_rpn_regr]
            loss_rpn = model_rpn.train_on_batch(X, Y)

            # Get predicted rpn from rpn model [rpn_cls, rpn_regr]
            P_rpn = model_rpn.predict_on_batch(X)

            # R: bboxes (shape=(300,4))
            # Convert rpn layer to roi bboxes
            R = rpn_to_roi(P_rpn[0], P_rpn[1], C, K.image_data_format(), use_regr=True, overlap_thresh=0.7, max_b
oxes=300)

            # note: calc_iou converts from (x1,y1,x2,y2) to (x,y,w,h) format
            # X2: bboxes that iou > C.classifier_min_overlap for all gt bboxes in 300 non_max_suppression bboxes
            # Y1: one hot code for bboxes from above => x_roi (X)
            # Y2: corresponding labels and corresponding gt bboxes
            X2, Y1, Y2, IouS = calc_iou(R, img_data, C, class_mapping)

            # If X2 is None means there are no matching bboxes
            if X2 is None:
                rpn_accuracy_rpn_monitor.append(0)
                rpn_accuracy_for_epoch.append(0)
                continue

            # Find out the positive anchors and negative anchors
            neg_samples = np.where(Y1[0, :, -1] == 1)
            pos_samples = np.where(Y1[0, :, -1] == 0)

            if len(neg_samples) > 0:
                neg_samples = neg_samples[0]
            else:
                neg_samples = []

            if len(pos_samples) > 0:
                pos_samples = pos_samples[0]
            else:
                pos_samples = []

            rpn_accuracy_rpn_monitor.append(len(pos_samples))
            rpn_accuracy_for_epoch.append((len(pos_samples)))

            if C.num_rois > 1:
                # If number of positive anchors is larger than 4//2 = 2, randomly choose 2 pos samples
                if len(pos_samples) < C.num_rois//2:
                    selected_pos_samples = pos_samples.tolist()
                else:
                    selected_pos_samples = np.random.choice(pos_samples, C.num_rois//2, replace=False).tolist()

                # Randomly choose (num_rois - num_pos) neg samples
                try:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois - len(selected_pos_samples),
replace=False).tolist()
                except:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois - len(selected_pos_samples),
replace=True).tolist()

                # Save all the pos and neg samples in sel_samples
                sel_samples = selected_pos_samples + selected_neg_samples
            else:
                # in the extreme case where num_rois = 1, we pick a random pos or neg sample
                selected_pos_samples = pos_samples.tolist()
                selected_neg_samples = neg_samples.tolist()
                if np.random.randint(0, 2):
                    sel_samples = random.choice(neg_samples)
                else:
                    sel_samples = random.choice(pos_samples)
            # training_data: [X, X2[:, sel_samples, :]]

            # labels: [Y1[:, sel_samples, :], Y2[:, sel_samples, :]]
            #  X                     => img_data resized image
            #  X2[:, sel_samples, :] => num_rois (4 in here) bboxes which contains selected neg and pos
            #  Y1[:, sel_samples, :] => one hot encode for num_rois bboxes which contains selected neg and pos
            #  Y2[:, sel_samples, :] => labels and gt bboxes for num_rois bboxes which contains selected neg and
pos
            loss_class = model_classifier.train_on_batch([X, X2[:, sel_samples, :]], [Y1[:, sel_samples, :], Y2[:
, sel_samples, :]])

            losses[iter_num, 0] = loss_rpn[1]
            losses[iter_num, 1] = loss_rpn[2]

            losses[iter_num, 2] = loss_class[1]
            losses[iter_num, 3] = loss_class[2]
            losses[iter_num, 4] = loss_class[3]

            iter_num += 1

            progbar.update(iter_num, [('rpn_cls', np.mean(losses[:iter_num, 0])), ('rpn_regr', np.mean(losses[:it
er_num, 1])),
                                      ('final_cls', np.mean(losses[:iter_num, 2])), ('final_regr', np.mean(losses
[:iter_num, 3]))])

            if iter_num == epoch_length:
                loss_rpn_cls = np.mean(losses[:, 0])
                loss_rpn_regr = np.mean(losses[:, 1])
                loss_class_cls = np.mean(losses[:, 2])
                loss_class_regr = np.mean(losses[:, 3])
                class_acc = np.mean(losses[:, 4])

                mean_overlapping_bboxes = float(sum(rpn_accuracy_for_epoch)) / len(rpn_accuracy_for_epoch)
                rpn_accuracy_for_epoch = []

                if C.verbose:
                    print('Mean number of bounding boxes from RPN overlapping ground truth boxes: {}'.format(mean
_overlapping_bboxes))
                    print('Classifier accuracy for bounding boxes from RPN: {}'.format(class_acc))
                    print('Loss RPN classifier: {}'.format(loss_rpn_cls))
                    print('Loss RPN regression: {}'.format(loss_rpn_regr))
                    print('Loss Detector classifier: {}'.format(loss_class_cls))
                    print('Loss Detector regression: {}'.format(loss_class_regr))
                    print('Total loss: {}'.format(loss_rpn_cls + loss_rpn_regr + loss_class_cls + loss_class_regr
))
                    print('Elapsed time: {}'.format(time.time() - start_time))
                    elapsed_time = (time.time()-start_time)/60

                curr_loss = loss_rpn_cls + loss_rpn_regr + loss_class_cls + loss_class_regr
                iter_num = 0
                start_time = time.time()

                if curr_loss < best_loss:
                    if C.verbose:
                        print('Total loss decreased from {} to {}, saving weights'.format(best_loss,curr_loss))
                    best_loss = curr_loss
                    model_all.save_weights(C.model_path)

                new_row = {'mean_overlapping_bboxes':round(mean_overlapping_bboxes, 3),
                           'class_acc':round(class_acc, 3),
                           'loss_rpn_cls':round(loss_rpn_cls, 3),
                           'loss_rpn_regr':round(loss_rpn_regr, 3),
                           'loss_class_cls':round(loss_class_cls, 3),
                           'loss_class_regr':round(loss_class_regr, 3),
                           'curr_loss':round(curr_loss, 3),
                           'elapsed_time':round(elapsed_time, 3),
                           'mAP': 0}

                record_df = record_df.append(new_row, ignore_index=True)
                record_df.to_csv(record_path, index=0)

                break

        except Exception as e:
            print('Exception: {}'.format(e))
            continue

print('Training complete, exiting.')
```

```
Epoch 21/60
WARNING:tensorflow:From c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\tens
orflow\python\ops\math_ops.py:3066: to_int32 (from tensorflow.python.ops.math_ops) is deprecated and
will be removed in a future version.
Instructions for updating:
```

```
Use tf.cast instead.
1000/1000 [==============================] - 6658s 7s/step - rpn_cls: 0.1160 - rpn_regr: 0.0411 - fi
nal_cls: 0.2718 - final_regr: 0.0454
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.257
Classifier accuracy for bounding boxes from RPN: 0.90125
Loss RPN classifier: 0.11807734000158983
Loss RPN regression: 0.04159095441875979
Loss Detector classifier: 0.2671130160544271
Loss Detector regression: 0.046028143291216114
Total loss: 0.47280945376599287
Elapsed time: 6658.1972188949585
Total loss decreased from 0.48100000000000004 to 0.47280945376599287, saving weights
Epoch 22/60
1000/1000 [==============================] - 6619s 7s/step - rpn_cls: 0.1186 - rpn_regr: 0.0411 - fi
nal_cls: 0.2739 - final_regr: 0.0552
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.655
Classifier accuracy for bounding boxes from RPN: 0.895
Loss RPN classifier: 0.11782723392872423
Loss RPN regression: 0.04106942688859999
Loss Detector classifier: 0.2694136625251267
Loss Detector regression: 0.05304797390440945
Total loss: 0.4813582972468604
Elapsed time: 6626.44500875473
Epoch 23/60
1000/1000 [==============================] - 6630s 7s/step - rpn_cls: 0.1272 - rpn_regr: 0.0409 - fi
nal_cls: 0.2898 - final_regr: 0.0545
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.406
Classifier accuracy for bounding boxes from RPN: 0.89825
Loss RPN classifier: 0.1207692439171513
Loss RPN regression: 0.040271821764297784
Loss Detector classifier: 0.2758692259452655
Loss Detector regression: 0.054434363960579504
Total loss: 0.49134465558729407
Elapsed time: 6629.686709880829
Epoch 24/60
1000/1000 [==============================] - 6631s 7s/step - rpn_cls: 0.1170 - rpn_regr: 0.0398 - fi
nal_cls: 0.2589 - final_regr: 0.0497
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.324
Classifier accuracy for bounding boxes from RPN: 0.9
Loss RPN classifier: 0.11595041293759849
Loss RPN regression: 0.03942981489049271
Loss Detector classifier: 0.2678287790752511
Loss Detector regression: 0.0518637368172931
Total loss: 0.4750727437206354
Elapsed time: 6630.655306816101
Epoch 25/60
1000/1000 [==============================] - 6602s 7s/step - rpn_cls: 0.1105 - rpn_regr: 0.0387 - fi
nal_cls: 0.2498 - final_regr: 0.0550
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.72
Classifier accuracy for bounding boxes from RPN: 0.90425
Loss RPN classifier: 0.11419604816053334
Loss RPN regression: 0.03847958496864885
Loss Detector classifier: 0.2505189337724223
Loss Detector regression: 0.05403771815347136
Total loss: 0.45723228505507585
Elapsed time: 6601.614814996719
Total loss decreased from 0.47280945376599287 to 0.45723228505507585, saving weights
Epoch 26/60
1000/1000 [==============================] - 6608s 7s/step - rpn_cls: 0.1068 - rpn_regr: 0.0382 - fi
nal_cls: 0.2625 - final_regr: 0.0479
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.615
Classifier accuracy for bounding boxes from RPN: 0.89975
Loss RPN classifier: 0.1102858470610948
Loss RPN regression: 0.03836969700921327
Loss Detector classifier: 0.26438278702087703
Loss Detector regression: 0.04982217071624473
Total loss: 0.4628605018074298
Elapsed time: 6614.85755109787
Epoch 27/60
1000/1000 [==============================] - 6632s 7s/step - rpn_cls: 0.1056 - rpn_regr: 0.0380 - fi
nal_cls: 0.2885 - final_regr: 0.0503
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.644
Classifier accuracy for bounding boxes from RPN: 0.89675
Loss RPN classifier: 0.10712604835032456
Loss RPN regression: 0.03757111555896699
Loss Detector classifier: 0.2770813032736187
Loss Detector regression: 0.049568512075886244
Total loss: 0.4713469792587965
Elapsed time: 6632.069177389145
Epoch 28/60
1000/1000 [==============================] - 7008s 7s/step - rpn_cls: 0.1139 - rpn_regr: 0.0379 - fi
nal_cls: 0.2539 - final_regr: 0.0554
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.569
```

```
Classifier accuracy for bounding boxes from RPN: 0.90425
Loss RPN classifier: 0.1154462793127617
Loss RPN regression: 0.03725408938527107
Loss Detector classifier: 0.2534667790173844
Loss Detector regression: 0.05125358709241846
Total loss: 0.4574207348078356
Elapsed time: 7007.908859968185
Epoch 29/60
1000/1000 [==============================] - 6815s 7s/step - rpn_cls: 0.0995 - rpn_regr: 0.0364 - fi
nal_cls: 0.2571 - final_regr: 0.0431
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.399
Classifier accuracy for bounding boxes from RPN: 0.904
Loss RPN classifier: 0.10617335651893497
Loss RPN regression: 0.036826753770466895
Loss Detector classifier: 0.26274583573415294
Loss Detector regression: 0.045050524049962404
Total loss: 0.45079647007351725
Elapsed time: 6814.92297244072
Total loss decreased from 0.45723228505507585 to 0.45079647007351725, saving weights
Epoch 30/60
1000/1000 [==============================] - 6673s 7s/step - rpn_cls: 0.1097 - rpn_regr: 0.0361 - fi
nal_cls: 0.2541 - final_regr: 0.0429
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.556
Classifier accuracy for bounding boxes from RPN: 0.90925
Loss RPN classifier: 0.11070015022043278
Loss RPN regression: 0.0363946777721867
Loss Detector classifier: 0.2427617718096153
Loss Detector regression: 0.046112146211613435
Total loss: 0.43596874601384816
Elapsed time: 6679.710359334946
Total loss decreased from 0.45079647007351725 to 0.43596874601384816, saving weights
Epoch 31/60
1000/1000 [==============================] - 6707s 7s/step - rpn_cls: 0.1045 - rpn_regr: 0.0358 - fi
nal_cls: 0.2394 - final_regr: 0.0508
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.827
Classifier accuracy for bounding boxes from RPN: 0.9075
Loss RPN classifier: 0.10291853225275975
Loss RPN regression: 0.035594304194673895
Loss Detector classifier: 0.24735158406710253
Loss Detector regression: 0.04699677362316288
Total loss: 0.4328611941376991
Elapsed time: 6713.413623571396
Total loss decreased from 0.43596874601384816 to 0.4328611941376991, saving weights
Epoch 32/60
1000/1000 [==============================] - 6689s 7s/step - rpn_cls: 0.1090 - rpn_regr: 0.0355 - fi
nal_cls: 0.2462 - final_regr: 0.0461
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.746
Classifier accuracy for bounding boxes from RPN: 0.9165
Loss RPN classifier: 0.100847793154957
Loss RPN regression: 0.03528749647084624
Loss Detector classifier: 0.24499301314618788
Loss Detector regression: 0.04559352451443556
Total loss: 0.42672182728642666
Elapsed time: 6695.565358877182
Total loss decreased from 0.4328611941376991 to 0.42672182728642666, saving weights
Epoch 33/60
1000/1000 [==============================] - 7117s 7s/step - rpn_cls: 0.1097 - rpn_regr: 0.0356 - fi
nal_cls: 0.2292 - final_regr: 0.0422
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.522
Classifier accuracy for bounding boxes from RPN: 0.90875
Loss RPN classifier: 0.10745790683930101
Loss RPN regression: 0.03513905373495072
Loss Detector classifier: 0.23251337823098583
Loss Detector regression: 0.04318725009125774
Total loss: 0.4182975888964953
Elapsed time: 7123.676680326462
Total loss decreased from 0.42672182728642666 to 0.4182975888964953, saving weights
Epoch 34/60
1000/1000 [==============================] - 7132s 7s/step - rpn_cls: 0.0988 - rpn_regr: 0.0347 - fi
nal_cls: 0.2351 - final_regr: 0.0489
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.235
Classifier accuracy for bounding boxes from RPN: 0.90475
Loss RPN classifier: 0.1020510193250756
Loss RPN regression: 0.034217009202111515
Loss Detector classifier: 0.2426972869188321
Loss Detector regression: 0.04605518557499454
Total loss: 0.42502050102101374
Elapsed time: 7138.141719341278
Epoch 35/60
1000/1000 [==============================] - 6677s 7s/step - rpn_cls: 0.1032 - rpn_regr: 0.0344 - fi
nal_cls: 0.2331 - final_regr: 0.0410
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.262
Classifier accuracy for bounding boxes from RPN: 0.91175
```

```
Loss RPN classifier: 0.10142016423171879
Loss RPN regression: 0.033969525176566095
Loss Detector classifier: 0.22888456909997695
Loss Detector regression: 0.04092641913876287
Total loss: 0.4052006776470247
Elapsed time: 6677.170350074768
Total loss decreased from 0.4182975888964953 to 0.4052006776470247, saving weights
Epoch 36/60
1000/1000 [==============================] - 6791s 7s/step - rpn_cls: 0.1006 - rpn_regr: 0.0342 - fi
nal_cls: 0.2491 - final_regr: 0.0535
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.252
Classifier accuracy for bounding boxes from RPN: 0.90875
Loss RPN classifier: 0.10443224312443374
Loss RPN regression: 0.03369969276385382
Loss Detector classifier: 0.240261354833332
Loss Detector regression: 0.05206727728797705
Total loss: 0.4304605680095966
Elapsed time: 6797.391136884689
Epoch 37/60
1000/1000 [==============================] - 6695s 7s/step - rpn_cls: 0.0924 - rpn_regr: 0.0325 - fi
nal_cls: 0.2046 - final_regr: 0.0352
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.017
Classifier accuracy for bounding boxes from RPN: 0.914
Loss RPN classifier: 0.09813417067534556
Loss RPN regression: 0.03288363159634173
Loss Detector classifier: 0.22372457921011665
Loss Detector regression: 0.036990514667530075
Total loss: 0.391732896149334
Elapsed time: 6695.0364990234375
Total loss decreased from 0.4052006776470247 to 0.391732896149334, saving weights
Epoch 38/60
1000/1000 [==============================] - 6659s 7s/step - rpn_cls: 0.0951 - rpn_regr: 0.0326 - fi
nal_cls: 0.2515 - final_regr: 0.0375
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.04
Classifier accuracy for bounding boxes from RPN: 0.91175
Loss RPN classifier: 0.10048948921331995
Loss RPN regression: 0.0324393208399415
Loss Detector classifier: 0.23213777049977943
Loss Detector regression: 0.04273577811778523
Total loss: 0.4078023586708261
Elapsed time: 6665.669176578522
Epoch 39/60
1000/1000 [==============================] - 6985s 7s/step - rpn_cls: 0.1062 - rpn_regr: 0.0324 - fi
nal_cls: 0.2503 - final_regr: 0.0396
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.162
Classifier accuracy for bounding boxes from RPN: 0.90975
Loss RPN classifier: 0.10159578559300268
Loss RPN regression: 0.03218281424511224
Loss Detector classifier: 0.24879504242786787
Loss Detector regression: 0.04449890523951035
Total loss: 0.4270725475054931
Elapsed time: 6985.484578132629
Epoch 40/60
1000/1000 [==============================] - 6713s 7s/step - rpn_cls: 0.0995 - rpn_regr: 0.0326 - fi
nal_cls: 0.2079 - final_regr: 0.0414
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.02
Classifier accuracy for bounding boxes from RPN: 0.922
Loss RPN classifier: 0.09965221582472605
Loss RPN regression: 0.03220390038844198
Loss Detector classifier: 0.21357021467956655
Loss Detector regression: 0.040151872951100814
Total loss: 0.38557820384383534
Elapsed time: 6712.984713554382
Total loss decreased from 0.391732896149334 to 0.38557820384383534, saving weights
Epoch 41/60
1000/1000 [==============================] - 6734s 7s/step - rpn_cls: 0.1035 - rpn_regr: 0.0317 - fi
nal_cls: 0.2129 - final_regr: 0.0405
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.648
Classifier accuracy for bounding boxes from RPN: 0.91975
Loss RPN classifier: 0.09946661539159732
Loss RPN regression: 0.03151931333076209
Loss Detector classifier: 0.22167023908011468
Loss Detector regression: 0.04201200652933039
Total loss: 0.3946681743318045
Elapsed time: 6740.639261007309
Epoch 42/60
1000/1000 [==============================] - 6636s 7s/step - rpn_cls: 0.0855 - rpn_regr: 0.0321 - fi
nal_cls: 0.2213 - final_regr: 0.0396
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.394
Classifier accuracy for bounding boxes from RPN: 0.91825
Loss RPN classifier: 0.09255007463199007
Loss RPN regression: 0.031731647902634
Loss Detector classifier: 0.22415420075099973
Loss Detector regression: 0.03859008222282864
Total loss: 0.38702600550845245
Elapsed time: 6635.649347782135
Epoch 43/60
 369/1000 [==========>...................] - ETA: 1:09:41 - rpn_cls: 0.0991 - rpn_regr: 0.0314 - fin
al_cls: 0.2094 - final_regr: 0.0408

---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-29-493fa9600e77> in <module>
     89             #  Y1[:, sel_samples, :] => one hot encode for num_rois bboxes which contains se
lected neg and pos
     90             #  Y2[:, sel_samples, :] => labels and gt bboxes for num_rois bboxes which conta
ins selected neg and pos
---> 91             loss_class = model_classifier.train_on_batch([X, X2[:, sel_samples, :]], [Y1[:,
sel_samples, :], Y2[:, sel_samples, :]])
     92 
     93             losses[iter_num, 0] = loss_rpn[1]

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\keras\engine\training.py in
train_on_batch(self, x, y, sample_weight, class_weight)
   1215             ins = x + y + sample_weights
   1216         self._make_train_function()
-> 1217         outputs = self.train_function(ins)
   1218         return unpack_singleton(outputs)
   1219 

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\keras\backend\tensorflow_bac
kend.py in __call__(self, inputs)
   2713                 return self._legacy_call(inputs)
   2714 
-> 2715             return self._call(inputs)
   2716         else:
   2717             if py_any(is_tensor(x) for x in inputs):

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\keras\backend\tensorflow_bac
kend.py in _call(self, inputs)
   2673             fetched = self._callable_fn(*array_vals, run_metadata=self.run_metadata)
   2674         else:
-> 2675             fetched = self._callable_fn(*array_vals)
   2676         return fetched[:len(self.outputs)]
   2677 

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\tensorflow\python\client\ses
sion.py in __call__(self, *args, **kwargs)
   1437           ret = tf_session.TF_SessionRunCallable(
   1438               self._session._session, self._handle, args, status,
-> 1439               run_metadata_ptr)
   1440         if run_metadata:
   1441           proto_data = tf_session.TF_GetBuffer(run_metadata_ptr)

KeyboardInterrupt: 
```

In [30]:

```python
start_time = time.time()
for epoch_num in range(num_epochs):

    progbar = generic_utils.Progbar(epoch_length)
    print('Epoch {}/{}'.format(r_epochs + 1, total_epochs))

    r_epochs += 1

    while True:
        try:

            if len(rpn_accuracy_rpn_monitor) == epoch_length and C.verbose:
                mean_overlapping_bboxes = float(sum(rpn_accuracy_rpn_monitor))/len(rpn_accuracy_rpn_monitor)
                rpn_accuracy_rpn_monitor = []
#                 print('Average number of overlapping bounding boxes from RPN = {} for {} previous iterations'.f
ormat(mean_overlapping_bboxes, epoch_length))
                if mean_overlapping_bboxes == 0:
                    print('RPN is not producing bounding boxes that overlap the ground truth boxes. Check RPN set
tings or keep training.')

            # Generate X (x_img) and label Y ([y_rpn_cls, y_rpn_regr])
            X, Y, img_data, debug_img, debug_num_pos = next(data_gen_train)

            # Train rpn model and get loss value [_, loss_rpn_cls, loss_rpn_regr]
            loss_rpn = model_rpn.train_on_batch(X, Y)

            # Get predicted rpn from rpn model [rpn_cls, rpn_regr]
            P_rpn = model_rpn.predict_on_batch(X)
            # R: bboxes (shape=(300,4))

            # Convert rpn layer to roi bboxes
            R = rpn_to_roi(P_rpn[0], P_rpn[1], C, K.image_data_format(), use_regr=True, overlap_thresh=0.7, max_b
oxes=300)

            # note: calc_iou converts from (x1,y1,x2,y2) to (x,y,w,h) format
            # X2: bboxes that iou > C.classifier_min_overlap for all gt bboxes in 300 non_max_suppression bboxes
            # Y1: one hot code for bboxes from above => x_roi (X)
            # Y2: corresponding labels and corresponding gt bboxes
            X2, Y1, Y2, IouS = calc_iou(R, img_data, C, class_mapping)

            # If X2 is None means there are no matching bboxes
            if X2 is None:
                rpn_accuracy_rpn_monitor.append(0)
                rpn_accuracy_for_epoch.append(0)
                continue

            # Find out the positive anchors and negative anchors
            neg_samples = np.where(Y1[0, :, -1] == 1)
            pos_samples = np.where(Y1[0, :, -1] == 0)

            if len(neg_samples) > 0:
                neg_samples = neg_samples[0]
            else:
                neg_samples = []

            if len(pos_samples) > 0:
                pos_samples = pos_samples[0]
            else:
                pos_samples = []

            rpn_accuracy_rpn_monitor.append(len(pos_samples))
            rpn_accuracy_for_epoch.append((len(pos_samples)))

            if C.num_rois > 1:
                # If number of positive anchors is larger than 4//2 = 2, randomly choose 2 pos samples
                if len(pos_samples) < C.num_rois//2:
                    selected_pos_samples = pos_samples.tolist()
                else:
                    selected_pos_samples = np.random.choice(pos_samples, C.num_rois//2, replace=False).tolist()

                # Randomly choose (num_rois - num_pos) neg samples
                try:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois - len(selected_pos_samples),
replace=False).tolist()
                except:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois - len(selected_pos_samples),
replace=True).tolist()

                # Save all the pos and neg samples in sel_samples
                sel_samples = selected_pos_samples + selected_neg_samples
            else:
                # in the extreme case where num_rois = 1, we pick a random pos or neg sample
                selected_pos_samples = pos_samples.tolist()
                selected_neg_samples = neg_samples.tolist()
                if np.random.randint(0, 2):
                    sel_samples = random.choice(neg_samples)
                else:
                    sel_samples = random.choice(pos_samples)

            # training_data: [X, X2[:, sel_samples, :]]
            # labels: [Y1[:, sel_samples, :], Y2[:, sel_samples, :]]
            #  X                     => img_data resized image
            #  X2[:, sel_samples, :] => num_rois (4 in here) bboxes which contains selected neg and pos
            #  Y1[:, sel_samples, :] => one hot encode for num_rois bboxes which contains selected neg and pos
            #  Y2[:, sel_samples, :] => labels and gt bboxes for num_rois bboxes which contains selected neg and
pos
            loss_class = model_classifier.train_on_batch([X, X2[:, sel_samples, :]], [Y1[:, sel_samples, :], Y2[:
, sel_samples, :]])

            losses[iter_num, 0] = loss_rpn[1]
            losses[iter_num, 1] = loss_rpn[2]

            losses[iter_num, 2] = loss_class[1]
            losses[iter_num, 3] = loss_class[2]
            losses[iter_num, 4] = loss_class[3]

            iter_num += 1

            progbar.update(iter_num, [('rpn_cls', np.mean(losses[:iter_num, 0])), ('rpn_regr', np.mean(losses[:it
er_num, 1])),
                                      ('final_cls', np.mean(losses[:iter_num, 2])), ('final_regr', np.mean(losses
[:iter_num, 3]))])
```

```
            if iter_num == epoch_length:
                loss_rpn_cls = np.mean(losses[:, 0])
                loss_rpn_regr = np.mean(losses[:, 1])
                loss_class_cls = np.mean(losses[:, 2])
                loss_class_regr = np.mean(losses[:, 3])
                class_acc = np.mean(losses[:, 4])

                mean_overlapping_bboxes = float(sum(rpn_accuracy_for_epoch)) / len(rpn_accuracy_for_epoch)
                rpn_accuracy_for_epoch = []

                if C.verbose:
                    print('Mean number of bounding boxes from RPN overlapping ground truth boxes: {}'.format(mean
_overlapping_bboxes))
                    print('Classifier accuracy for bounding boxes from RPN: {}'.format(class_acc))
                    print('Loss RPN classifier: {}'.format(loss_rpn_cls))
                    print('Loss RPN regression: {}'.format(loss_rpn_regr))
                    print('Loss Detector classifier: {}'.format(loss_class_cls))
                    print('Loss Detector regression: {}'.format(loss_class_regr))
                    print('Total loss: {}'.format(loss_rpn_cls + loss_rpn_regr + loss_class_cls + loss_class_regr
))
                    print('Elapsed time: {}'.format(time.time() - start_time))
                    elapsed_time = (time.time()-start_time)/60

                curr_loss = loss_rpn_cls + loss_rpn_regr + loss_class_cls + loss_class_regr
                iter_num = 0
                start_time = time.time()

                if curr_loss < best_loss:
                    if C.verbose:
                        print('Total loss decreased from {} to {}, saving weights'.format(best_loss,curr_loss))
                    best_loss = curr_loss
                    model_all.save_weights(C.model_path)

                new_row = {'mean_overlapping_bboxes':round(mean_overlapping_bboxes, 3),
                           'class_acc':round(class_acc, 3),
                           'loss_rpn_cls':round(loss_rpn_cls, 3),
                           'loss_rpn_regr':round(loss_rpn_regr, 3),
                           'loss_class_cls':round(loss_class_cls, 3),
                           'loss_class_regr':round(loss_class_regr, 3),
                           'curr_loss':round(curr_loss, 3),
                           'elapsed_time':round(elapsed_time, 3),
                           'mAP': 0}

                record_df = record_df.append(new_row, ignore_index=True)
                record_df.to_csv(record_path, index=0)

                break

        except Exception as e:
            print('Exception: {}'.format(e))
            continue

print('Training complete, exiting.')
```

```
Epoch 44/60
1000/1000 [==============================] - 4112s 4s/step - rpn_cls: 0.0968 - rpn_regr: 0.0310 - fi
nal_cls: 0.2313 - final_regr: 0.0406
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.732267732267733
Classifier accuracy for bounding boxes from RPN: 0.91675
Loss RPN classifier: 0.09480264727612873
Loss RPN regression: 0.030832126575522124
Loss Detector classifier: 0.2313676630154514
Loss Detector regression: 0.04163726036626031
Total loss: 0.39863969723336257
Elapsed time: 4112.340172529221
Epoch 45/60
1000/1000 [==============================] - 6491s 6s/step - rpn_cls: 0.0960 - rpn_regr: 0.0316 - fi
nal_cls: 0.2268 - final_regr: 0.0416
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.513
Classifier accuracy for bounding boxes from RPN: 0.913
Loss RPN classifier: 0.1013936174448152
Loss RPN regression: 0.03132245784485713
Loss Detector classifier: 0.21898887856697546
Loss Detector regression: 0.03721406426321482
Total loss: 0.3889190181198626
Elapsed time: 6491.239631414413
Epoch 46/60
1000/1000 [==============================] - 6468s 6s/step - rpn_cls: 0.0887 - rpn_regr: 0.0305 - fi
nal_cls: 0.2238 - final_regr: 0.0414
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.555
Classifier accuracy for bounding boxes from RPN: 0.9115
Loss RPN classifier: 0.09526262625788308
Loss RPN regression: 0.030720088307280093
Loss Detector classifier: 0.22758960149202176
Loss Detector regression: 0.04345497762534069
Total loss: 0.3970272936825256
Elapsed time: 6468.121001005173
Epoch 47/60
1000/1000 [==============================] - 6485s 6s/step - rpn_cls: 0.0983 - rpn_regr: 0.0300 - fi
nal_cls: 0.2253 - final_regr: 0.0428
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.337
Classifier accuracy for bounding boxes from RPN: 0.91675
Loss RPN classifier: 0.10031528114789791
Loss RPN regression: 0.030573817584197967
Loss Detector classifier: 0.2163071359490241
Loss Detector regression: 0.04019758452288807
Total loss: 0.387393819204008
Elapsed time: 6485.168669223785
Epoch 48/60
1000/1000 [==============================] - 6559s 7s/step - rpn_cls: 0.0896 - rpn_regr: 0.0303 - fi
nal_cls: 0.2119 - final_regr: 0.0378
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.78
Classifier accuracy for bounding boxes from RPN: 0.918
Loss RPN classifier: 0.08977624706713519
Loss RPN regression: 0.030273045294452457
Loss Detector classifier: 0.22148577985723386
Loss Detector regression: 0.03855257619940676
Total loss: 0.3800876484182283
Elapsed time: 6559.0743288993835
Total loss decreased from 0.38557820384383534 to 0.3800876484182283, saving weights
Epoch 49/60
1000/1000 [==============================] - 6513s 7s/step - rpn_cls: 0.0970 - rpn_regr: 0.0305 - fi
nal_cls: 0.2466 - final_regr: 0.0409
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.792
Classifier accuracy for bounding boxes from RPN: 0.917
Loss RPN classifier: 0.09325333117622031
Loss RPN regression: 0.030040268636308612
Loss Detector classifier: 0.2176199974304218
Loss Detector regression: 0.03907573258181219
Total loss: 0.3799893298247629
Elapsed time: 6519.672699689865
Total loss decreased from 0.3800876484182283 to 0.3799893298247629, saving weights
Epoch 50/60
1000/1000 [==============================] - 6538s 7s/step - rpn_cls: 0.0986 - rpn_regr: 0.0301 - fi
nal_cls: 0.2329 - final_regr: 0.0403
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.26
Classifier accuracy for bounding boxes from RPN: 0.913
Loss RPN classifier: 0.09807945486999675
Loss RPN regression: 0.03022242967132479
Loss Detector classifier: 0.22925493818837275
Loss Detector regression: 0.03941572333170916
Total loss: 0.3969725460614035
Elapsed time: 6544.05663895607
Epoch 51/60
1000/1000 [==============================] - 6543s 7s/step - rpn_cls: 0.0860 - rpn_regr: 0.0299 - fi
nal_cls: 0.2261 - final_regr: 0.0427
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.297
Classifier accuracy for bounding boxes from RPN: 0.9215
Loss RPN classifier: 0.09165163525424776
Loss RPN regression: 0.029743433545809238
Loss Detector classifier: 0.2220420637607167
Loss Detector regression: 0.04080049236839113
Total loss: 0.3842376249291648
Elapsed time: 6543.292579650879
Epoch 52/60
1000/1000 [==============================] - 6567s 7s/step - rpn_cls: 0.0931 - rpn_regr: 0.0293 - fi
nal_cls: 0.2026 - final_regr: 0.0358
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.159
Classifier accuracy for bounding boxes from RPN: 0.92225
Loss RPN classifier: 0.0916378595904703
Loss RPN regression: 0.02942511829920113
Loss Detector classifier: 0.21518157064427942
Loss Detector regression: 0.037811569972880536
Total loss: 0.3740561185068314
Elapsed time: 6567.345048427582
Total loss decreased from 0.3799893298247629 to 0.3740561185068314, saving weights
Epoch 53/60
1000/1000 [==============================] - 6591s 7s/step - rpn_cls: 0.1065 - rpn_regr: 0.0300 - fi
nal_cls: 0.2361 - final_regr: 0.0369
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.083
Classifier accuracy for bounding boxes from RPN: 0.915
Loss RPN classifier: 0.09882478144480818
Loss RPN regression: 0.02927976182475686
Loss Detector classifier: 0.2374572048778209
Loss Detector regression: 0.04129856610603747
```

```
Total loss: 0.40686031425342345
Elapsed time: 6597.741457700729
Epoch 54/60
1000/1000 [==============================] - 6542s 7s/step - rpn_cls: 0.0850 - rpn_regr: 0.0288 - fi
nal_cls: 0.2274 - final_regr: 0.0369
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.688
Classifier accuracy for bounding boxes from RPN: 0.9145
Loss RPN classifier: 0.09353903577013213
Loss RPN regression: 0.029075160819571465
Loss Detector classifier: 0.23022568281030545
Loss Detector regression: 0.03780363255699922
Total loss: 0.39064351195700825
Elapsed time: 6542.307960271835
Epoch 55/60
1000/1000 [==============================] - 6517s 7s/step - rpn_cls: 0.0897 - rpn_regr: 0.0283 - fi
nal_cls: 0.2143 - final_regr: 0.0353
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.915
Classifier accuracy for bounding boxes from RPN: 0.917
Loss RPN classifier: 0.0892308457678817
Loss RPN regression: 0.028916025575250387
Loss Detector classifier: 0.22323644369255635
Loss Detector regression: 0.040438077310245714
Total loss: 0.38182139234593415
Elapsed time: 6517.243902921677
Epoch 56/60
1000/1000 [==============================] - 6675s 7s/step - rpn_cls: 0.0944 - rpn_regr: 0.0282 - fi
nal_cls: 0.1945 - final_regr: 0.0398
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.055
Classifier accuracy for bounding boxes from RPN: 0.9165
Loss RPN classifier: 0.09500249721317915
Loss RPN regression: 0.028564296107273547
Loss Detector classifier: 0.21244981669155277
Loss Detector regression: 0.03741885763384926
Total loss: 0.3734354676458547
Elapsed time: 6674.747141599655
Total loss decreased from 0.3740561185068314 to 0.3734354676458547, saving weights
Epoch 57/60
1000/1000 [==============================] - 7154s 7s/step - rpn_cls: 0.0869 - rpn_regr: 0.0290 - fi
nal_cls: 0.2597 - final_regr: 0.0444
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.563
Classifier accuracy for bounding boxes from RPN: 0.915
Loss RPN classifier: 0.08741258042244042
Loss RPN regression: 0.028741906793788075
Loss Detector classifier: 0.23332779714961543
Loss Detector regression: 0.042586724686610976
Total loss: 0.3920690090524549
Elapsed time: 7160.246055364609
Epoch 58/60
1000/1000 [==============================] - 7068s 7s/step - rpn_cls: 0.0877 - rpn_regr: 0.0283 - fi
nal_cls: 0.2406 - final_regr: 0.0383
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.978
Classifier accuracy for bounding boxes from RPN: 0.91275
Loss RPN classifier: 0.09007062605315588
Loss RPN regression: 0.028226366539951413
Loss Detector classifier: 0.23168107176774355
Loss Detector regression: 0.035717669277248204
Total loss: 0.38569573363809906
Elapsed time: 7068.10417842865
Epoch 59/60
1000/1000 [==============================] - 12864s 13s/step - rpn_cls: 0.0846 - rpn_regr: 0.0280 -
final_cls: 0.2156 - final_regr: 0.0374
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.187
Classifier accuracy for bounding boxes from RPN: 0.92075
Loss RPN classifier: 0.09067106092443782
Loss RPN regression: 0.028097780546173453
Loss Detector classifier: 0.21660219645729376
Loss Detector regression: 0.033900706659682325
Total loss: 0.3692717445875873
Elapsed time: 12864.175436258316
Total loss decreased from 0.3734354676458547 to 0.3692717445875873, saving weights
Epoch 60/60
1000/1000 [==============================] - 13579s 14s/step - rpn_cls: 0.0981 - rpn_regr: 0.0282 -
final_cls: 0.2163 - final_regr: 0.0386
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.019
Classifier accuracy for bounding boxes from RPN: 0.925
Loss RPN classifier: 0.09359698915168893
Loss RPN regression: 0.027945981163065882
Loss Detector classifier: 0.20443624131978141
Loss Detector regression: 0.03719004324974958
Total loss: 0.3631692548842858
Elapsed time: 13584.201281785965
Total loss decreased from 0.3692717445875873 to 0.3631692548842858, saving weights
Epoch 61/60
```

```
1000/1000 [==============================] - 9791s 10s/step - rpn_cls: 0.1007 - rpn_regr: 0.0284 - f
inal_cls: 0.2235 - final_regr: 0.0414
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.18
Classifier accuracy for bounding boxes from RPN: 0.9185
Loss RPN classifier: 0.09230075119745491
Loss RPN regression: 0.027733842933550478
Loss Detector classifier: 0.22442549457127461
Loss Detector regression: 0.04125755919120275
Total loss: 0.38571764789348273
Elapsed time: 9796.864559173584
Epoch 62/60
1000/1000 [==============================] - 8085s 8s/step - rpn_cls: 0.0916 - rpn_regr: 0.0277 - fi
nal_cls: 0.2134 - final_regr: 0.0331
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.258
Classifier accuracy for bounding boxes from RPN: 0.921
Loss RPN classifier: 0.091740586793692
Loss RPN regression: 0.02772451371187344
Loss Detector classifier: 0.1943105812009162
Loss Detector regression: 0.035450264052931744
Total loss: 0.34922594575941335
Elapsed time: 8084.806133270264
Total loss decreased from 0.3631692548842858 to 0.34922594575941335, saving weights
Epoch 63/60
 818/1000 [=======================>......] - ETA: 23:33 - rpn_cls: 0.0826 - rpn_regr: 0.0273 - final
_cls: 0.2155 - final_regr: 0.0359

---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-30-493fa9600e77> in <module>
     89             #  Y1[:, sel_samples, :] => one hot encode for num_rois bboxes which contains se
lected neg and pos
     90             #  Y2[:, sel_samples, :] => labels and gt bboxes for num_rois bboxes which conta
ins selected neg and pos
---> 91             loss_class = model_classifier.train_on_batch([X, X2[:, sel_samples, :]], [Y1[:,
sel_samples, :], Y2[:, sel_samples, :]])
     92 
     93             losses[iter_num, 0] = loss_rpn[1]

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\keras\engine\training.py in
train_on_batch(self, x, y, sample_weight, class_weight)
   1215             ins = x + y + sample_weights
   1216         self._make_train_function()
-> 1217         outputs = self.train_function(ins)
   1218         return unpack_singleton(outputs)
   1219 

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\keras\backend\tensorflow_bac
kend.py in __call__(self, inputs)
   2713                 return self._legacy_call(inputs)
   2714 
-> 2715             return self._call(inputs)
   2716         else:
   2717             if py_any(is_tensor(x) for x in inputs):

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\keras\backend\tensorflow_bac
kend.py in _call(self, inputs)
   2673             fetched = self._callable_fn(*array_vals, run_metadata=self.run_metadata)
   2674         else:
-> 2675             fetched = self._callable_fn(*array_vals)
   2676         return fetched[:len(self.outputs)]
   2677 

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\tensorflow\python\client\ses
sion.py in __call__(self, *args, **kwargs)
   1437           ret = tf_session.TF_SessionRunCallable(
   1438               self._session._session, self._handle, args, status,
-> 1439               run_metadata_ptr)
   1440         if run_metadata:
   1441           proto_data = tf_session.TF_GetBuffer(run_metadata_ptr)

KeyboardInterrupt: 
```

In [38]:

```
start_time = time.time()
for epoch_num in range(num_epochs):

    progbar = generic_utils.Progbar(epoch_length)
    print('Epoch {}/{}'.format(r_epochs + 1, total_epochs))

    r_epochs += 1

    while True:
        try:
            if len(rpn_accuracy_rpn_monitor) == epoch_length and C.verbose:
                mean_overlapping_bboxes = float(sum(rpn_accuracy_rpn_monitor))/len(rpn_accuracy_rpn_monitor)
                rpn_accuracy_rpn_monitor = []
#                 print('Average number of overlapping bounding boxes from RPN = {} for {} previous iterations'.f
ormat(mean_overlapping_bboxes, epoch_length))
                if mean_overlapping_bboxes == 0:
                    print('RPN is not producing bounding boxes that overlap the ground truth boxes. Check RPN set
tings or keep training.')

            # Generate X (x_img) and label Y ([y_rpn_cls, y_rpn_regr])
            X, Y, img_data, debug_img, debug_num_pos = next(data_gen_train)

            # Train rpn model and get loss value [_, loss_rpn_cls, loss_rpn_regr]
            loss_rpn = model_rpn.train_on_batch(X, Y)

            # Get predicted rpn from rpn model [rpn_cls, rpn_regr]
            P_rpn = model_rpn.predict_on_batch(X)

            # R: bboxes (shape=(300,4))
            # Convert rpn layer to roi bboxes
            R = rpn_to_roi(P_rpn[0], P_rpn[1], C, K.image_data_format(), use_regr=True, overlap_thresh=0.7, max_b
oxes=300)

            # note: calc_iou converts from (x1,y1,x2,y2) to (x,y,w,h) format
            # X2: bboxes that iou > C.classifier_min_overlap for all gt bboxes in 300 non_max_suppression bboxes
            # Y1: one hot code for bboxes from above => x_roi (X)
            # Y2: corresponding labels and corresponding gt bboxes
            X2, Y1, Y2, IouS = calc_iou(R, img_data, C, class_mapping)

            # If X2 is None means there are no matching bboxes
            if X2 is None:
                rpn_accuracy_rpn_monitor.append(0)
                rpn_accuracy_for_epoch.append(0)
                continue

            # Find out the positive anchors and negative anchors
            neg_samples = np.where(Y1[0, :, -1] == 1)
            pos_samples = np.where(Y1[0, :, -1] == 0)

            if len(neg_samples) > 0:
                neg_samples = neg_samples[0]
            else:
                neg_samples = []

            if len(pos_samples) > 0:
                pos_samples = pos_samples[0]
            else:
                pos_samples = []

            rpn_accuracy_rpn_monitor.append(len(pos_samples))
            rpn_accuracy_for_epoch.append((len(pos_samples)))

            if C.num_rois > 1:
                # If number of positive anchors is larger than 4//2 = 2, randomly choose 2 pos samples
                if len(pos_samples) < C.num_rois//2:
                    selected_pos_samples = pos_samples.tolist()
                else:
                    selected_pos_samples = np.random.choice(pos_samples, C.num_rois//2, replace=False).tolist()

                # Randomly choose (num_rois - num_pos) neg samples
                try:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois - len(selected_pos_samples),
replace=False).tolist()
                except:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois - len(selected_pos_samples),
replace=True).tolist()

                # Save all the pos and neg samples in sel_samples
                sel_samples = selected_pos_samples + selected_neg_samples
            else:
                # in the extreme case where num_rois = 1, we pick a random pos or neg sample
                selected_pos_samples = pos_samples.tolist()
                selected_neg_samples = neg_samples.tolist()
                if np.random.randint(0, 2):
                    sel_samples = random.choice(neg_samples)
                else:
                    sel_samples = random.choice(pos_samples)

            # training_data: [X, X2[:, sel_samples, :]]
            # labels: [Y1[:, sel_samples, :], Y2[:, sel_samples, :]]
            #  X                     => img_data resized image
            #  X2[:, sel_samples, :] => num_rois (4 in here) bboxes which contains selected neg and pos
```

```python
            #  Y1[:, sel_samples, :] => one hot encode for num_rois bboxes which contains selected neg and pos

            #  Y2[:, sel_samples, :] => labels and gt bboxes for num_rois bboxes which contains selected neg and
pos
            loss_class = model_classifier.train_on_batch([X, X2[:, sel_samples, :]], [Y1[:, sel_samples, :], Y2[:
, sel_samples, :]])

            losses[iter_num, 0] = loss_rpn[1]
            losses[iter_num, 1] = loss_rpn[2]

            losses[iter_num, 2] = loss_class[1]
            losses[iter_num, 3] = loss_class[2]
            losses[iter_num, 4] = loss_class[3]

            iter_num += 1

            progbar.update(iter_num, [('rpn_cls', np.mean(losses[:iter_num, 0])), ('rpn_regr', np.mean(losses[:it
er_num, 1])),
                                      ('final_cls', np.mean(losses[:iter_num, 2])), ('final_regr', np.mean(losses
[:iter_num, 3]))])

            if iter_num == epoch_length:
                loss_rpn_cls = np.mean(losses[:, 0])
                loss_rpn_regr = np.mean(losses[:, 1])
                loss_class_cls = np.mean(losses[:, 2])
                loss_class_regr = np.mean(losses[:, 3])
                class_acc = np.mean(losses[:, 4])

                mean_overlapping_bboxes = float(sum(rpn_accuracy_for_epoch)) / len(rpn_accuracy_for_epoch)
                rpn_accuracy_for_epoch = []

                if C.verbose:
                    print('Mean number of bounding boxes from RPN overlapping ground truth boxes: {}'.format(mean
_overlapping_bboxes))
                    print('Classifier accuracy for bounding boxes from RPN: {}'.format(class_acc))
                    print('Loss RPN classifier: {}'.format(loss_rpn_cls))
                    print('Loss RPN regression: {}'.format(loss_rpn_regr))
                    print('Loss Detector classifier: {}'.format(loss_class_cls))
                    print('Loss Detector regression: {}'.format(loss_class_regr))
                    print('Total loss: {}'.format(loss_rpn_cls + loss_rpn_regr + loss_class_cls + loss_class_regr
))
                    print('Elapsed time: {}'.format(time.time() - start_time))
                    elapsed_time = (time.time()-start_time)/60

                curr_loss = loss_rpn_cls + loss_rpn_regr + loss_class_cls + loss_class_regr
                iter_num = 0
                start_time = time.time()

                if curr_loss < best_loss:
                    if C.verbose:
                        print('Total loss decreased from {} to {}, saving weights'.format(best_loss,curr_loss))
                    best_loss = curr_loss
                    model_all.save_weights(C.model_path)

                new_row = {'mean_overlapping_bboxes':round(mean_overlapping_bboxes, 3),
                           'class_acc':round(class_acc, 3),
                           'loss_rpn_cls':round(loss_rpn_cls, 3),
                           'loss_rpn_regr':round(loss_rpn_regr, 3),
                           'loss_class_cls':round(loss_class_cls, 3),
                           'loss_class_regr':round(loss_class_regr, 3),
                           'curr_loss':round(curr_loss, 3),
                           'elapsed_time':round(elapsed_time, 3),
                           'mAP': 0}

                record_df = record_df.append(new_row, ignore_index=True)
                record_df.to_csv(record_path, index=0)

                break

        except Exception as e:
            print('Exception: {}'.format(e))
            continue

print('Training complete, exiting.')
```

```
Epoch 71/60
1000/1000 [==============================] - 4857s 5s/step - rpn_cls: 0.0898 - rpn_regr: 0.0263 - fi
nal_cls: 0.2064 - final_regr: 0.0344
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.82817182817183
Classifier accuracy for bounding boxes from RPN: 0.91975
Loss RPN classifier: 0.09022031766299607
Loss RPN regression: 0.026486909247236325
Loss Detector classifier: 0.21372186969842005
Loss Detector regression: 0.0363265582160966
```

```
Total loss: 0.36675565482474903
Elapsed time: 4856.786927938461
Epoch 72/60
1000/1000 [==============================] - 6588s 7s/step - rpn_cls: 0.0902 - rpn_regr: 0.0260 - fi
nal_cls: 0.2003 - final_regr: 0.0320
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.639
Classifier accuracy for bounding boxes from RPN: 0.9205
Loss RPN classifier: 0.09165122284271056
Loss RPN regression: 0.026410833253059537
Loss Detector classifier: 0.21670526572967355
Loss Detector regression: 0.03268067950633122
Total loss: 0.3674480013317749
Elapsed time: 6588.502975940704
Epoch 73/60
1000/1000 [==============================] - 6406s 6s/step - rpn_cls: 0.0893 - rpn_regr: 0.0265 - fi
nal_cls: 0.1848 - final_regr: 0.0357
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.85
Classifier accuracy for bounding boxes from RPN: 0.926
Loss RPN classifier: 0.08670951823089974
Loss RPN regression: 0.026488075774163008
Loss Detector classifier: 0.19481045832968813
Loss Detector regression: 0.034301459086113026
Total loss: 0.3423095114208639
Elapsed time: 6406.508780002594
Epoch 74/60
1000/1000 [==============================] - 6385s 6s/step - rpn_cls: 0.0821 - rpn_regr: 0.0263 - fi
nal_cls: 0.2136 - final_regr: 0.0371
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.746
Classifier accuracy for bounding boxes from RPN: 0.92325
Loss RPN classifier: 0.0857051216614807
Loss RPN regression: 0.02619671509694308
Loss Detector classifier: 0.20954990848651506
Loss Detector regression: 0.03280616429602378
Total loss: 0.3542579095409626
Elapsed time: 6385.03294301033
Epoch 75/60
1000/1000 [==============================] - 6886s 7s/step - rpn_cls: 0.0833 - rpn_regr: 0.0256 - fi
nal_cls: 0.2089 - final_regr: 0.0339
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.984
Classifier accuracy for bounding boxes from RPN: 0.92475
Loss RPN classifier: 0.08896835664981836
Loss RPN regression: 0.0256532618268393
Loss Detector classifier: 0.20648167268144424
Loss Detector regression: 0.03369727983717166
Total loss: 0.3548005709952735
Elapsed time: 6886.030679225922
Epoch 76/60
1000/1000 [==============================] - 8518s 9s/step - rpn_cls: 0.0846 - rpn_regr: 0.0260 - fi
nal_cls: 0.2104 - final_regr: 0.0404
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.027
Classifier accuracy for bounding boxes from RPN: 0.921
Loss RPN classifier: 0.08351130840361266
Loss RPN regression: 0.02565418884344399
Loss Detector classifier: 0.21655876730852425
Loss Detector regression: 0.03829951621680811
Total loss: 0.364023780772389
Elapsed time: 8517.655005455017
Epoch 77/60
1000/1000 [==============================] - 7159s 7s/step - rpn_cls: 0.0894 - rpn_regr: 0.0256 - fi
nal_cls: 0.2144 - final_regr: 0.0273
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.128
Classifier accuracy for bounding boxes from RPN: 0.925
Loss RPN classifier: 0.08668458760272245
Loss RPN regression: 0.025463407805189492
Loss Detector classifier: 0.2084601966033588
Loss Detector regression: 0.030844870865010306
Total loss: 0.35145306287628103
Elapsed time: 7158.766051769257
Epoch 78/60
1000/1000 [==============================] - 6741s 7s/step - rpn_cls: 0.0862 - rpn_regr: 0.0254 - fi
nal_cls: 0.1909 - final_regr: 0.0327
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.916
Classifier accuracy for bounding boxes from RPN: 0.92875
Loss RPN classifier: 0.09105294343576026
Loss RPN regression: 0.025722461233846843
Loss Detector classifier: 0.20501419723482467
Loss Detector regression: 0.03442757703360985
Total loss: 0.3562171789380416
Elapsed time: 6741.2857921123505
Epoch 79/60
1000/1000 [==============================] - 6632s 7s/step - rpn_cls: 0.0835 - rpn_regr: 0.0252 - fi
nal_cls: 0.2086 - final_regr: 0.0355
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.951
```

```
Classifier accuracy for bounding boxes from RPN: 0.92475
Loss RPN classifier: 0.08591906025962552
Loss RPN regression: 0.02556355657428503
Loss Detector classifier: 0.19671864169563197
Loss Detector regression: 0.03521551633630588
Total loss: 0.3434167748658484
Elapsed time: 6631.824587583542
Epoch 80/60
1000/1000 [==============================] - 7689s 8s/step - rpn_cls: 0.0900 - rpn_regr: 0.0246 - fi
nal_cls: 0.2005 - final_regr: 0.0315
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.367
Classifier accuracy for bounding boxes from RPN: 0.92475
Loss RPN classifier: 0.09132514196468469
Loss RPN regression: 0.024924593943636865
Loss Detector classifier: 0.20961455817628302
Loss Detector regression: 0.03223358319763793
Total loss: 0.3580978772822425
Elapsed time: 7689.335340499878
Epoch 81/60
1000/1000 [==============================] - 7662s 8s/step - rpn_cls: 0.0819 - rpn_regr: 0.0250 - fi
nal_cls: 0.1970 - final_regr: 0.0389
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.415
Classifier accuracy for bounding boxes from RPN: 0.921
Loss RPN classifier: 0.08188241765210545
Loss RPN regression: 0.02523372013051994
Loss Detector classifier: 0.20460195709143592
Loss Detector regression: 0.036888282925378005
Total loss: 0.3486063777994393
Elapsed time: 7661.690681219101
Epoch 82/60
1000/1000 [==============================] - 7694s 8s/step - rpn_cls: 0.0849 - rpn_regr: 0.0253 - fi
nal_cls: 0.2007 - final_regr: 0.0359
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.496
Classifier accuracy for bounding boxes from RPN: 0.929
Loss RPN classifier: 0.08424107030714119
Loss RPN regression: 0.025151378917973487
Loss Detector classifier: 0.19834954072417169
Loss Detector regression: 0.03698524066535174
Total loss: 0.3447272306146381
Elapsed time: 7693.543663978577
Epoch 83/60
1000/1000 [==============================] - 7873s 8s/step - rpn_cls: 0.0765 - rpn_regr: 0.0254 - fi
nal_cls: 0.2056 - final_regr: 0.0337
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.868
Classifier accuracy for bounding boxes from RPN: 0.92
Loss RPN classifier: 0.08310100098651435
Loss RPN regression: 0.025274386775679888
Loss Detector classifier: 0.20901364847868537
Loss Detector regression: 0.03381574360148806
Total loss: 0.35120477984236764
Elapsed time: 7873.088410615921
Epoch 84/60
1000/1000 [==============================] - 7000s 7s/step - rpn_cls: 0.0927 - rpn_regr: 0.0255 - fi
nal_cls: 0.2168 - final_regr: 0.0369
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.389
Classifier accuracy for bounding boxes from RPN: 0.91825
Loss RPN classifier: 0.08458953772819183
Loss RPN regression: 0.025137307644821704
Loss Detector classifier: 0.21442992895807403
Loss Detector regression: 0.03596716099162586
Total loss: 0.3601239353227135
Elapsed time: 7000.206551790237
Epoch 85/60
1000/1000 [==============================] - 7406s 7s/step - rpn_cls: 0.0802 - rpn_regr: 0.0250 - fi
nal_cls: 0.1920 - final_regr: 0.0318
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.328
Classifier accuracy for bounding boxes from RPN: 0.92875
Loss RPN classifier: 0.0797471123497962
Loss RPN regression: 0.02484606988960877
Loss Detector classifier: 0.2014502203487973
Loss Detector regression: 0.03484744988333114
Total loss: 0.34089085247153345
Elapsed time: 7406.070822477341
Total loss decreased from 0.34100244315510664 to 0.34089085247153345, saving weights
Epoch 86/60
1000/1000 [==============================] - 8324s 8s/step - rpn_cls: 0.0808 - rpn_regr: 0.0244 - fi
nal_cls: 0.1874 - final_regr: 0.0295
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.74
Classifier accuracy for bounding boxes from RPN: 0.92775
Loss RPN classifier: 0.08624852863784978
Loss RPN regression: 0.024741748275468125
Loss Detector classifier: 0.1935795916817733
Loss Detector regression: 0.030097318743442885
```

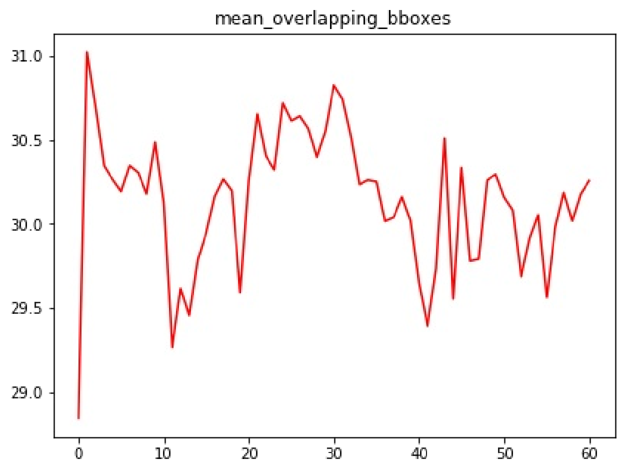
```
Total loss: 0.3346671873385341
Elapsed time: 8331.613528251648
Total loss decreased from 0.34089085247153345 to 0.3346671873385341, saving weights
Epoch 87/60
1000/1000 [==============================] - 8771s 9s/step - rpn_cls: 0.0830 - rpn_regr: 0.0245 - fi
nal_cls: 0.1998 - final_regr: 0.0340
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.406
Classifier accuracy for bounding boxes from RPN: 0.9225
Loss RPN classifier: 0.08458911336152294
Loss RPN regression: 0.02479158409126103
Loss Detector classifier: 0.19903594863948093
Loss Detector regression: 0.03394268846153864
Total loss: 0.3423593345538036
Elapsed time: 8777.779419660568
Epoch 88/60
1000/1000 [==============================] - 7110s 7s/step - rpn_cls: 0.0862 - rpn_regr: 0.0243 - fi
nal_cls: 0.2012 - final_regr: 0.0293
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.386
Classifier accuracy for bounding boxes from RPN: 0.91875
Loss RPN classifier: 0.0845082330193399
Loss RPN regression: 0.024557393630966543
Loss Detector classifier: 0.21334326624153802
Loss Detector regression: 0.030743608420249074
Total loss: 0.3531525013120936
Elapsed time: 7110.43545627594
Epoch 89/60
1000/1000 [==============================] - 6640s 7s/step - rpn_cls: 0.0824 - rpn_regr: 0.0241 - fi
nal_cls: 0.1857 - final_regr: 0.0391
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.535
Classifier accuracy for bounding boxes from RPN: 0.9285
Loss RPN classifier: 0.08567835044613271
Loss RPN regression: 0.02433119602734223
Loss Detector classifier: 0.19577110953834198
Loss Detector regression: 0.037844057447218804
Total loss: 0.3436247134590357
Elapsed time: 6639.919267177582
Epoch 90/60
1000/1000 [==============================] - 6667s 7s/step - rpn_cls: 0.0886 - rpn_regr: 0.0243 - fi
nal_cls: 0.2093 - final_regr: 0.0303
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.508
Classifier accuracy for bounding boxes from RPN: 0.92075
Loss RPN classifier: 0.08628362230237988
Loss RPN regression: 0.024162452936172484
Loss Detector classifier: 0.20400028790373106
Loss Detector regression: 0.029091070202965058
Total loss: 0.34353743334524844
Elapsed time: 6666.587341308594
Epoch 91/60
1000/1000 [==============================] - 6627s 7s/step - rpn_cls: 0.0819 - rpn_regr: 0.0246 - fi
nal_cls: 0.2259 - final_regr: 0.0392
Mean number of bounding boxes from RPN overlapping ground truth boxes: 30.031
Classifier accuracy for bounding boxes from RPN: 0.921
Loss RPN classifier: 0.08407078277728965
Loss RPN regression: 0.024675528082996608
Loss Detector classifier: 0.21517194391562497
Loss Detector regression: 0.03587911587485723
Total loss: 0.35979737065076844
Elapsed time: 6626.699509382248
Epoch 92/60
1000/1000 [==============================] - 6617s 7s/step - rpn_cls: 0.0897 - rpn_regr: 0.0240 - fi
nal_cls: 0.1904 - final_regr: 0.0322
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.627
Classifier accuracy for bounding boxes from RPN: 0.934
Loss RPN classifier: 0.08543467678808205
Loss RPN regression: 0.02398829758935608
Loss Detector classifier: 0.1817812535882481
Loss Detector regression: 0.03701828640169697
Total loss: 0.3282225143673832
Elapsed time: 6617.477471828461
Total loss decreased from 0.3346671873385341 to 0.3282225143673832, saving weights
Epoch 93/60
1000/1000 [==============================] - 7190s 7s/step - rpn_cls: 0.0865 - rpn_regr: 0.0245 - fi
nal_cls: 0.1970 - final_regr: 0.0290
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.85
Classifier accuracy for bounding boxes from RPN: 0.931
Loss RPN classifier: 0.08796051931356712
Loss RPN regression: 0.02445066845556721
Loss Detector classifier: 0.18809619675452996
Loss Detector regression: 0.03208878068238846
Total loss: 0.3325961652060528
Elapsed time: 7196.86708855629
Epoch 94/60
1000/1000 [==============================] - 7768s 8s/step - rpn_cls: 0.0825 - rpn_regr: 0.0242 - fi
```

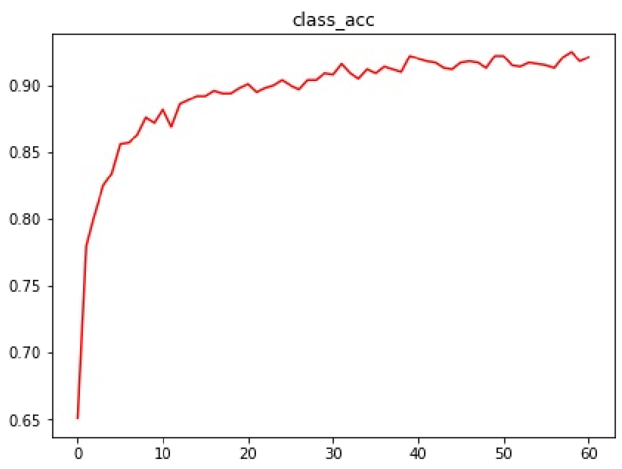
```
nal_cls: 0.1874 - final_regr: 0.0339
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.584
Classifier accuracy for bounding boxes from RPN: 0.9245
Loss RPN classifier: 0.08458400031384367
Loss RPN regression: 0.024367157872999087
Loss Detector classifier: 0.1874653723888805
Loss Detector regression: 0.0338291128263736
Total loss: 0.33024564340209683
Elapsed time: 7767.974817991257
Epoch 95/60
1000/1000 [==============================] - 7073s 7s/step - rpn_cls: 0.0873 - rpn_regr: 0.0242 - fi
nal_cls: 0.1924 - final_regr: 0.0323
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.52
Classifier accuracy for bounding boxes from RPN: 0.9275
Loss RPN classifier: 0.08959633706115516
Loss RPN regression: 0.02446211666613817
Loss Detector classifier: 0.1901608225685486
Loss Detector regression: 0.03159658499984653
Total loss: 0.33581586129568847
Elapsed time: 7072.525515556335
Epoch 96/60
1000/1000 [==============================] - 7833s 8s/step - rpn_cls: 0.0850 - rpn_regr: 0.0237 - fi
nal_cls: 0.2028 - final_regr: 0.0358
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.355
Classifier accuracy for bounding boxes from RPN: 0.9285
Loss RPN classifier: 0.08474577624000222
Loss RPN regression: 0.023984080183552577
Loss Detector classifier: 0.2015761950266551
Loss Detector regression: 0.036200154920647036
Total loss: 0.34650620637085694
Elapsed time: 7834.14253950119
Epoch 97/60
1000/1000 [==============================] - 7627s 8s/step - rpn_cls: 0.0831 - rpn_regr: 0.0242 - fi
nal_cls: 0.2013 - final_regr: 0.0319
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.914
Classifier accuracy for bounding boxes from RPN: 0.9225
Loss RPN classifier: 0.08324233218219376
Loss RPN regression: 0.02398933415627107
Loss Detector classifier: 0.20810372451122333
Loss Detector regression: 0.03180024007341126
Total loss: 0.3471356309230994
Elapsed time: 7628.251050710678
Epoch 98/60
1000/1000 [==============================] - 8377s 8s/step - rpn_cls: 0.0836 - rpn_regr: 0.0236 - fi
nal_cls: 0.1997 - final_regr: 0.0381
Mean number of bounding boxes from RPN overlapping ground truth boxes: 29.628
Classifier accuracy for bounding boxes from RPN: 0.93
Loss RPN classifier: 0.08448654507957898
Loss RPN regression: 0.023630094278603794
Loss Detector classifier: 0.1940618000472532
Loss Detector regression: 0.03760375516103522
Total loss: 0.3397821945664712
Elapsed time: 8377.038774967194
Epoch 99/60
 376/1000 [==========>...................] - ETA: 1:26:45 - rpn_cls: 0.0899 - rpn_regr: 0.0245 - fin
al_cls: 0.2822 - final_regr: 0.0475
```

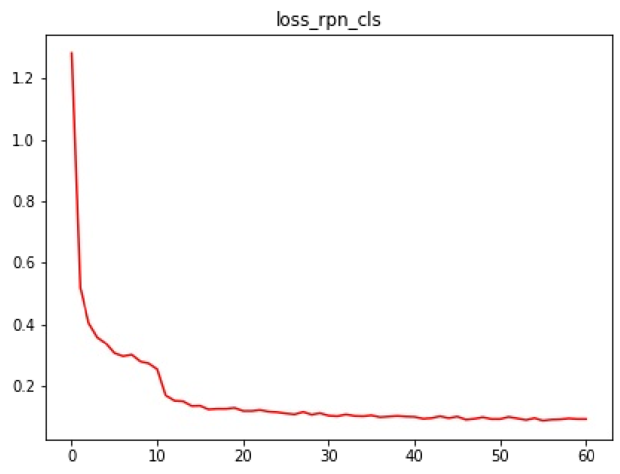
```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-38-493fa9600e77> in <module>
     21 
     22             # Train rpn model and get loss value [_, loss_rpn_cls, loss_rpn_regr]
---> 23             loss_rpn = model_rpn.train_on_batch(X, Y)
     24 
     25             # Get predicted rpn from rpn model [rpn_cls, rpn_regr]

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\keras\engine\training.py in
train_on_batch(self, x, y, sample_weight, class_weight)
   1215             ins = x + y + sample_weights
   1216         self._make_train_function()
-> 1217         outputs = self.train_function(ins)
   1218         return unpack_singleton(outputs)
   1219 

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\keras\backend\tensorflow_bac
kend.py in __call__(self, inputs)
   2713                 return self._legacy_call(inputs)
   2714 
-> 2715             return self._call(inputs)
   2716         else:
   2717             if py_any(is_tensor(x) for x in inputs):

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\keras\backend\tensorflow_bac
kend.py in _call(self, inputs)
   2673             fetched = self._callable_fn(*array_vals, run_metadata=self.run_metadata)
   2674         else:
-> 2675             fetched = self._callable_fn(*array_vals)
   2676         return fetched[:len(self.outputs)]
   2677 

c:\users\srila\appdata\local\programs\python\python36\lib\site-packages\tensorflow\python\client\ses
sion.py in __call__(self, *args, **kwargs)
   1437           ret = tf_session.TF_SessionRunCallable(
   1438               self._session._session, self._handle, args, status,
-> 1439               run_metadata_ptr)
   1440         if run_metadata:
   1441           proto_data = tf_session.TF_GetBuffer(run_metadata_ptr)

KeyboardInterrupt: 
```

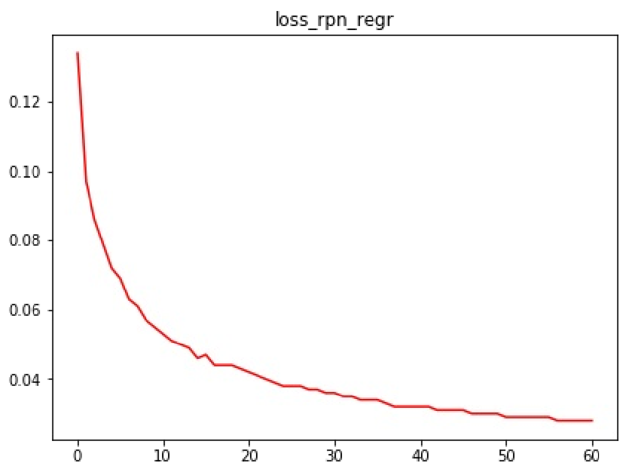
In [32]:

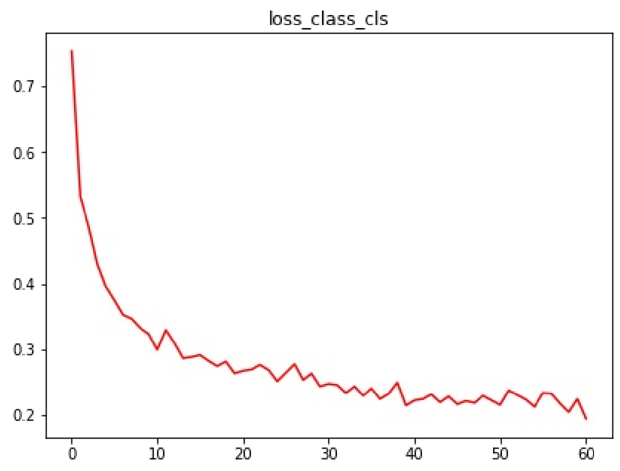
```
r_epochs=61
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.arange(0, r_epochs), record_df['mean_overlapping_bboxes'], 'r')
plt.title('mean_overlapping_bboxes')
plt.subplot(1,2,2)
plt.plot(np.arange(0, r_epochs), record_df['class_acc'], 'r')
plt.title('class_acc')

plt.show()

plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.arange(0, r_epochs), record_df['loss_rpn_cls'], 'r')
plt.title('loss_rpn_cls')
plt.subplot(1,2,2)
plt.plot(np.arange(0, r_epochs), record_df['loss_rpn_regr'], 'r')
plt.title('loss_rpn_regr')
plt.show()


plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.arange(0, r_epochs), record_df['loss_class_cls'], 'r')
plt.title('loss_class_cls')
plt.subplot(1,2,2)
plt.plot(np.arange(0, r_epochs), record_df['loss_class_regr'], 'r')
plt.title('loss_class_regr')
plt.show()

plt.plot(np.arange(0, r_epochs), record_df['curr_loss'], 'r')
plt.title('total_loss')
plt.show()

# plt.figure(figsize=(15,5))
# plt.subplot(1,2,1)
# plt.plot(np.arange(0, r_epochs), record_df['curr_loss'], 'r')
# plt.title('total_loss')
# plt.subplot(1,2,2)
# plt.plot(np.arange(0, r_epochs), record_df['elapsed_time'], 'r')
# plt.title('elapsed_time')
# plt.show()

# plt.title('loss')
# plt.plot(np.arange(0, r_epochs), record_df['loss_rpn_cls'], 'b')
# plt.plot(np.arange(0, r_epochs), record_df['loss_rpn_regr'], 'g')
# plt.plot(np.arange(0, r_epochs), record_df['loss_class_cls'], 'r')
# plt.plot(np.arange(0, r_epochs), record_df['loss_class_regr'], 'c')
# # plt.plot(np.arange(0, r_epochs), record_df['curr_loss'], 'm')
# plt.show()
```

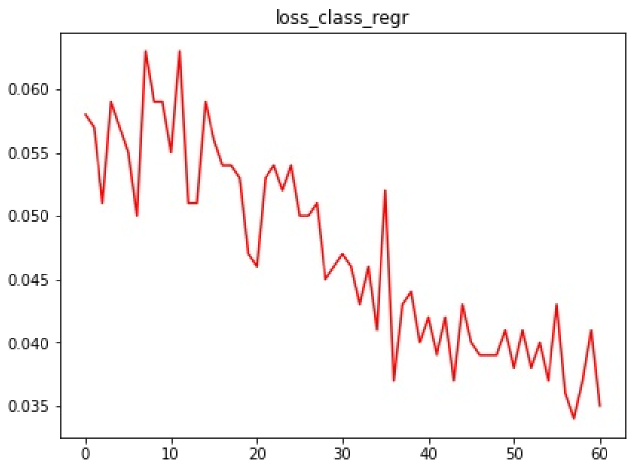


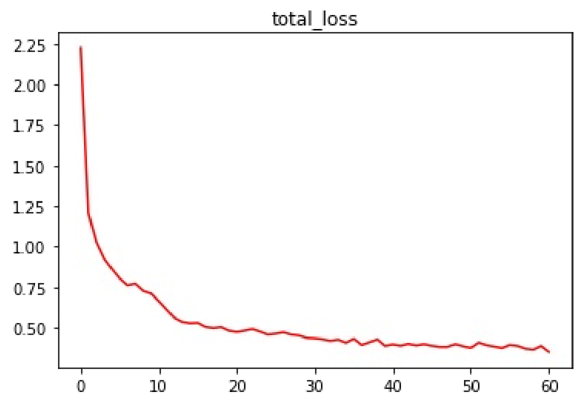



loss_rpn_cls
1.2
1.0
0.8
0.6
0.4
0.2
0
10
20
30
40
50
60


loss_rpn_regr
0.12
0.10
0.08
0.06
0.04
0
10
20
30
40
50
60


loss_class_cls
0.7
0.6
0.5
0.4
0.3
0.2
0
10
20
30
40
50
60


loss_class_regr
0.060
0.055
0.050
0.045
0.040
0.035
0
10
20
30
40
50
60


total_loss
2.25
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0
10
20
30
40
50
60

SUMMARY

Faster R-CNN is an object detection algorithm. Given an input images which has several objects, using Faster R-CNN different objects in the input image are identified and given a class label along with the probability of identifying the image. As part of this case study - Object Detection on Malaria images, the task is to identify and classify the cells in the given image into red blood cells,trophozoite,difficult,ring,schizont,gametocyte,leukocyte. At a higher level, steps in Faster R-CNN are,

1. Take an input image and pass it to convnet like VGG16, ResNet50, Xception, Densenet which can deeply learn the image to extract feature maps.
2. Anchors (bounding boxes) are placed across the entire image with different dimensions and aspect ratio.
3. These anchors are an input to the Region Proposal Network which is a fully convolutional layer, where it outputs a good set of proposals which has objects. a. An objectness score which says if the anchor consists an object and filter out anchors which does not have objects. b. Bounding box regression coefficients to better adjust and fit the anchors to the object.
4. We will get a bunch of output proposals from RPN layer without any class labels assigned to them. ROI pooling layer will reconstruct the extracted proposals as fixed size images to feed them to R-CNN layer to classify and label the image.
5. Features extracted from ROI pooling will be used to classify and assign a probability score for the classification label in Region Based-CNN which is the final layer.
6. There will be 4 losses 2 for RPN layer and 2 for R-CNN layer.
7. By seeing the above graphs, loss has been decreased from 2.25 to 0.34 and accuracy has been increased from 65% to 93%.
8. Mean Overlapping bounding boxes provides the mean number of bounding boxes which overlap with the actual objects.

Faster R-CNN has a complex model and the code for different classes used is written by https://github.com/yhenon (https://github.com/yhenon). I used the code provided and used several base models to train on the provided images to see which base model gives best accuracy with preferabally low time. The base models tried are:

1. VGG16 - Accuracy achieved is 93% with a total loss of 0.32. Each epoch took about 2 hours. Trained the model for 62 epochs.
2. Xception, Densenet, Resnet50 - Each epoch was taking 5-6 hours to train. The accuracy was low at 50%.

In [ ]:

```

```