Automatic Detection of Photographing or Filming

Zhiyuan Chen

BSc: Creative Technology

July 2020

Faculty of Electrical Engineering, Mathematics and Computer Science

University of Twente

Supervisor: Raymond Veldhuis

Critical Observer: Mannes Poel

Abstract

A wide use of cameras and portable devices with built-in cameras such as smartphones and social media makes taking photos and sharing very easy, while unintentional capture of facial image brings privacy concerns. To detect photographing or filming, a detection model using the state of the art architecture of Faster R-CNN is designed to detect camera objects on still images, which lays a foundation of the work of real-time detection of photographing or filming. Choices such as selection of feature extractors, dataset selection and number of region proposals are made on different models. Each model is tested and evaluated concerning performance, false negative rate and bounding boxes etc.

Table of Contents

Chapter 1 – Introduction

1.1 Problem description

Camera has become strongly needed in human daily lives. It is used to record special and meaningful moments. Nowadays actions of taking selfies, photos of others and photos of sceneries can be frequently seen in public places. However, unintentional capture of facial image can be a big issue of privacy due to the popularity of portable devices with built-in cameras and advances in social networking services [1]. Because cameras are widely used, the chance of being photographed is increasingly high. On the other hand, Social media increases the efficiency of communication and online sharing can cause unwanted photos spreading broadly in no time.

1.2 Project background

This topic of this research is one of three parts of a big project. The other two parts are inhibition of photographing or filming and wearable design. The big project is aimed for a wearable that can be worn on a person, which can perform automatic detection on photographing and filming in sight from the perspective of the user and prevent the facial image of the user being captured.

This big project is provided by DMB (Data Management & Biometrics) research group of University of Twente. It is a new project based on the concept of protecting the privacy of facial image from photographing or filming. There is some prior research conducted about the inhibition part of the big project, that is blocking the capture of unwanted images. The camera system and light sources are analyzed as well. However, the detection part is not discussed in prior work.

Photographing or filming is a human-object activity of a person using a camera, where hand gestures, actions and object positioning are involved. Therefore, photographing or filming has its own characteristics of human object interactions, but it's not easy to detect complex interactions. Considering the study goal of a bachelor program and feasibility, this research will focus on the object detection on cameras to build the foundation of the big project, At the same time errors may also occur if photographing or filming is predicted by camera object detection only. For example, a camera can be carried in hand or hanging in front of the chest of someone. Hence much space can be explored for further research.

1.2.1 Research question

Cameras have been developed over a long history. Various types of camera can be commonly seen in daily life such as digital cameras, phone camera and movie camera. There are variations in technologies in different types of cameras. For example, when it comes to auto focusing, some cameras with assist lamps can project visible or IR signals to achieve this, while phone cameras usually use image processing algorithms to achieve autofocusing. Therefore, sensor-based detection on camera objects may have many limitations.

To have a general approach on detection of different types of cameras, a vision-based detection method, object detection, will be explored in this research. Object detection in computer vision has been a challenge until today. It belongs to the field of machine learning and deep learning, in which many applications arise such as face recognition and action recognition. The main goal of this research is to build custom camera object detection models and train the detector in order to detect camera objects in natural scenes. Thus, the following questions need to be answered:

Main question:
How to detect different types of handheld camera objects in daily scenes from still images using existing architectures of detection models?
Sub questions:
How many cameras can be detected?
How to detect portable devices with built-in cameras such as smartphones?
How to precisely localize the camera objects?
How well can the cameras be detected?
What are the challenges of accurate detection?

1.3 Summary

This research consists of four parts. The first part is the background of this project and the state-of-the-art review related to object detection. The second part is ideation and specification. This part covers the designing process of the object detection model and choices made. The realization and evaluation are in the third part. Here, several detection models are tested and evaluated. Then, an ethical preview of the detection system is in the next part. The last part discusses the future work for improvement of the detection on the final purpose of this project.

Chapter 2 – State of the art

The main task of this project is to perform an object detection on the camera object. To understand the vision-based recognition techniques, a state-of-the-art review is conducted on data collection, object detection model and evaluation metrics.

2.1 Data collection

2.1.1 ImageNet Dataset

ImageNet [2] is a large-scale database that consists of tens of millions of annotated images organized by the semantic hierarchy of WordNet. Up to the publication of this paper, there are 12 subtrees with 5247 synsets and 3.2 million images in total in the database. This means it can provide a comprehensive coverage of objects in life. The data collection scheme is described with Amazon Mechanical Turk, which is used to label vision data. ImageNet is intended to offer opportunities to researchers in the computer vision community and beyond. It is also useful in applications such as object recognition, image classification and automatic object clustering.

There are 9 synsets of photographic cameras [3] included in this dataset, which are flash camera, candid camera, motion-picture camera (movie camera, cine-camera), point-and-shoot camera, portrait camera, Polaroid camera (Polaroid Land camera), reflex camera, digital camera and box camera (box Kodak).

2.1.2 Open Images Dataset

Open Images Dataset [4] is a dataset of 9.2M images with unified annotations for image classification. It contains nearly 600 object classes together with image-level labels and bounding boxes. There are also visual relationship annotations involving 57 classes. The images often show complex scenes with several objects. It is advanced in scale, quality of annotations and variety. It is also aimed on research and innovation purposes on image classification, object detection, and visual relationship detection and beyond.

Camera is one of the categories of this dataset. There are a large amount of image resources including cameras with bounding boxes or segmentation.

2.1.3 Custom dataset

There are two necessary steps for building a custom dataset. The first step is to gather image data from the internet. J. Deng et al. [2] collect candidate images from the Internet by querying several image search engines. And for each synset, the queries are the set of WordNet synonyms in order to limit the images retrievable. The queries are also expanded with the word from parent synsets or translated in other languages to further enlarge and diversify the candidate pool. In another paper, A. Kuznetsova et al. [4] explain their image acquisition as starting from Flickr to collect images with CC-BY license and removing images with inappropriate content, near-duplicate images and images that appear elsewhere on the internet. Then, the set of classes included in the dataset are derived from JFT, an internal dataset at Google. Both papers show that for the selection of images in the dataset, it is necessary to collect image data from the internet and do classification.

The other step is labelling. J. Deng et al. [2] use the service of Amazon Mechanical Turk (AMT), a platform on which users can get paid by completing posted tasks. This means the labelling is done by global users of AMT manually, which ensures the accuracy of labelling. On the other hand, A. Kuznetsova et al. [4] point out that manually labelling large numbers of images with the presence or absence of 19,794 different classes is not feasible considering the large database to build. Therefore, they use a google-internal variant of the InceptionV2-based image classifier, which is publicly available through the Google Cloud Vision API, to generate predictions about labels. Afterwards, they use human verification of candidate labels. Although two papers show different main methods for labelling, human verification is important. Therefore, for self-developed datasets with a small number of classes, manual labelling can be promising.

Besides, data augmentation can be an additional step when it comes to limited images. C . Shorten and T. M. Khoshgoftaar [5] suggests a data-space solution to the problem of limited data. In this survey there are various image augmentation algorithms discussed, including geometric transformations, color space augmentations, kernel filters, mixing images, random erasing, feature space augmentation etc. Data augmentation is aimed at expanding limited datasets and improving the performance of the deep learning models.

2.2 Object detection model

2.2.1 R-CNN model Family

R-CNN is abbreviated from Region-based Convolutional Neural Networks. It was introduced in the paper of R. Girshick et al. [6] in 2014, of which the architecture can be seen in fig 1. Their R-CNN model consists of 3 modules. The first module generates category-independent region proposals such as bounding boxes. The second one is a large convolutional neural network that extracts a feature vector from each region. The third one is a set of linear SVM (support-vector machine) classifiers that can classify the features. This R-CNN model is measured on VOC 2012 database and achieves a good result of performance. It achieves a mAP of 53.3% on VOC 2012 with more than 30% relative to the previous best results. This shows that R-CNN is a milestone in object detection. A downside is that the feature extraction has to pass on each of the candidate regions generated by the region proposals. This leaves space for improvement in processing time.

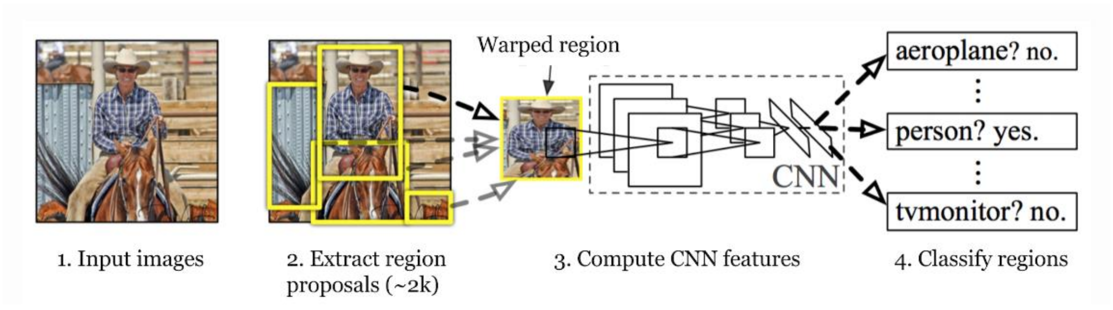


Fig 1. The architecture of R-CNN [6]

R-CNN was further improved and developed into Fast R-CNN (Fast Region-based Convolutional Network) in 2015. R. Girshick [7] proposed a Fast R-CNN architecture that takes as input an entire image and a set of object proposals as shown in Fig 2. The network first processes the whole image with several layers to produce a convolutional feature map. Then, for each object proposal a RoI (region of interest) pooling layer extracts a fixed-length feature vector from the feature map. Each feature vector is fed into a sequence of connected layers. This results in two output layers: one produces a class prediction through softmax probability estimates and the other outputs four real-valued numbers for each bounding box position. Fast R-CNN model trains the very deep VGG16 network 9× faster than R-CNN and achieves a higher mAP, which is 68.4%, on PASCAL VOC 2012. The Python and C++ source code was made available online.

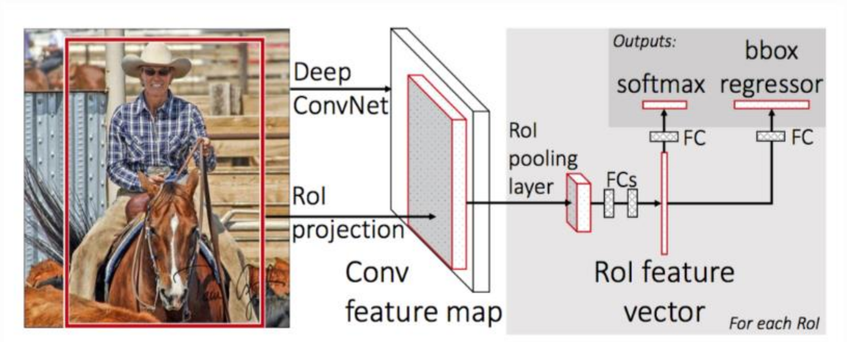


Fig 2. The architecture of Fast R-CNN [7]

Fast R-CNN was further improved into Faster R-CNN for detection speed and detection accuracy by S. Ren et al. [8] in 2016, who introduce a Region Proposal Network (RPN) that shares full-image convolutional features with the detection network, in which there is a change in algorithm of computing proposals with a deep CNN , thus enabling nearly cost-free region proposals. RPN is merged together with Fast R-CNN into a single network design. This improves the detection system by a frame rate of 5fps (including all steps) on a GPU, while achieving state-of-the-art object detection accuracy on PASCAL VOC 2007, 2012, and MS COCO datasets with 300 proposals per image. It achieves 75.9% mAP on the union set of PASCAL VOC 2007, 2012, and MS COCO datasets with RPN and VGG16 pre-trained network. The architecture of it can be seen in Fig 3. The source code in Python and C++ is also available online.

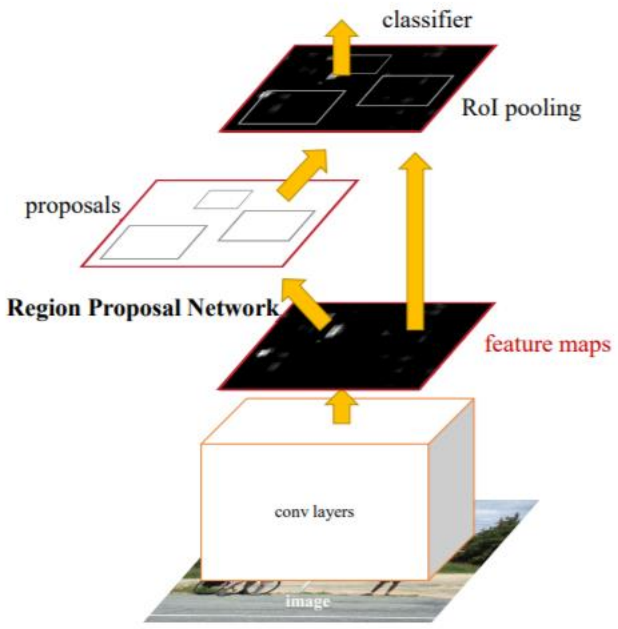


Fig 3. An illustration of Faster R-CNN model [8]

In addition, Mask R-CNN proposed by K. He et al. [9] extends Faster R-CNN by adding a branch for predicting an object mask in parallel with the existing branch for bounding box recognition. This supports Faster R-CNN for object instance segmentation. Mask R-CNN is also easy to generalize to other tasks such as allowing us to estimate human poses in the same framework. The simple and effective approach provides opportunities for future research in instance-level recognition.

2.2.2 YOLO model family

YOLO (You Only Look Once) method was first presented in 2015 and revised in 2016 by J. Redmon et al. [10]. It frames object detection as a regression problem. It uses a single neural network that predicts bounding boxes and classes from the full image in one evaluation. The architecture can be seen in Fig 4. The model first divides an image into a grid of cells. Each grid cell predicts bounding boxes and confidence scores for those boxes. Each bounding box contains the prediction of relation to the bounds of the grid cell and confidence. Each grid cell also predicts class labels. The unified architecture is extremely fast. The YOLO model can process images in real-time at 45 frames per second. A small version of the network, Fast YOLO, can even process 155 frames per second. This shows the advantage of detection speed of YOLO compared to other detectors, while it makes more localization errors. On the VOC 2012 test set, YOLO scores 57.9% mAP.

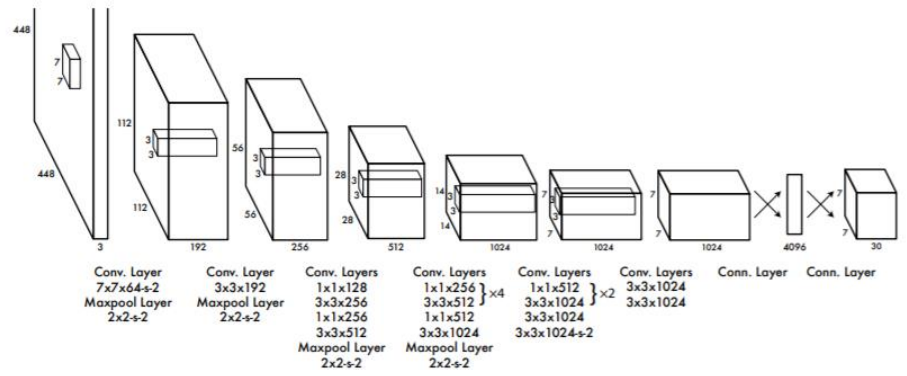


Fig 4. The architecture of YOLO [10]

YOLOv2 (YOLO9000) was an updated version of YOLO that was proposed by J. Redmon and A. Farhadi [11] in 2016. The performance of the detection model is further improved. Batch normalization is added on all the convolutional layers in YOLO, which leads to more improvement in performance and helps regularize the model. YOLOv2 model, with high resolution classifier, can run at varying sizes, offering an easy tradeoff between speed and accuracy. It uses anchor boxes to predict coordinates of bounding boxes, but unlike Fast R-CNN, it uses only convolutional layers to predict offsets and confidences for anchor boxes. It also runs k-means clustering on the training set bounding boxes to automatically find good priors for the network. For location prediction, YOLOv2 predicts location coordinates relative to the location of the grid cell. This makes the network more stable. YOLOv2 is better, faster and stronger and achieved good results in VOC 2007. At 67 FPS, YOLOv2 gets 76.8% mAP on VOC 2007. At 40 FPS, YOLOv2 gets 78.6% mAP.

YOLOv3 was added further improvements and proposed by J. Redmon and A. Farhadi [12] in 2018. There are some minor improvements from YOLOv2. A new network of feature extractor is used. It has an improvement in inference time, with 57.9% mAP in 51 ms.

2.2.3 Conclusion

Both R-CNN and YOLO approaches are state of the art detection method in object detection. R-CNN approach has more accurate detection result than YOLO approach. However, YOLO approach shows outstanding advantage in detection speed. There are some common network components design. The updating history through years also shows the learning and improvement process. Both methods give insights to further research in the field of computer vision and applications.

2.3 Evaluation metrics

2.3.1 Intersection over Union (IoU)

IoU computes intersection over the union of the two bounding boxes; the bounding box for the ground truth and the predicted bounding box. It implies how the prediction fits the ground truth [13]. A typical threshold value is 0.5. If IoU is larger than 0.5, the object detection is classified as True Positive (TP). If IoU is smaller than 0.5, it is False Positive (FP). When a ground truth is present in the image and model fails to detect the object. It will be classified as False Negative (FN). True Negative (TN) means no prediction on an object, which is not useful in object detection.

2.3.2 Average Precision (AP),

Average Precision (AP) is the most commonly used metric, derived from precision and recall are the metrics to evaluate the performance of object detection [13]. AP is usually evaluated on specific object categories. This means that it is computed for each object category separately.

2.3.3 mean AP (mAP)

Mean AP (mAP) is adopted to compare performance over all object categories and result in an average performance score as the final measure of performance. According to M. Everingham et al., the AP summarizes the shape of the precision/recall curve and is defined as thx`e mean precision at a set of eleven equally spaced recall 11 levels. O. Russakovsky et al. [3] also states the metric of mAP to measure object detection accuracy.

Chapter 3 – Ideation

Unlike other graduation projects that are aimed for a complete system or a product, the goal of this project is to build the detection system of a big project only. Thus, the ideation phase of the design process of Creative Technology, including user interview, storyboard and user needs analysis, does not fit this project because a sub system is far from users. This chapter will give an overview of the design process of the camera object detector itself. This chapter also works as a summary of the realization part.

3.1 Selection of object detection model

In Chapter 2, 2 popular object detection models YOLO family and R-CNN family are discussed. YOLO has an advantage of detection speed in real-time detection, while R-CNN can achieve high accuracy of detection without considering the speed. As this project is focusing on the detection on still images, R-CNN is suitable for it. Thus, this project features object detection using Faster R-CNN architecture, which is the state-of-the-art architecture within the R-CNN family. Since detection is the only purpose of the project, there is no need of using Mask R-

CNN, which allows object segmentation.

3.2 Selection of Dataset

Chapter 2 also covers a few candidates of the dataset for training. ImageNet and Open Image Dataset both have images of camera objects, with ImageNet having 1325 images [3] and Open Image Dataset having 5037 images with bounding boxes [4]. Therefore, it is obvious that Open Image Dataset can provide more training data as input to the camera detector. Additionally, Open Image Dataset has images with a large variety of categories of objects, including “Mobile Phone” category, which is of good use in detection specification.

3.3 Training data preprocessing

3.3.1 Downloading subset

The first thing to do is downloading a subset of the entire dataset because this project is only aimed for specific camera detection instead of detection on many objects. Also, the entire dataset is too large to download. Thus, it is necessary to download a subset of it. On the site of Open Image Dataset, there are three .csv files that are important for this task. The first one contains all the image names and their URLs. The second one contains image IDs, label names and annotations. The third one contains label names and class names. Thus, the first step is to find the label name of class of interest in the third file, which is ‘’Camera’’. Then, by matching the label name with that in the annotation file, images of other classes are filtered out and those of ‘’Camera’’ class remains. The next step is to match the image IDs in the second file with those in the first file and download them.

3.3.2 Number of images for training

Here the number of downloading images is set to 1000 and images are randomly chosen from all. The number is not too small nor too large. With this number, there might be adequate resources for the detector to learn and it will not take too long for training. Besides, the number is only an estimate of how much is needed at the very beginning. There is also space for adjustment after some experiments.

3.3.3 Training format

After downloading, all camera images are saved in one folder. Then, all the images are randomly divided into 2 sets, train set and test set, with the ratio of 8: 2. After this, all the images will be parsed to find the coordinates of the bounding boxes and these are made into .csv files together with image names. In the end, the .csv files will be written into .txt files with the file path. Here the file path will be the path in the google drive when using the Colab notebook.

3.4 Training environment

Due to the limitation on the access to labs and the lack of powerful GPU in home environment, Colab notebook is used to speed up the training process. Colab notebook is a hosted Jupyter notebook service that allows Python code writing and it is suitable for machine learning, data science and research. It has free access to Tesla K80 GPU and abundant library resources, so no prior configuration is needed. Like Jupyter notebook, it has a similar interactive interface, which allows live code and data visualization.

3.5 Training setting

3.5.1 Data augmentation

To make the best of resources and to ensure higher accuracy in detection, data augmentation is used in this project. It also helps to avoid overfitting. In this project, horizontal flipping, vertical flipping and 90-degree rotation are used for data augmentation. The data augmentation is only used in the process of training and not in testing. With data augmentation methods listed, the dataset is supposed to be three times bigger.

3.5.2 Batch size

Batch size is a term used in machine learning and it represents the number of training samples in one iteration. There are batch mode, mini-batch mode and stochastic mode respectively. Batch mode means the batch size is equal to the total number of training samples so that the iteration and epoch values become equivalent. In mini-batch mode, the batch size is larger than one but smaller than the total number of training samples. In stochastic mode the batch size is one.

With data augmentation, it is possible to have nearly 3200 images for training, with the 800 original training samples. As explained by Z. Yao et al. [14], large batch training has a number of well-known drawbacks, including degradation of accuracy, poor generalization, wasted computation, and even poor robustness to adversarial perturbations. However, large batch size can decrease the computational time proportional to the increase in batch size. This means that a batch size of 16 will take less computation than twice of the amount of a batch of 8. Thus, a big size can provide a stable enough estimate of the gradient of the dataset. In this project the number of batch size is set to 1000 from the beginning to start with.

3.5.3 Learning rate

Learning rate or step size is referred to as the amount that weights are updated during training [15]. It is a configurable parameter used in the training of neural networks. In this project Adam optimizer is used with the learning rate set to 0.0001, which is also chosen in the official Faster R-CNN paper on PASCAL VOC dataset. However, a learning rate of 0.001 is also used in the paper. In the process of designing the detection system, learning rate is not taken seriously as it might not affect the result significantly.

3.5.4 Input image resize

According to J. Huang et al., reducing input image size by half in both dimensions can lower the accuracy consistently (by 15.88% on average) but also reduces inference time by a relative factor of 27.4% on average. Subsequently, input image resolution has a great impact on both accuracy and training time. In this project, resizing images is involved to have faster training. The width and height of images both are reduced by half. Faster training gives a good overview of how well the result can be and leaves the space for improvement.

3.6 Matching strategy and criteria

3.6.1 IoU threshold

IoU is the ratio of the intersecting area to the union area of two bounding boxes, as shown in Fig 5. The value ranges from 0 to 1, with 0 meaning there is no overlapping areas between two bounding boxes and 1 meaning two bounding boxes are equal. IoU can be used in ground truth matching, non max suppression and mAP calculation.
By comparing the predicted bounding box with the ground truth box in the stage of classification, IoU helps to decide whether the prediction is "foreground" or "background", where "foreground" means a detected class. Additionally, an IoU threshold is involved in this process. For example, it is set to 0.5, 0.75 or between 0.5 and 0.95 on different datasets among papers. By setting different thresholds of IoU in the classification stage, the mAP varies. In the project, this IoU threshold is set to 0.5 and the evaluation is based on it.

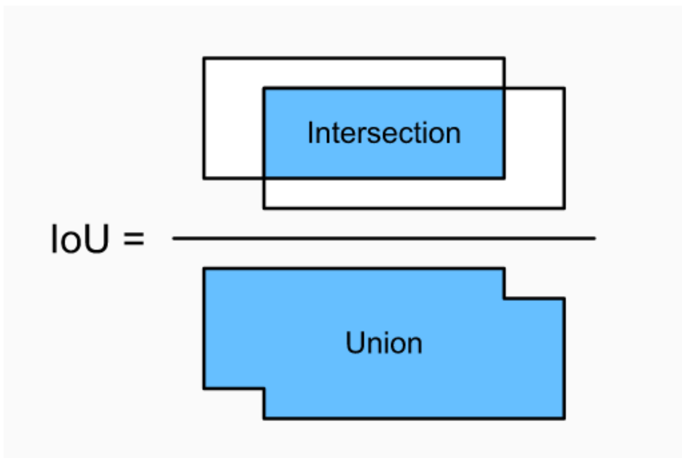


Fig 5. Illustration of IoU [17]

3.6.2 Non max suppression

IoU is also applied in non max suppression. Non max suppression is functional when there are more than one predicted bounding box candidates in a specific region of an image. It is used to reduce the number of bounding box candidates to one by ignoring redundant, overlapping bounding boxes. Non max suppression is able to ignore the small overlapping bounding boxes and return only the large ones so that it is also applicable in multiple object detection. There exists a threshold of IoU in non max suppression to determine it. If IoU in non max suppression is larger than this threshold, bounding boxes will be dropped, while bounding boxes will be kept if IoU is smaller. An example of this project can be seen in Fig 6.

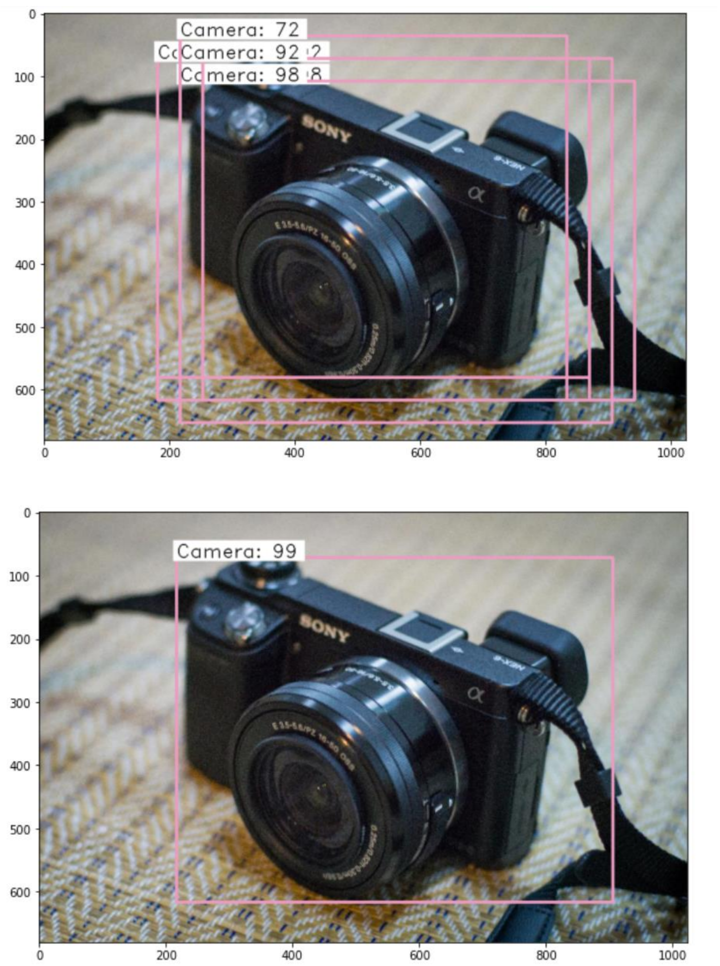


Fig 6. Examples of before and after applying non max suppression in the order of top to bottom, where the threshold of the top one is set to 1 and that of the bottom one is set to 0.

3.6.3 mAP calculation

Since IoU helps to determine positive cases, it is applied to obtain precision and recall. In this project, the average precision score is calculated using average_precision_score function in sklearn.metrics, the function of calculation can be seen in Fig 7. P denotes precision and R denotes recall. R contains true binary labels and P contains target scores, which is the probability estimates of the positive class.

$$\mathrm{AP} = \sum_n (R_n - R_{n-1}) P_n$$

Fig 7. Function of average_precision_score of sklearn.metrics to calculate AP [21]

3.7 Multiple camera detection task

Due to the characteristics of R-CNN structure, the detection is region-based just as the name indicates. A single CNN network may not be able to locate multiple objects, but R-CNN allows more than one region candidates for further searching. In the Faster R-CNN, anchor boxes plays an important role. An anchor is a box, and the default value is 9 at each position. With the stride of 16, a feature map with 37×50×256 dimensions can be outputted from a 600×800×3 image. This leads to 1850 (37×50) positions. And each position there are 9 anchors with different color or scales, which makes it 16650 (1850×9) anchor boxes. This makes detection of small objects and multiple objects possible. With a regional proposal network (RPN), a number of regions of interest will be selected, which can significantly reduce the number. Depending on the number and boundaries of regions of interest, the coverage of training resources regarding variants of camera objects, the feature similarity between training and testing data, angles, lighting, contrast of the testing images etc., the research question ”how many cameras can be detected?” can be answered.

Chapter 4 – Specification

4.1 Dataset reselection

For the trial, the purpose is to make the detection model work, so the selection of the dataset is not necessary to be very strict. As stated in chapter 3, 1000 random images containing camera objects are chosen for training. To match the detection model with a specific detection task, a reselection of the dataset is crucial. Since smartphones with built-in cameras are popular today, detection models should be extended to detect phone cameras. In addition, limited to the content of detection, which is daily scenes, some adjustments are also made for existing camera images. Accordingly, a number of criteria are listed below:
Camera class:

- Remove images of cameras with only screen side
- Remove images of professional filming cameras (Big, can be easily noticed, little chance appearing in public places, so not necessary for detection)
- Remove images of a few phone cameras to avoid duplicates of images in Mobile phone class

- Remove surveillance cameras, robot built-in cameras, too old cameras and digital camera icons etc. and only keep handheld cameras

Mobile phone class:

- Images of phones with only back side (main camera side)
- Only keep smartphone images, so filter out the old keyboard mobile phones

4.2 Feature extractor selection

The convolutional layer and the classifier layer have important roles in the Faster R-CNN network. In the workflow of Faster R-CNN model, A pre-trained CNN network is used to work as a feature extractor on image classification tasks. The region proposal task is initialized by the convolutional layer in the Faster R-CNN network and a feature map will be generated. The feature map is then passed to the Region Proposal Network (RPN) to have proposed regions. Therefore, the choice of feature extractor will have influence on what features can be processed in RPN. As shown in Fig 7, feature extractors have different accuracy levels and the choice of feature extractor will impact the accuracy of the detector (mAP). Another finding from this figure is that the choice of feature extractor has a greater impact on Faster R-CNN and R-FCN architecture than SSD. In this project, the VGG16 network is first used as the feature extractor, the same as that in the official paper. Afterwards, ResNet50 (Residual neural network) is used to have some trials. Compared to ResNet50, ResNet101 has more layers and is more expensive in computation, as VGG19 to VGG16. As shown in Fig 8, the choice of feature extractor will also have influence on the GPU running time on testing images in milliseconds. Since the project focuses on the detection on still images, processing time will not be a concern, so that achieving high accuracy will be a top priority.

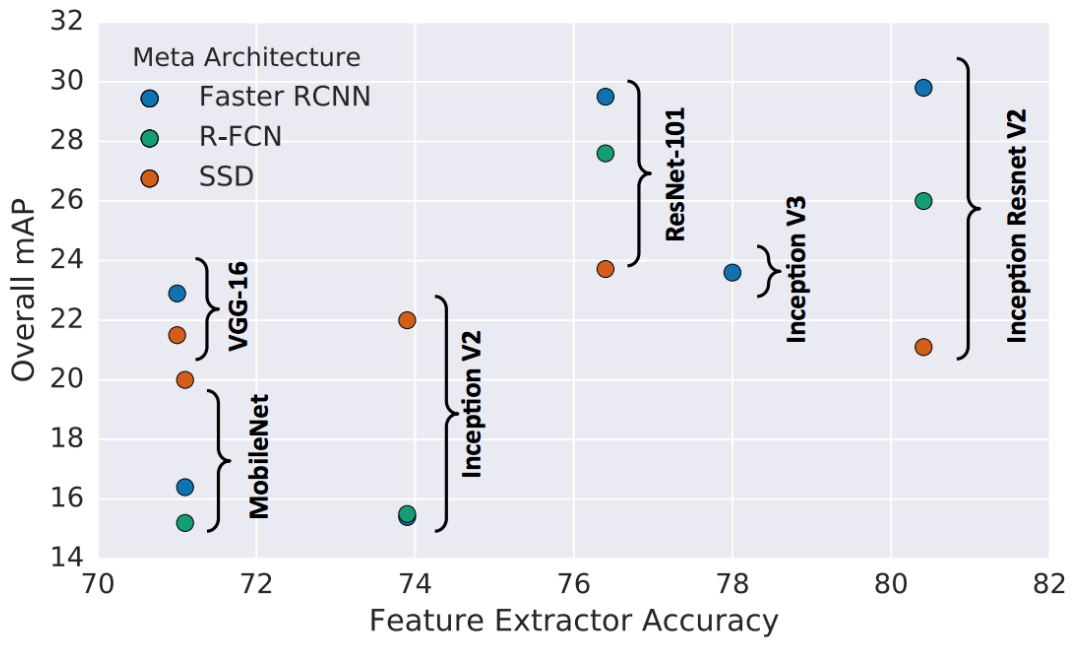


Fig 7. Accuracy of detector (mAP on COCO) vs accuracy of feature extractor (as measured by

top-1 accuracy on ImageNet-CLS) [16]

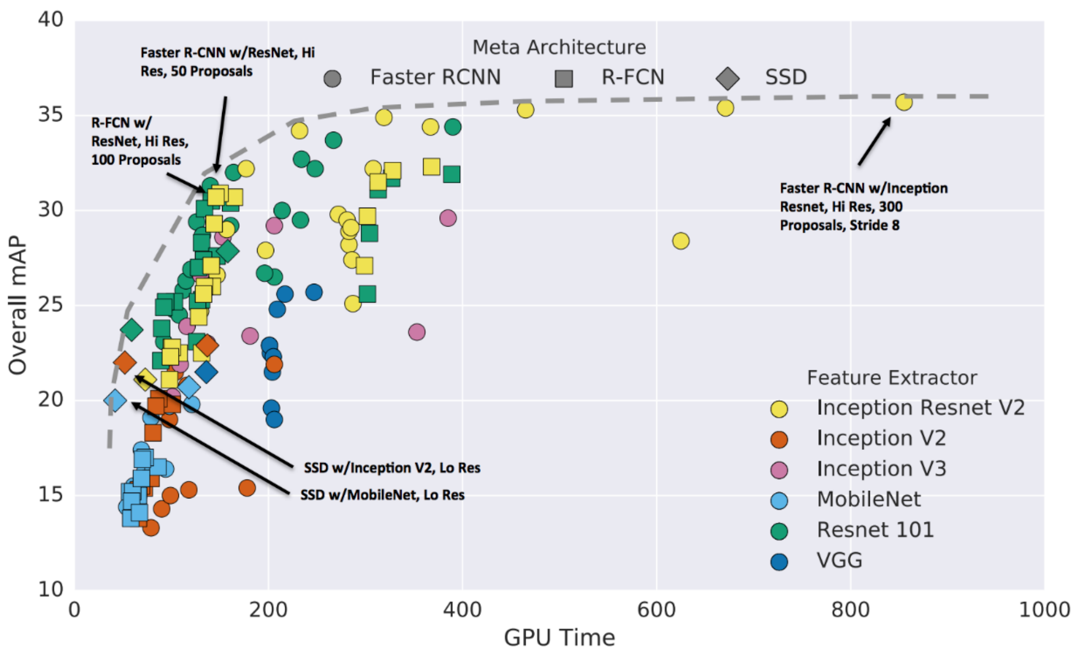


Fig 8. Accuracy vs time, with marker shapes indicating meta-architecture and colors indicating feature extractor [16]

4.2.1 Description of VGG16 and Resnet50 network

4.2.1.1 VGG16

VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman [18]. The architecture can be seen in Fig 9. It consists of five blocks, where there are a stack of convolutional layers and pooling layers. This is followed by three fully connected layers and a softmax layer. All convolutional layers are also equipped with ReLU, which stands for rectification. Due to its depth and a number of fully connected nodes, VGG16 is more than 500MB, which is very large compared to many other networks. The application of VGG16 network can be seen in the base layer and classification layer of Faster R-CNN network. This also starts from the pretrained weight file from keras.

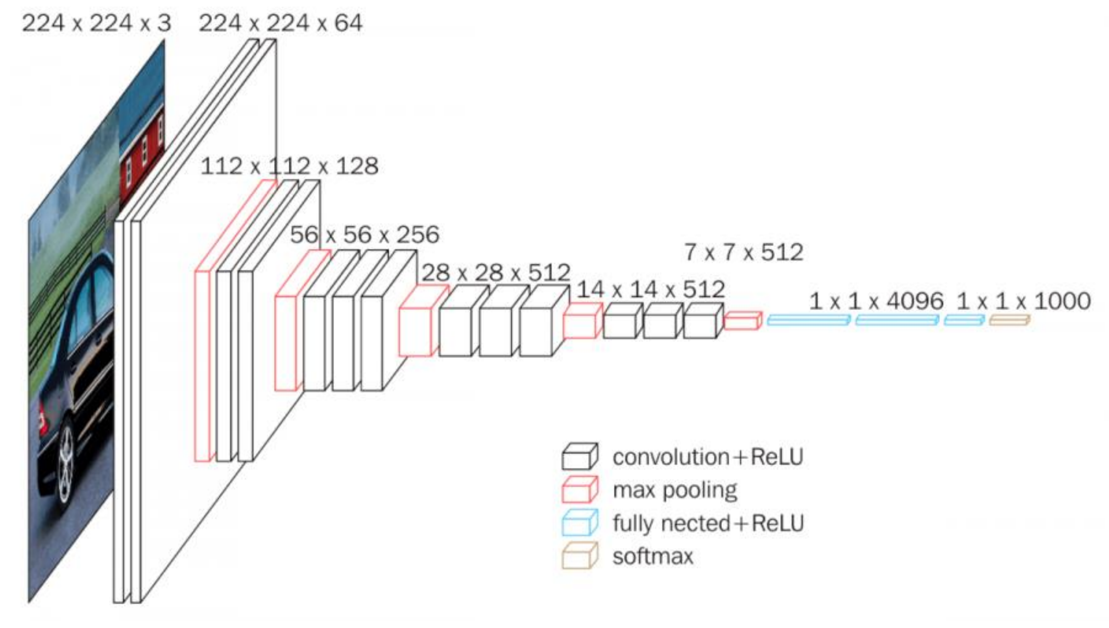


Fig 9. Architecture of VGG16 network. [22]

4.2.1.2 ResNet50

The ResNet50 Model proposed by K. He et al. [19] consists of 5 stages, which can be seen in Fig 10. From stage 2 to stage 5, each stage contains a convolutional block and an identification block. Each convolutional block and each identification block have 3 convolution layers. There is a skip connection function in this network. If the convolution and batch norm operations in convolutional block or identification block are done in a way that the output shape is the same, there will be a shortcut created. The skip connection can add the output from earlier layer to later layer, which helps to mitigate the vanishing gradient problem.

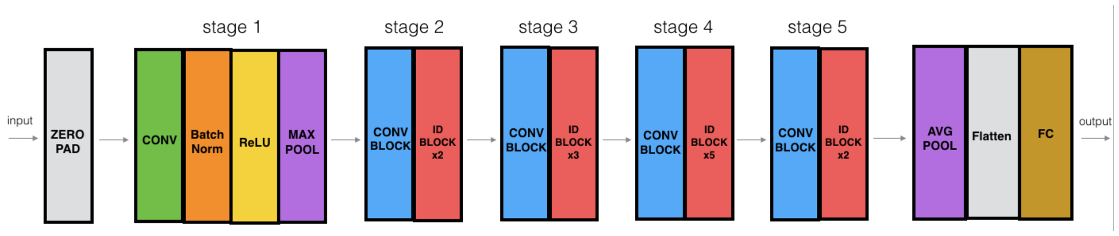


Fig 10. Architecture of ResNet50 network [22]

4.3 Number of region proposals

Region proposals are regions of interest. Faster R-CNN is a two-stage detection network. In the first stage, features outputted from an intermediate level of the feature extractor is selected and used to predict class-agnostic box proposals [17]. Usually the first stage ends at the second last block or stage of the network of the feature extractor. In the second stage, the box proposals are used to crop the features from the feature map and then fed to the remainder of the feature extractor (the last block or stage of the network)) in order to predict a class and class-specific box refinement for each proposal. Thus, the number of proposals will affect the feature extraction. It is also proved in Fig 11, where an increasing number of box proposals will make a difference in detection accuracy, but it varies among different feature extractors. According to the paper from S. Ren at al. [8], the number of proposals is set to 300 when using VGG and RPN. In this project, the number of box proposals is set to 4 at the beginning for saving GPU time. A higher number will also be tested afterwards.

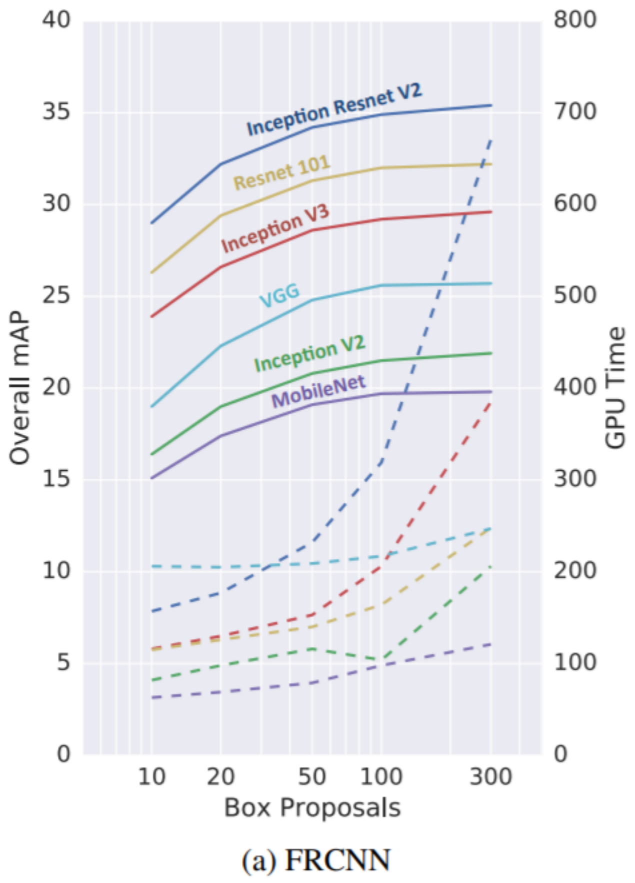


Fig 11. Effect of proposing increasing number of regions on mAP accuracy (solid lines) and GPU inference time (dotted) for Faster R-CNN [16]

Chapter 5 – Realization and evaluation

This chapter covers all the important experiments. All experiments are controlled experiments. One variable or one setting is tested and evaluated between two experiments at a time. This can bring out clear comparisons between two experiments in order to make improvements on detection performance.

5.1 Model 1: Using VGG16 network as feature extractor

The first experiment builds on Faster R-CNN architecture with VGG16 CNN network as the feature extractor. Since Faster R-CNN is a two-stage detection network, the first 4 blocks of the VGG16 network are applied in the base layer, which is the convolution layer. The base layer will output a feature map in order to predict class-agnostic box proposals. The base layer is shared with the RPN layer and classification layer. The last block of VGG16 is applied in the classification layer to predict class-specific boxes by cropping the features outputted from the base layer.

5.1.1 Training setting

- Dataset: 1000 randomly selected camera images from the category of Camera of Open Image Dataset (800 in the training set and 200 in the testing set)
- Learning rate: 0.0001
- Batch size: 1000
- Number of regions of interest: 4
- Image size: 300 px

5.1.2 Training process evaluation

Results are evaluated after training 80 epochs, 100 epochs, 120 epochs and 140 epochs separately. The performance of the detector after training 120 epochs is the best among four. The changes in the training process can be seen from Fig 12 to Fig 17. Fig 12 shows the graph of the mean number of bounding boxes from RPN overlapping ground truth boxes. The number grows very fast within the first 20 epochs and grows slowly with fluctuations in the rest epochs. It indicates how the localization of camera objects becomes precise over time. It can also represent the accuracy of RPN.

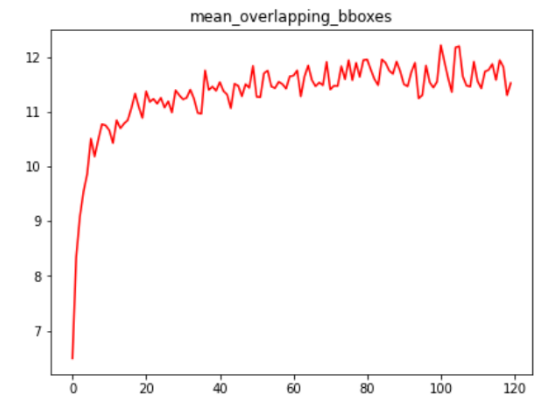


Fig 12. Mean number of bounding boxes from RPN overlapping ground truth boxes

Fig 13 shows how the refined bounding boxes from RPN fit the ground truth. The slope of the curve changes less quickly than that of mean overlapping bounding boxes. In general, but the curve has an increasing pattern as well. It indicates the accuracy of the classifier.

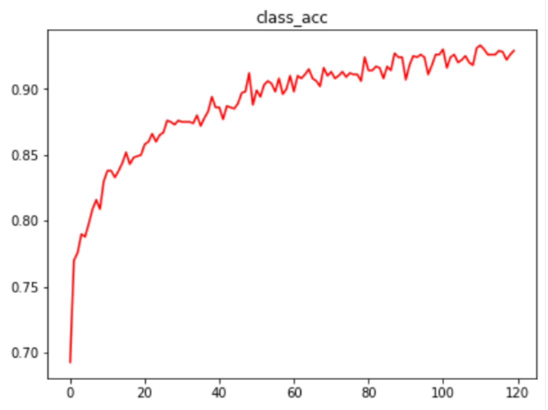


Fig 13. Classifier accuracy for bounding boxes from RPN.

Figure 14 shows the loss curve of the RPN classifier and that of RPN regression. They both present a trend of decreasing over time. The curve of the loss of RPN classifier becomes stable after 60 epochs of training, which is satisfying.

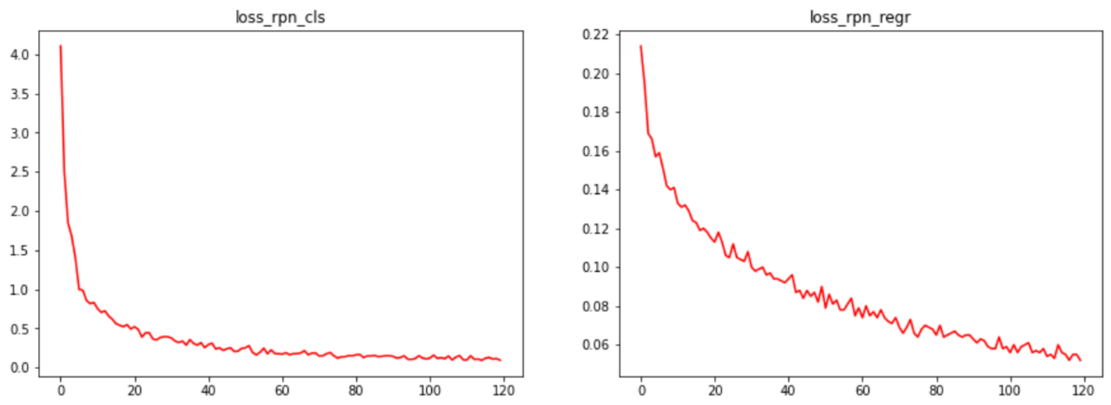


Fig 14. Loss of RPN classifier and loss of RPN regression

The loss of detector classifier and loss of detector regression can be seen in Fig 15. These two curves have similar patterns and both look like the curve of the loss of RPN regression.

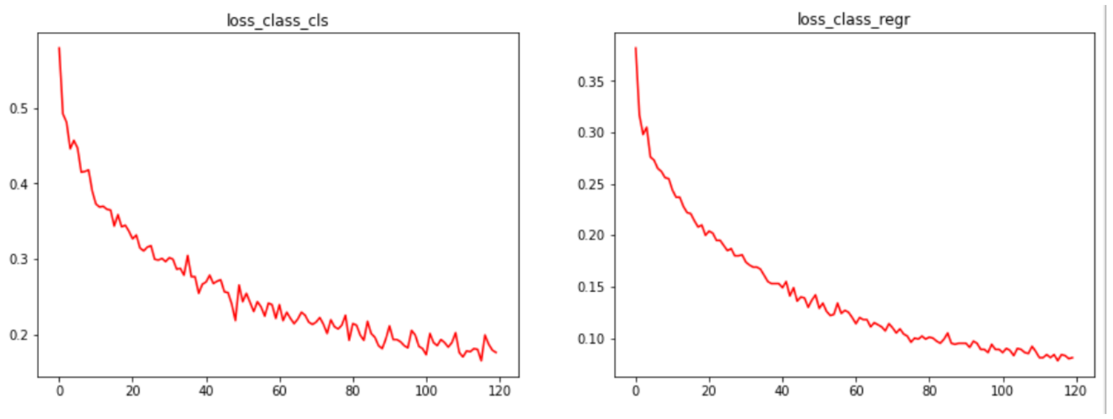


Fig 15. Loss of detector classifier and loss of detector regression

The total loss in Fig 16 is the total loss of all the losses in Fig 14 and Fig 15. The curve has a similar pattern to that of the loss of RPN classifier since the loss of RPN classifier takes the major part of the total loss and thus has the biggest effect. It indicates the predicting ability of the detector is becoming stable over time.

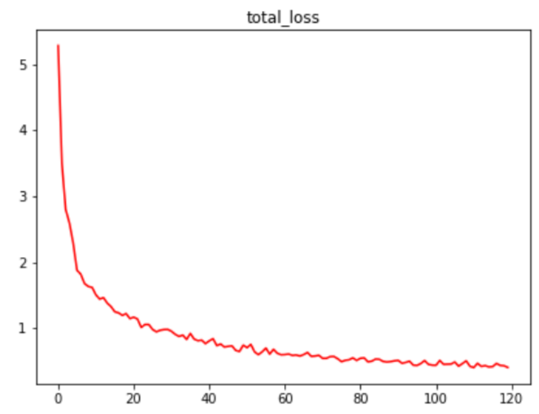


Fig 16. Total loss

The graph below shows the elapsed time in the training process. The total training process is split into 3 sessions due to the need of testing detection models and avoiding overfitting. Three maximums are noticeable in the graph, which were exactly at the starting point of each training session. This is probably because the loading of saved weights requires extra time. At each training session, the elapsed time is stable among epochs in general but may differ between sessions. The changes in the training environment might be the reason, since the training relies on the GPU of Colab.

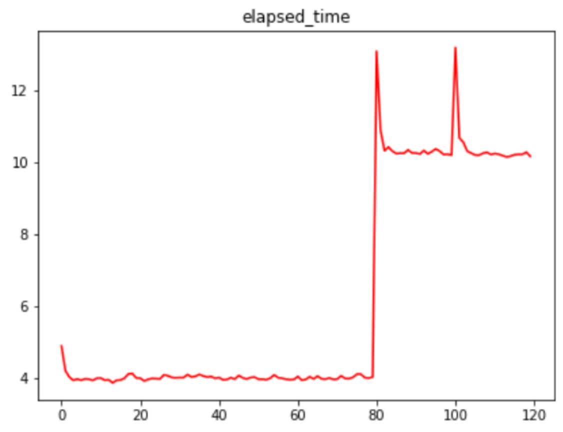


Fig 17. Elapsed time of training in epochs

5.1.3 Testing process evaluation

The mAP of performing testing in the test set of 200 images after training 80 epochs, 100 epochs, 120 epochs and 140 epochs is 0.601, 0.673, 0.684 and 0.642 separately, which can be seen in Fig 12. Also, the percentage of images without bounding boxes among all is 0%, 2.5%, 1.5%, 1.5% separately. This indicates the false negative rate under these training settings is very low. After training 140 epochs, the detection turns to be overfitting since the mAP drops from 0.684 to 0.642. The performance can also be explained through sample images outputted from the testing file. The average testing time per image is 0.62 second. Overall, camera objects can be correctly located after training 80 epochs and more. False positive cases happen more frequently after training 80 epochs and 100 epochs than after training 120 epochs. This is because the detection is refined after more iterations. They are also more false positive cases after training 140 epochs compared to training 120 epochs. This is probably due to overfitting of the detection model.

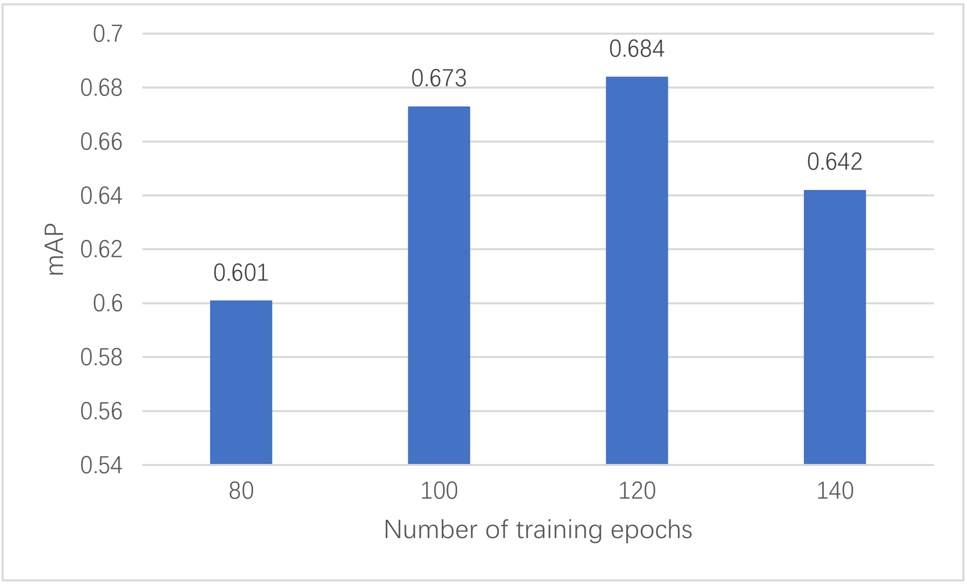


Fig 12. mAP values after training different numbers of epochs.

Some sample tested images are collected, which can be seen in Fig 18, 19 and 20. T , so only the classification value is greater than the threshold, the bounding box is drawn in the tested images. Fig 18 contains the sample images of single cameras that are correctly detected. It is obvious that cameras can be correctly located, even small ones. The boundaries of bounding boxes seem to be fine as well.

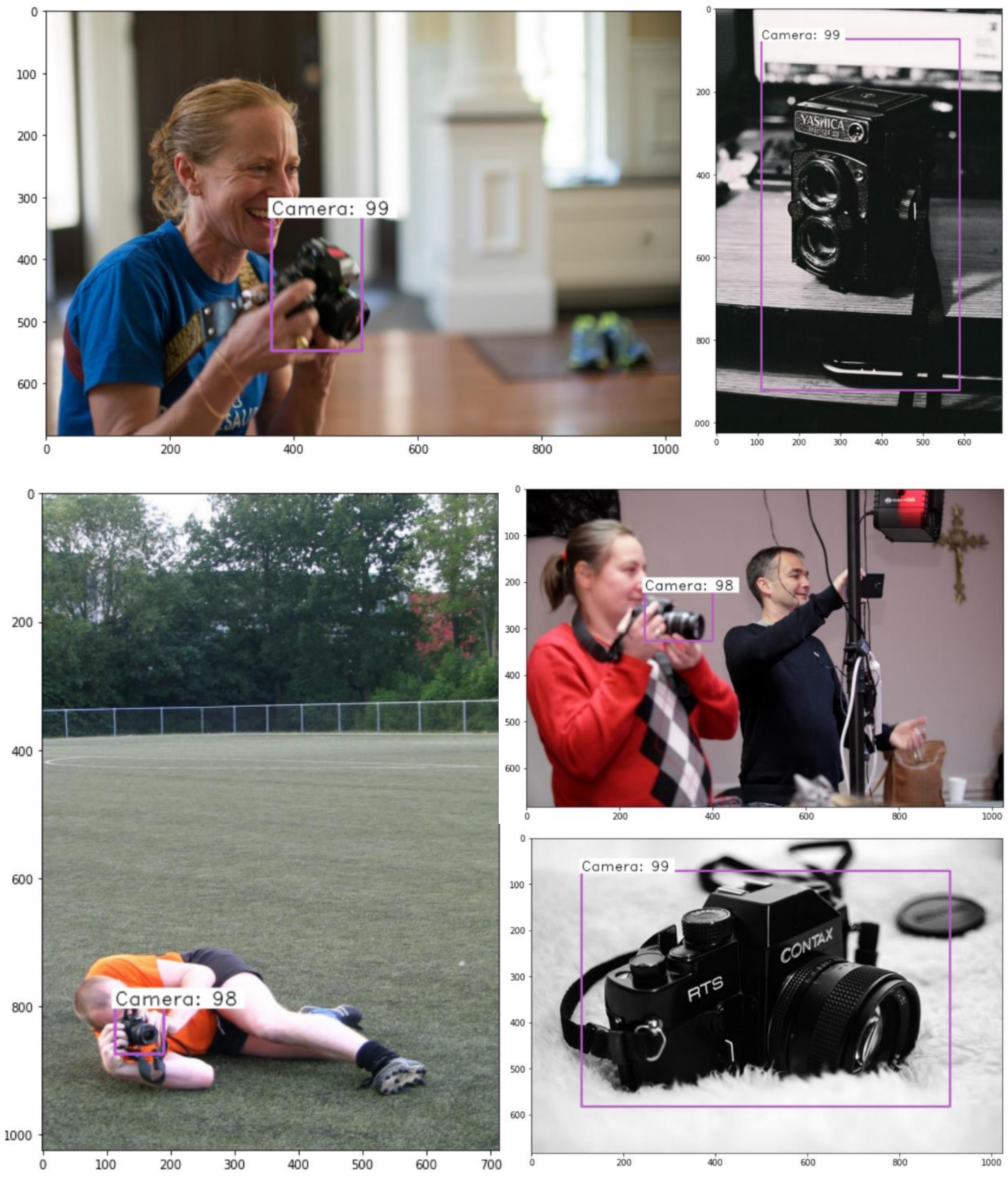
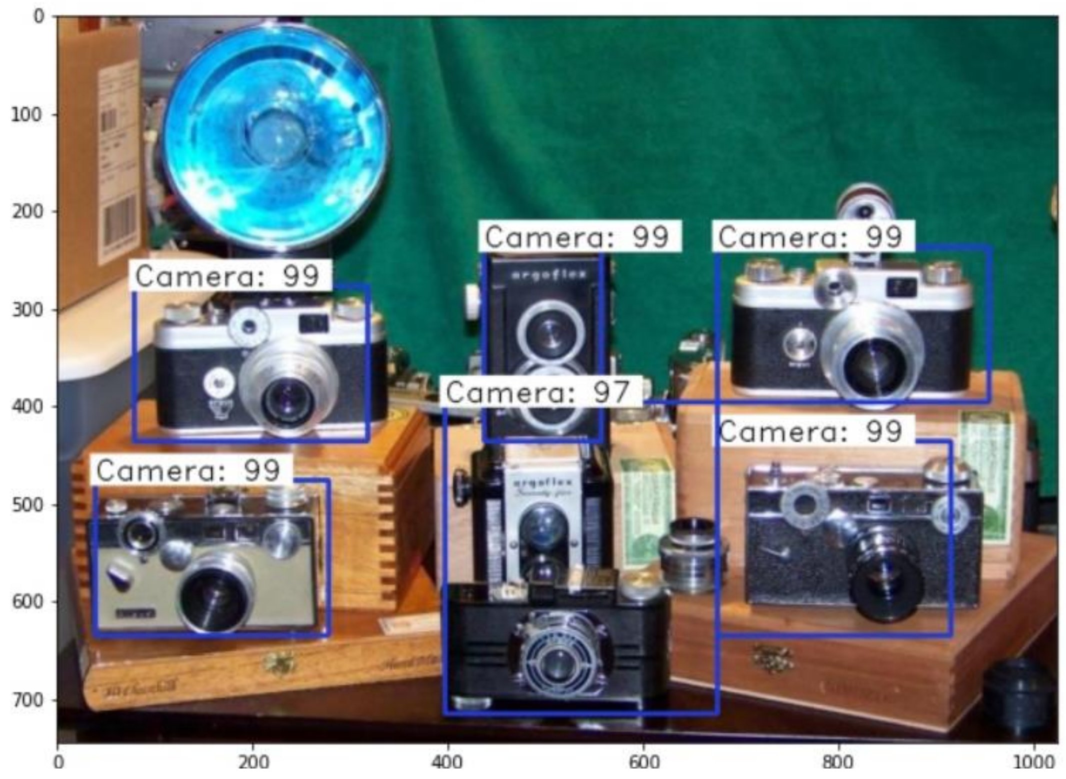


Fig 18. A image collection of successful detected single cameras

Multiple cameras can also be detected. Some sample images are shown below. From Fig 19, the image on the bottom left contains three cameras, but only one big bounding box is drawn. This is probably due to the IoU calculation, where the IoU of the big box with any of the ground truth boxes is greater than the threshold and smaller boxes with large overlapping areas are dropped. As a comparison, the top image also contains multiple camera objects, while all of them are in a distance with each other so that the bounding boxes are drawn separately. The bottom right image shows a big bounding box that contains multiple cameras as well, but some cameras are outside the box. IoU in non max suppression and the feature of the image probably are the reasons. With the feature of a "U" shape of multiple cameras, bounding boxes with small intersection areas are dropped.



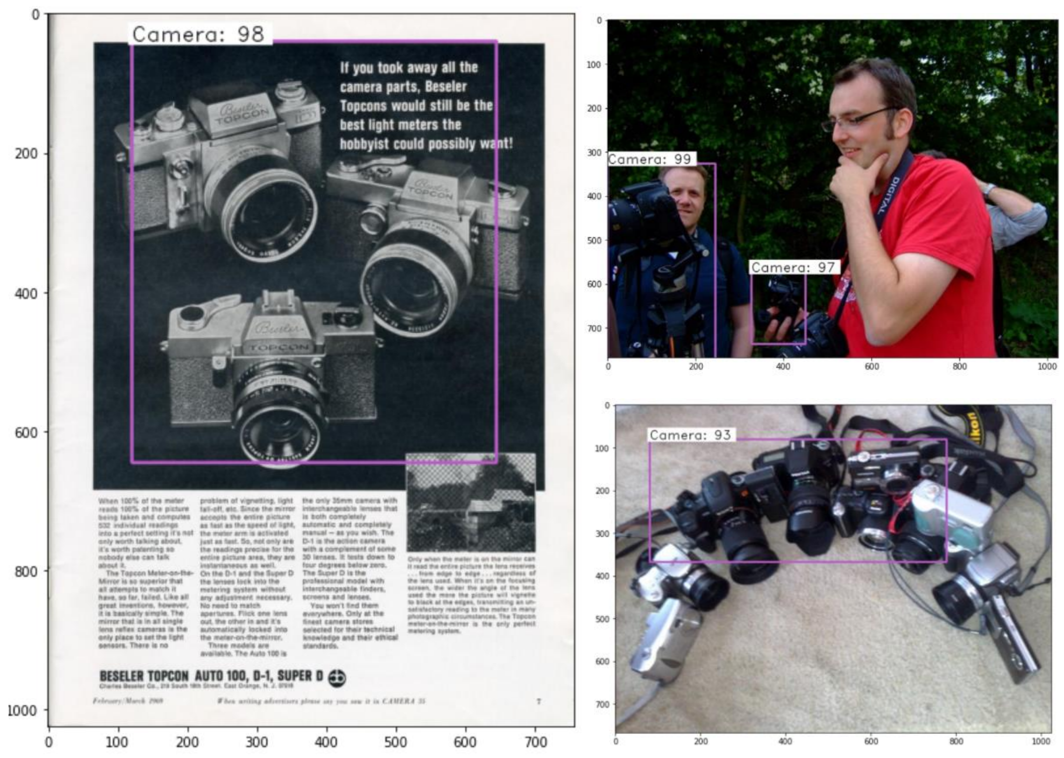

Camera: 98
If you took away all the camera parts, Beseler Topcons would still be the best light meters the hobbyist could possibly want!
BESELER TOPCON AUTO 100, D-1, SUPER D
Camera: 99
Camera: 97
Camera: 93

Fig 19. A image collection of successful detected multiple cameras

Some false positive cases can be seen in Fig 20. The reason can be the similar patterns with some cameras or camera parts. The bottom left picture indicates that a painting with frames can also be regarded as the screen of a camera because of similar features. In the final stage of this project, the screen side of the cameras is not necessary to detect since the goal is to detect photographing or filming, where the lens side of the camera should be faced with instead of the screen side. Therefore, a reselection of training images is needed.

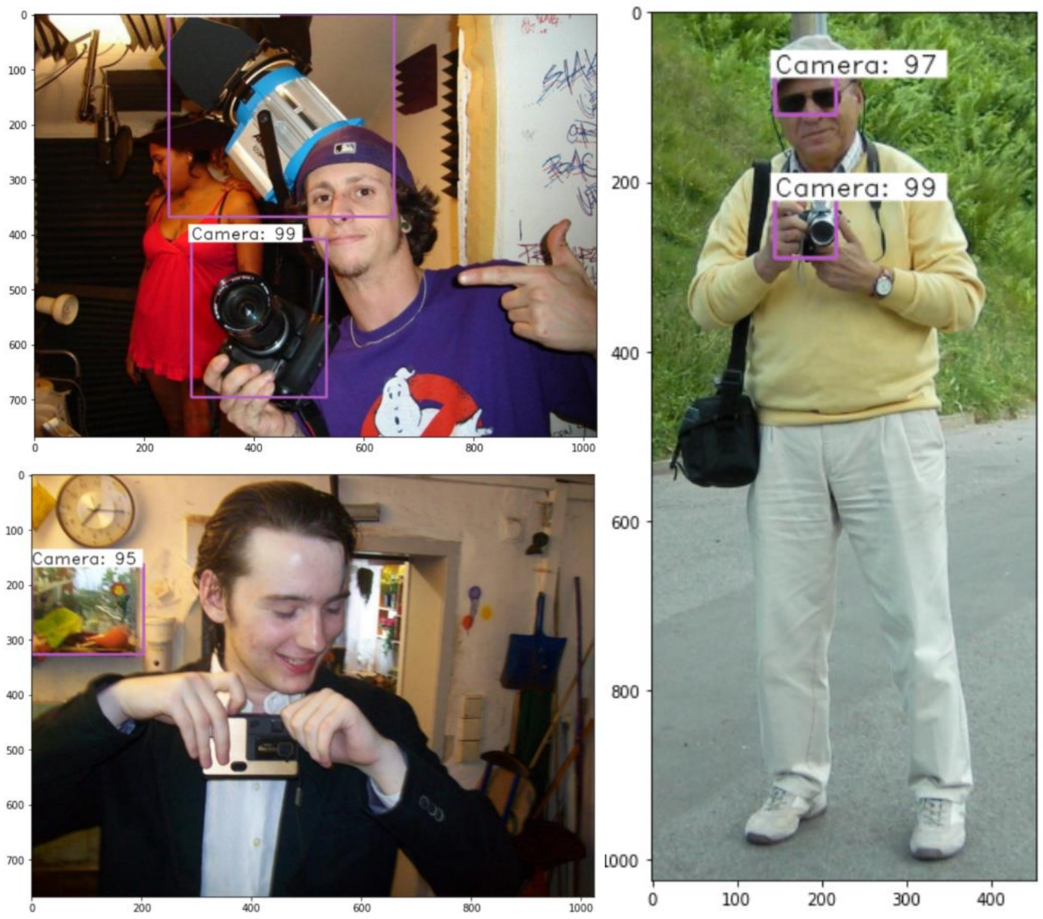


Fig 20. A image collection of false positive cases

There are also some images without bounding boxes, which can be seen in Fig 20. This is probably due to the classification value of bounding boxes being lower than the threshold or the limitation of detector or training data.

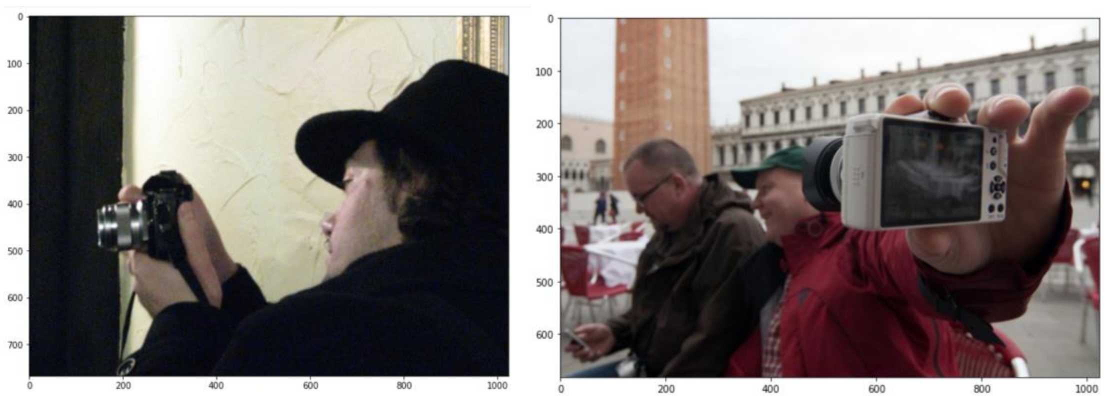


Fig 20. A image collection of false negative cases

5.1.4 Reflection

Overall, the result is satisfying, the detection model can correctly locate most camera objects with different sizes, types and angles. It can also detect multiple camera objects well. This model has a decent performance regarding mAP. The bounding boxes of camera objects that are drawn can detect boundaries of camera objects correctly with high confidence scores. The model has a low false negative rate as well. Although the result is satisfying, there is still space for improvement. This can be found out by changing training settings, feature extractor and the selection of training data.

5.2 Model 2: Using ResNet50 network as feature extractor

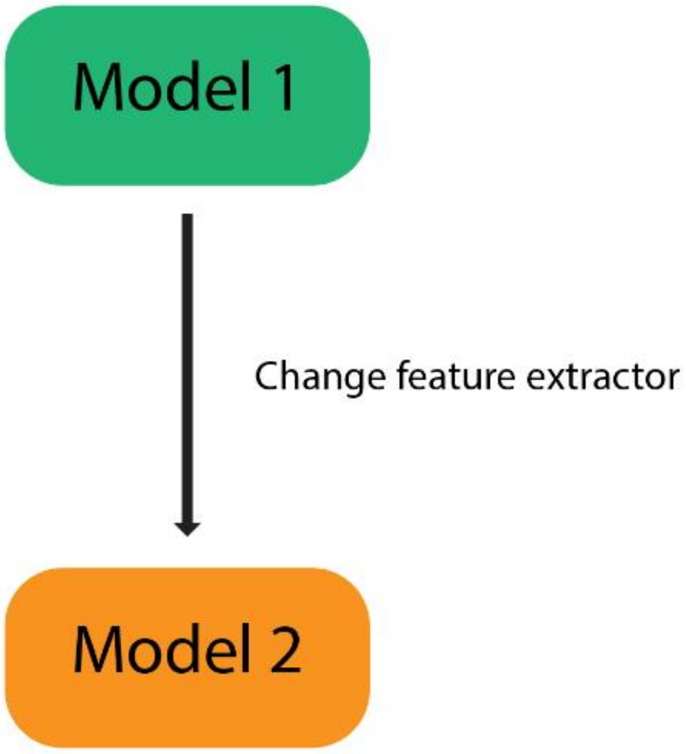


Fig 21. An illustration of the change between Model 1 and Model 2

The second experiment uses ResNet50 CNN network as the feature extractor. An illustration can be seen in Fig 21. In the base layer, VGG16 network with four blocks is replaced with ResNet50 network with 4 stages. The ResNet50 network has 5 stages. Identification block and convolutional block are defined separately and are used in stages. The first 4 stages are applied in the base layer and the last stage of ResNet50 replaces the last block of VGG16 in the classification layer. The input shape of Resnet50 is the same as VGG16, while the shape of feature map input changes and the number of features changes from 512 to 1024.

5.2.1 Training setting

The training settings are the same as Model 1.

- Dataset: 1000 randomly selected camera images from the category of Camera of Open Image Dataset (800 in the training set and 200 in the testing set)
- Learning rate: 0.0001
- Batch size: 1000
- Number of regions of interest: 4
- Image size: 300 px

5.2.2 Training process evaluation

The performance of the model becomes stable after training 240 epochs. The graphs about bounding boxes, accuracy and loss in the training process can be seen from Fig 22 to Fig 25. Overall, all the curves have similar trends as the ones of Model 1. Different from Model 1, the curves of mean number of overlapping boxes and classifier accuracy in Fig 22 grow faster in the early stage. Also, the curve of the mean number of overlapping boxes has a tendency to drop after training 150 epochs, but the numbers are still over 11, which is acceptable. The

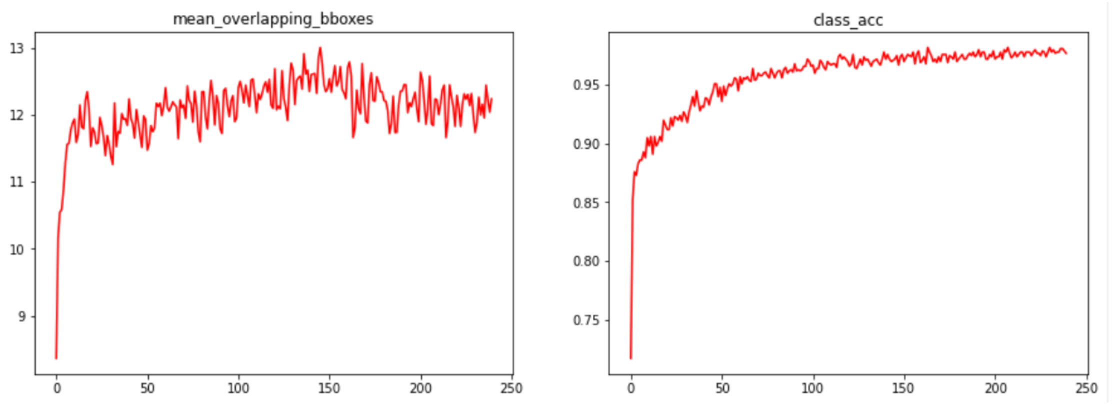


Fig 22. Mean number of bounding boxes from RPN overlapping ground truth boxes (left) and classifier accuracy for bounding boxes from RPN (right)

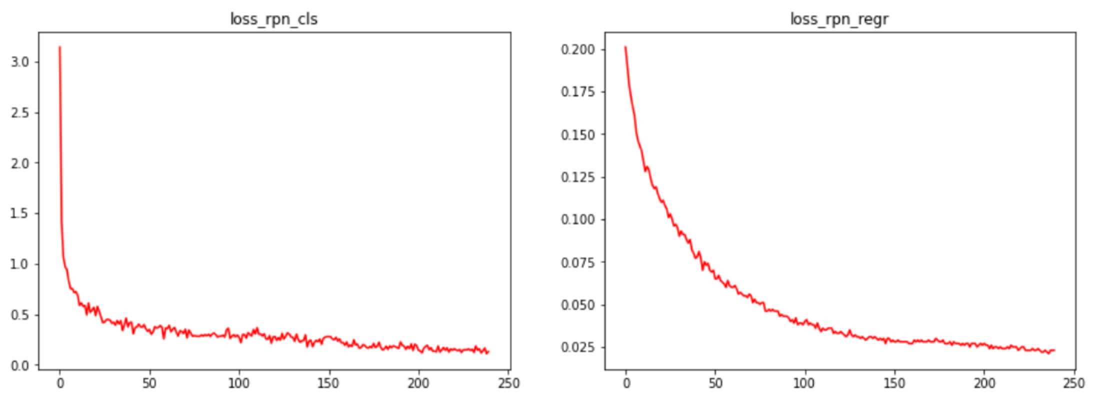


Fig 23. Loss of RPN classifier and loss of RPN regression

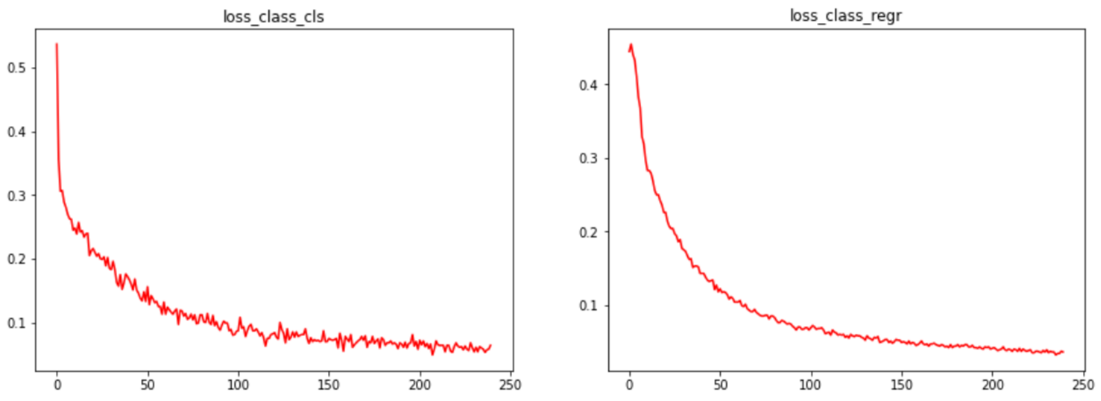


Fig 24. Loss of detector classifier and loss of detector regression

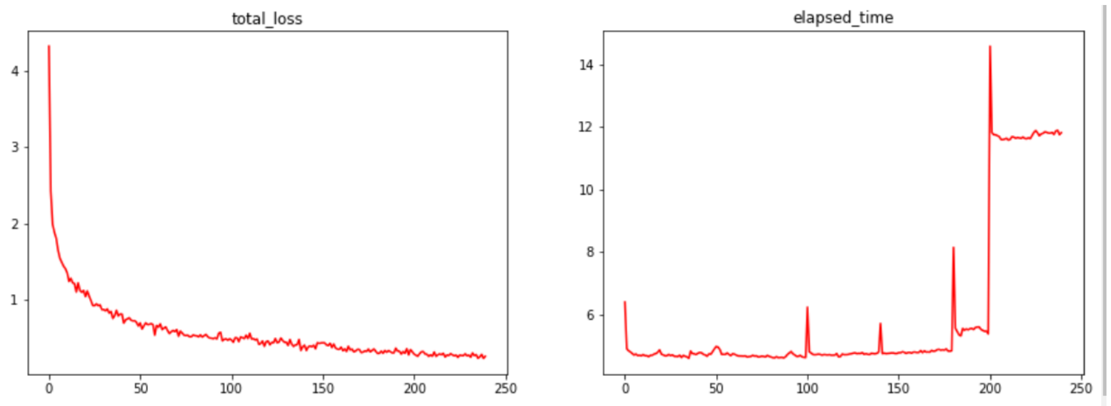


Fig 25. Total loss and the elapsed time of training in epochs

5.2.3 Testing process evaluation

After training 240 epochs, with the batch size of 1000, the detection models gives a stable result of mAP, which is 0.609. It is lower compared to that of Model 1. In addition, the percentage of images without bounding boxes among all is 24.5%, which indicates a high false negative rate. This can also be seen from the outputted sample images, where images without bounding boxes appear more frequently than Model 1. The bounding box threshold is set to 0.9 (90%) as well. Among the images where camera objects are correctly located, boundaries of bounding boxes correspond to the boundaries of camera objects, which can be seen in Fig 26. Besides, false positive cases are barely seen among all sample images.

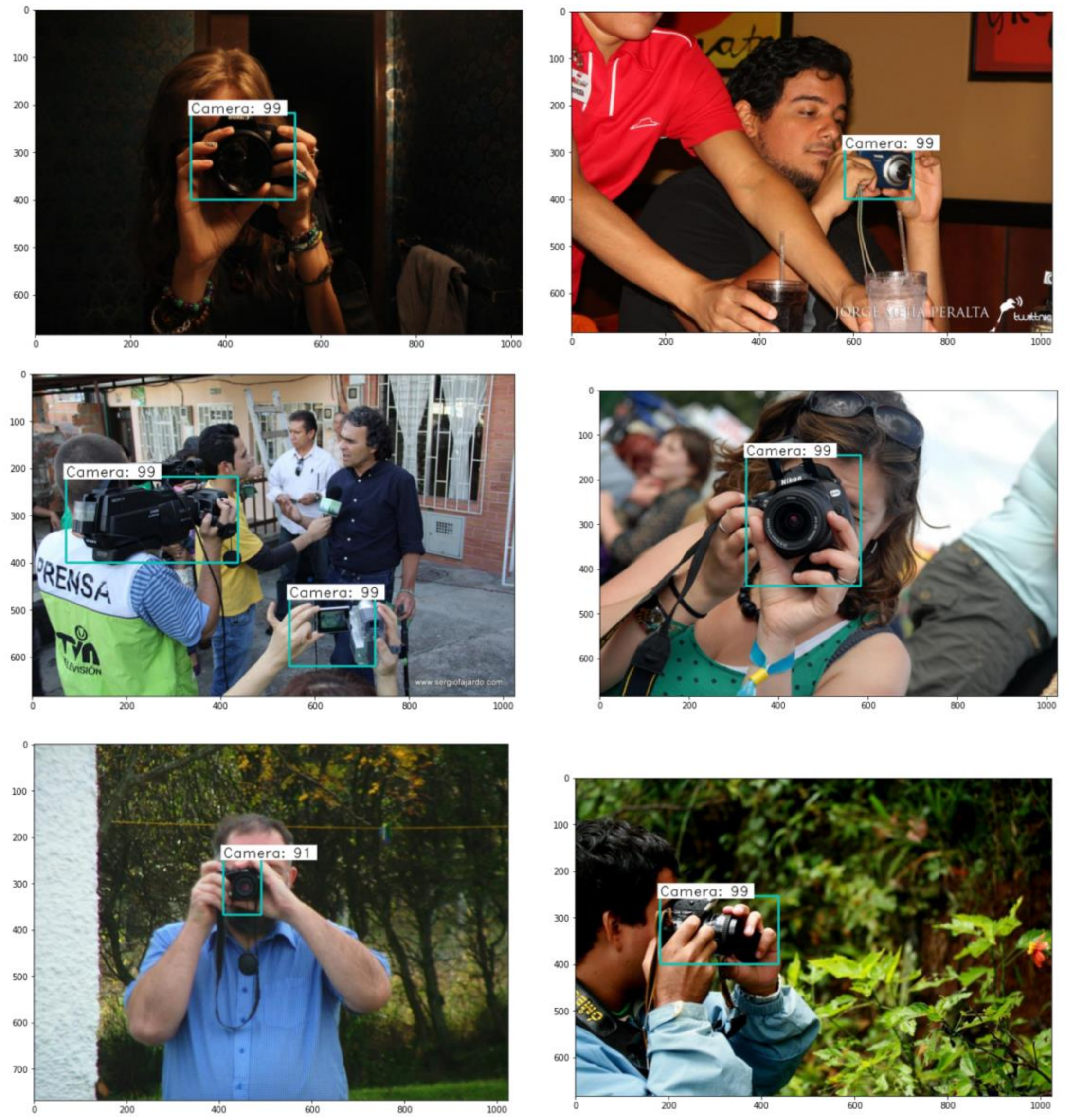


Fig 26. Sample images of successful detected camera objects

5.2.3 Reflection

Compared to Model 1, this model has a lower false positive rate and a higher false negative rate. The ResNet50 network has a deeper structure than VGG16, while the mAP of this model is not as good as Model 1. Since camera objects can be correctly in most of the cases, RPN and the classifier should work fine. Therefore, the feature generation might be a reason. In the official paper written by S. Ren et al. [8], the number of proposals is set to 300 when using VGG and RPN together. However, the number is set to 4 in Model 1 and different CNN networks might have different sensibilities to this. To analyse the effect of the number of proposed regions, another experiment will be made in the next step.

5.3 Model 3: Change the number of RoI of Model 2

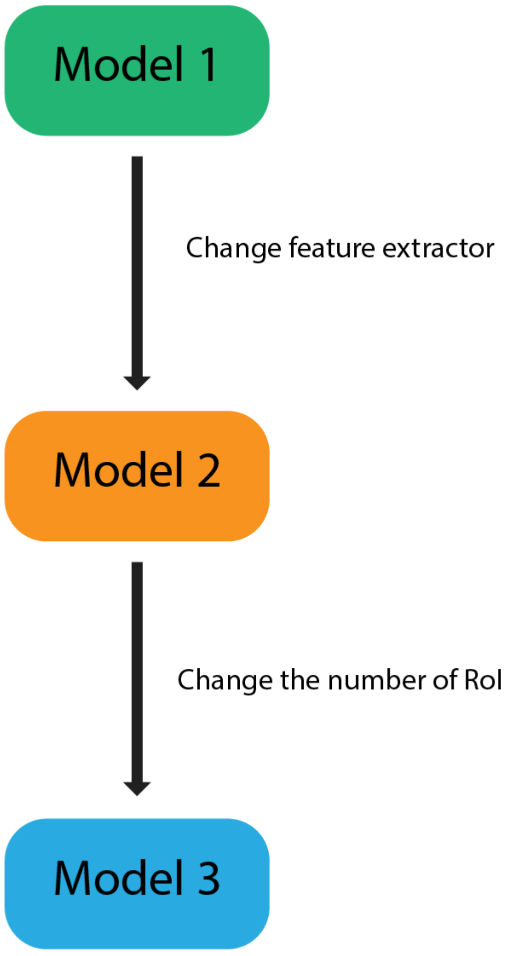


Fig 27. An illustration of the changes between models

This model is based on Model 2 with the ResNet50 network as the feature extractor so that the structure remains the same. Different from Model 2, the number of RoI is changed from 4 to 50, which can be seen from Fig 27. The number of RoI (region of interest) represents the number of proposed regions to be selected for predicting bounding boxes. While increasing the number of RoI, more memory is needed in the training phase.

5.3.1 Training setting

- Dataset: 1000 randomly selected camera images from the category of Camera of Open Image Dataset (800 in the training set and 200 in the testing set)

- Learning rate: 0.0001
- Batch size: 1000
- Number of regions of interest: 50
- Image size: 300 px

5.3.2 Evaluation and reflection

The performance of the model is evaluated after training different numbers of epochs such as 80, 100 and 120. After training 120 epochs, the mAP can reach 0.77, while the percentage of images without bounding boxes among all is 48%, which indicates a high false negative rate. Compared to Model 2 with the number of RoI equal to 4, the mAP increases, while the false negative rate even increases. Training other different numbers of epochs, the false negative rate is also very high. This shows that the model is probably overfitting and the implementation of the ResNet50 is unsuccessful. Thus, the model is given up for further training. Some sample images of successfully detected cameras can be seen in Fig 28. The average testing time per image is 0.28 seconds. It can be seen that successfully detected cameras are correctly located by bounding boxes, while not all cameras in one image can be detected. Although the implementation of ResNet50 network is not successful, this model shows the potential of gaining higher mAP by increasing the number of RoI.

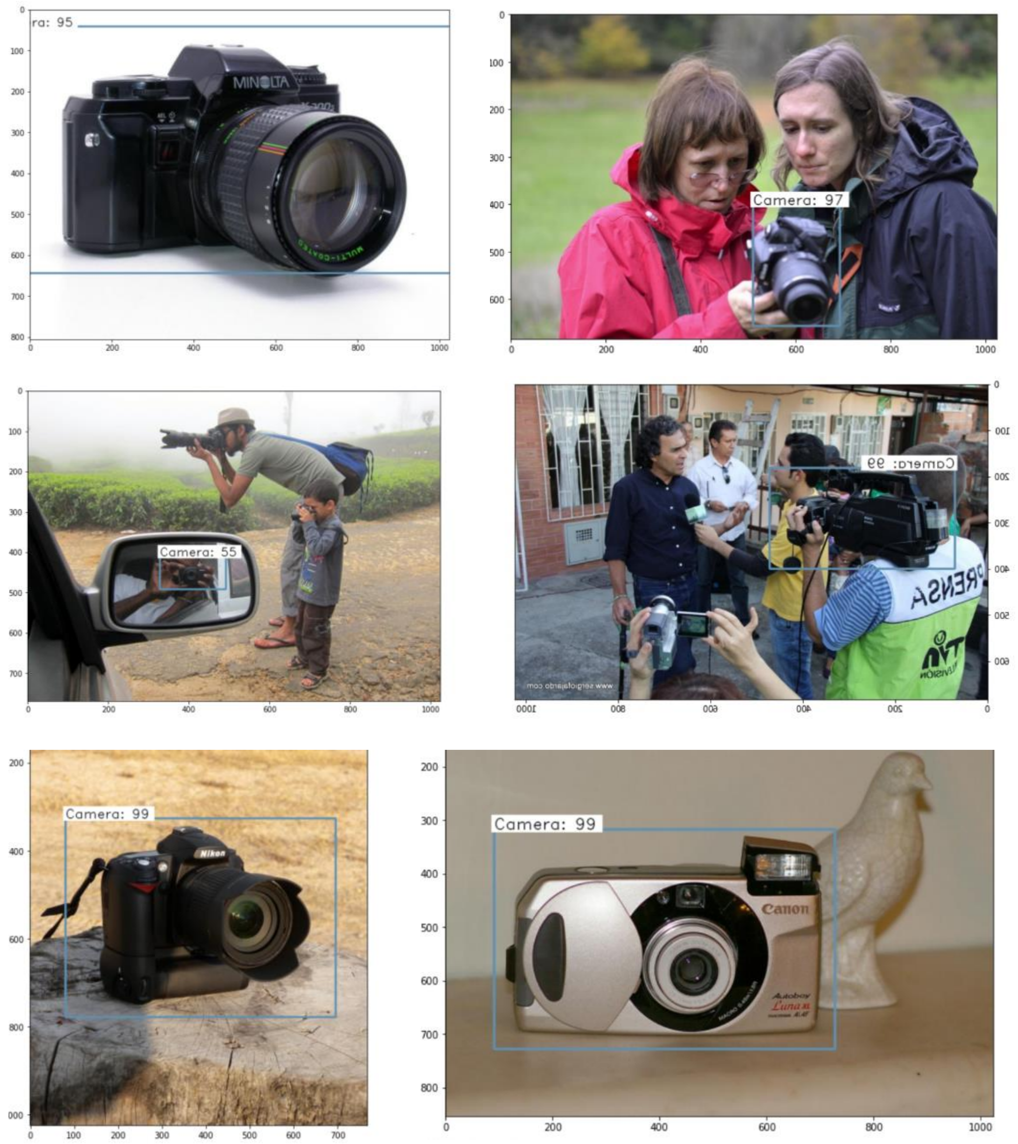


Fig 28. Sample images of successful detected camera objects

5.4 Model 4: Change the number of RoI of Model 1

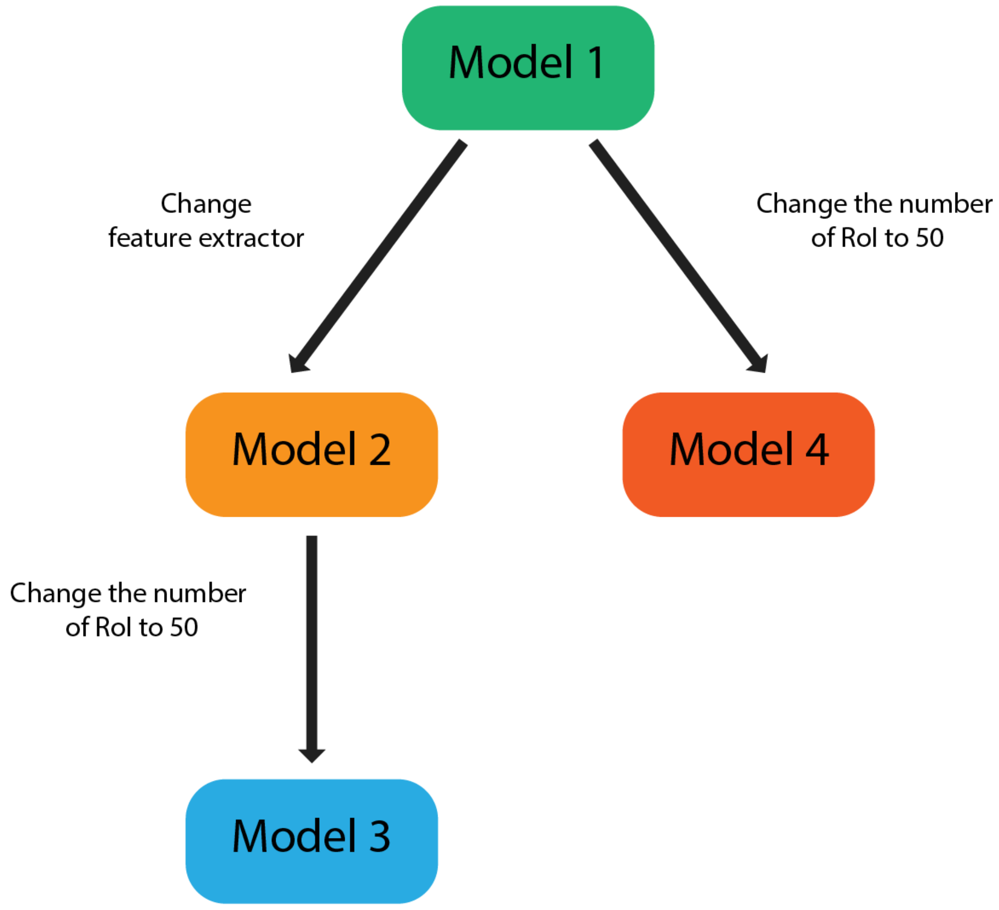

Model 1
Change feature extractor
Change the number of RoI to 50
Model 2
Model 4
Change the number of RoI to 50
Model 3

Fig 29. An illustration of the changes between models

This model is based on Model 1 with VGG 16 as the feature extractor so that the structure remains the same. The number of RoI is also changed to 50 like Model 3. An illustration of the changes can be seen in Fig 29.

5.4.1 Training setting

- Dataset: 1000 randomly selected camera images from the category of Camera of Open Image Dataset (800 in the training set and 200 in the testing set)
- Learning rate: 0.0001
- Batch size: 1000

- Number of regions of interest: 50
- Image size: 300 px

5.4.2 Evaluation

The performance of the model is evaluated after training different numbers of epochs such as 130, 150 and 170. It achieves the best result after training 150 epochs. The mAP reaches 0.764 and the percentage of images without bounding boxes is 4.5%. Compared to Model 1, the improvement of mAP is obvious. After training 170 epochs, the mAP drops and the model turns to be overfitting. The result shows that the increasing number of RoI can have a great impact on the performance of the model. Some sample images can be seen in Fig 30. The average testing time is 0.41 seconds. Overall, the cameras can be correctly located. There is no big difference between this model and model 1, while by setting the bounding box threshold to a lower value, there are only a few bounding boxes showing up. This indicates that there are fewer false positive cases compared to Model 1 so that the mAP is higher. For the mAP calculation, all the boundning boxes are considered, including those with relatively lower classification scores. This model implies that with a higher number of RoI, the model can have better performance.

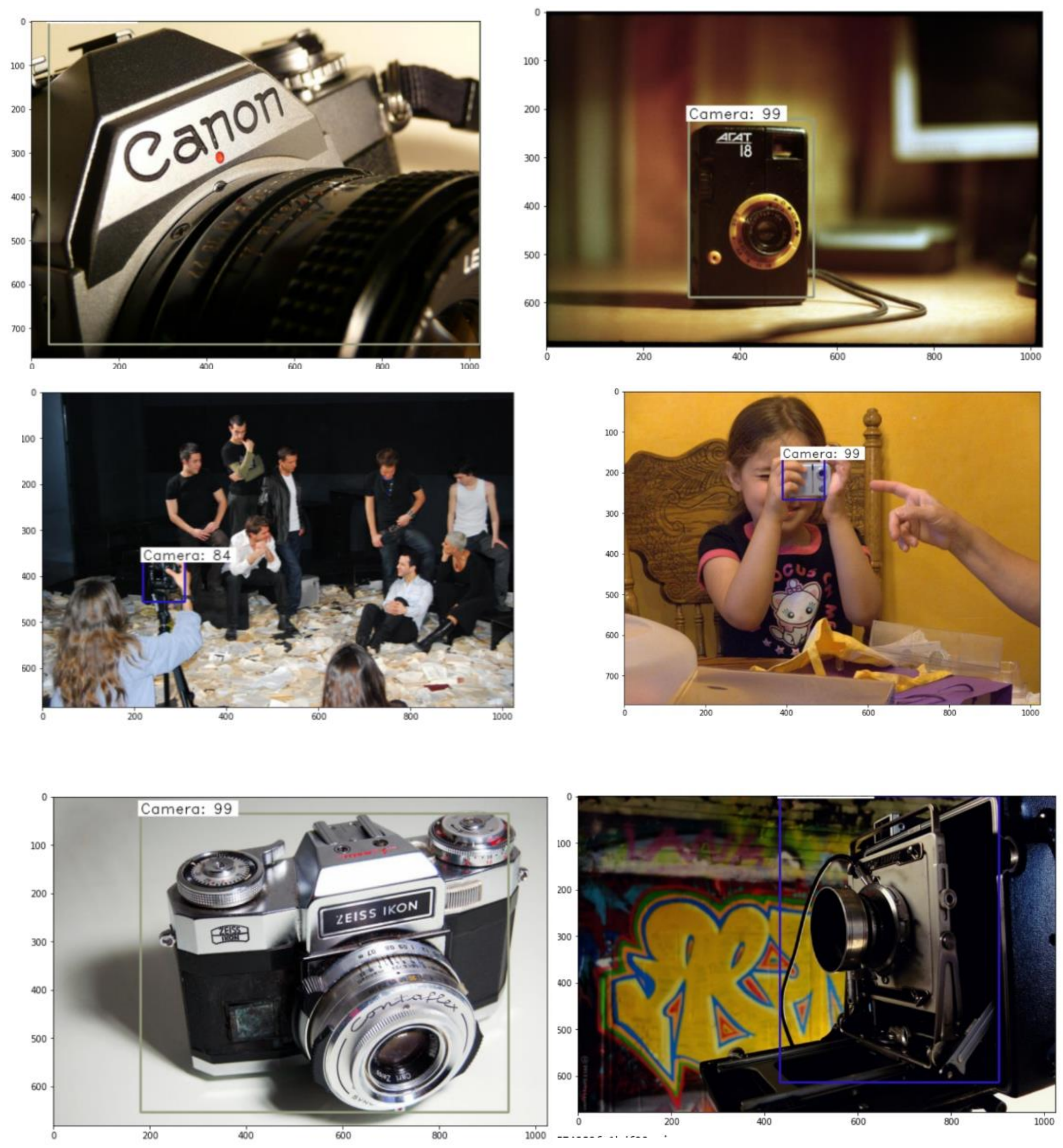


Fig 30. Sample images of successful detected camera objects

5.5 Model 5: Training data reselection from Model 1

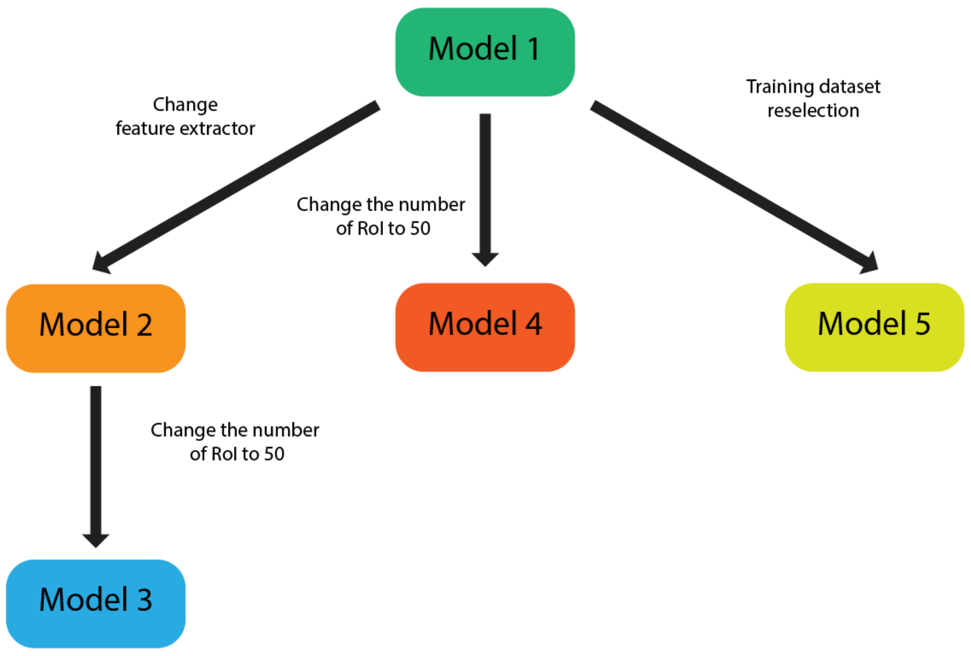

Model 1
Change feature extractor
Change the number of RoI to 50
Training dataset reselection
Model 2
Model 4
Model 5
Change the number of RoI to 50
Model 3

Fig 31. An illustration of changes between models

To answer the research question "How to detect portable devices with built-in cameras such as smartphones", the training images are reselected to fit the purpose of detecting smartphones. The criteria of reselection are as described in the specification part. Smartphones with camera side are included as part of the dataset. Some images in the existing Camera set are removed such as Cameras with the screen side. In total 1000 images are chosen for the new set. The ratio of number of training images and testing images is 8:2 for both categories but regarded as one class in the training process.

5.5.1 Training setting

- Dataset: 800 images from the category of Camera combined with 200 images of smartphone images from the category of Mobile Phone of Open Image Dataset (800 in the training set and 200 in the testing set)
- Learning rate: 0.0001
- Batch size: 1000
- Number of regions of interest: 4
- Image size: 600 px

5.5.2 Evaluation

After training 100 epochs, this model reaches the best result, with mAP of 0.70. The percentage of images without bounding boxes is very low, which is 2%, while there are a few more false detections compared to Model 1. The images of successfully detected cameras and smartphones can be seen in Fig 32. Most camera or smartphone objects can be well located. This model can get overfitted and have a huge drop of mAP easily. The newly included smartphone images can be a big factor causing this, since there is a big difference on the number of training images of smartphones and cameras. Smartphone images are trained at same iterations as camera images so that the smartphone detection gets overfitted earlier than camera detection. From sample images in Fig 33, a number of false detections can be seen. To avoid the overfitting of the model, in the next step, images of two categories can be training one by one. Also, more training images of smartphones can be selected. Due to the limit of smartphone image resources in the Open Image Dataset, only around 200 smartphone images are chosen. Thus, dark colored objects and box shaped objects are easy to detect. For further improvement of the model, a custom dataset of smartphone images can be a good option to reduce the rate of false positive.

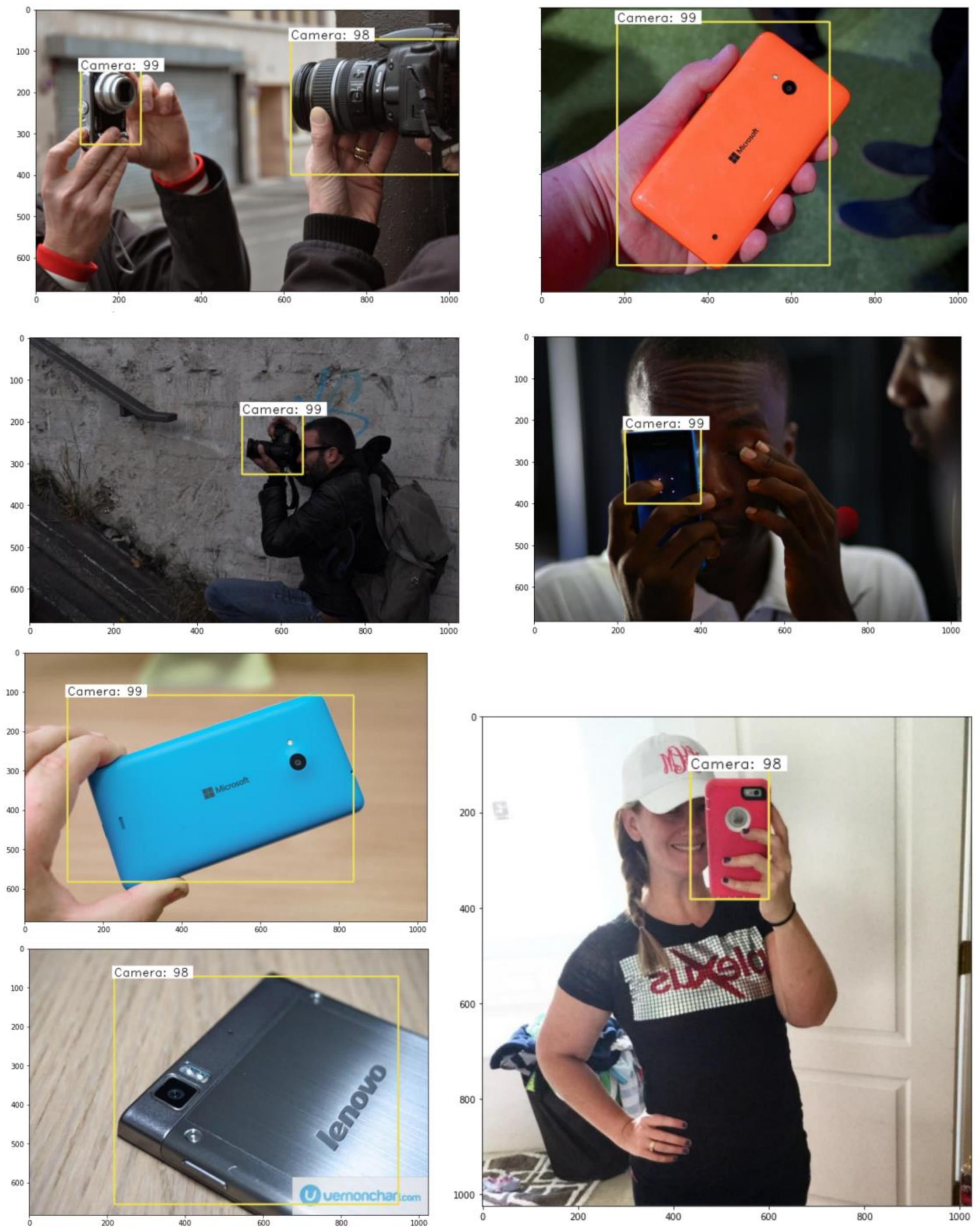


Fig 32. Sample images of successful detected camera objects and smartphones

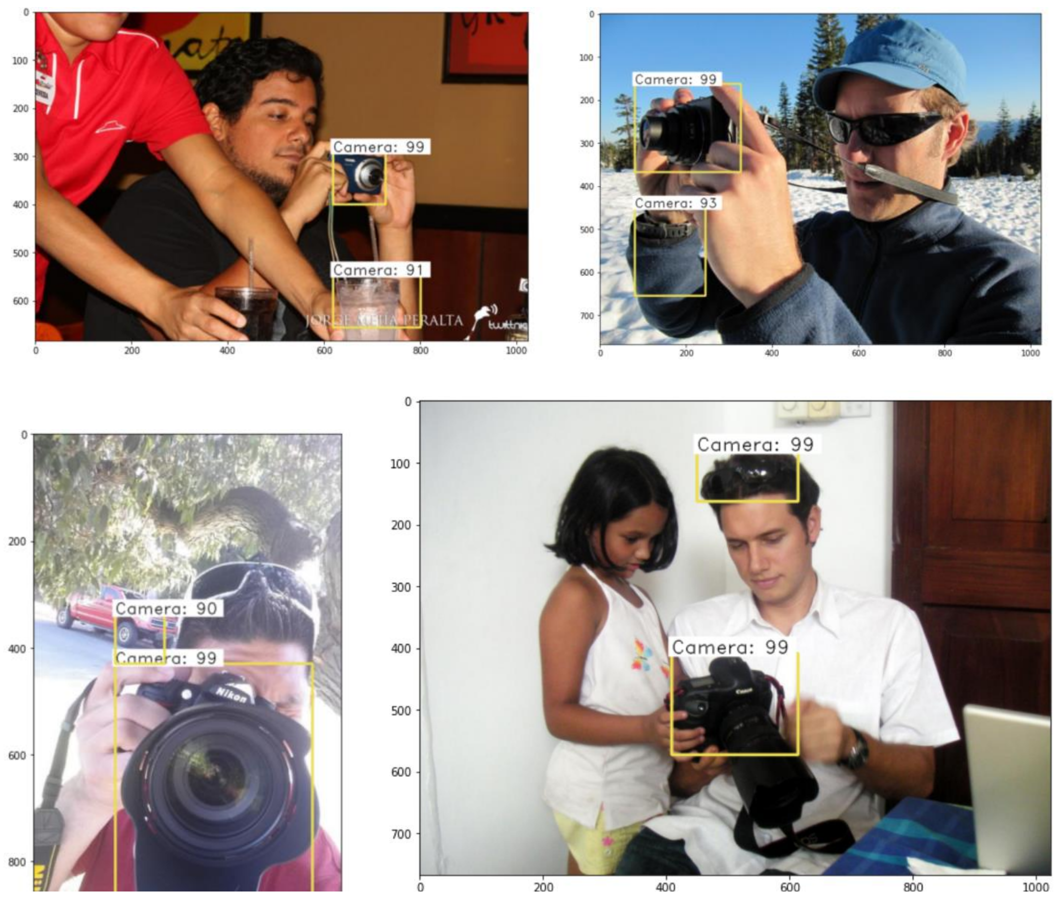


Fig 32. Sample images of false detections

5.6 Testing detection model on images of other categories

While the former experiments are done with camera and smartphone images only, it might not be able to give an overview of the performance of the detection model in daily scenes. To validate the detection models and answer the research questions, the detection model is tested on images of other categories. In total, 100 images of 10 categories of Open Image Dataset are chosen for this task. The categories are Book, Coffee cup, Handbag, Hat, Laptop, Pillow, Remote control, Suitcase, Sunglasses andTelephone. These are the objects that can be held, carried or worn and they are the common objects in daily life.

5.6.1 Testing on Model 1

First, Model 1 (using VGG16 and the old training set) is tested on this set of images. The classification threshold is set to 0.9 and the percentage of images with bounding boxes among all is 17%. Among all the testing images, two images contain lenses of camera and the lenses are detected, which can be seen in Fig 33. Falsely detected cameras are analysed and categorized with possible reasons. For example, in Fig 34, dark and reflective might be the reasons for detection, with handbags, leather shoes and sunglasses being detected. From Fig

35, A cylinder-like shape might also be a reason for the wrong detection. In this figure, coffee cups are detected as camera objects. In the training dataset, camera parts such as lenses are also treated as camera objects so that it is reasonable some coffee cups are detected for the similar shape to lenses. Wheels can also be falsely detected for the similar feature to lenses as shown in Fig 36.

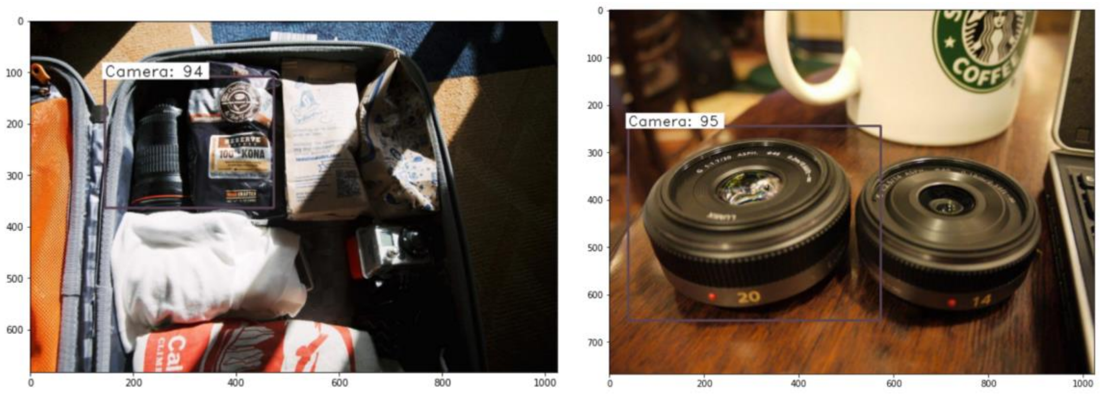


Fig 33. Detected lenses

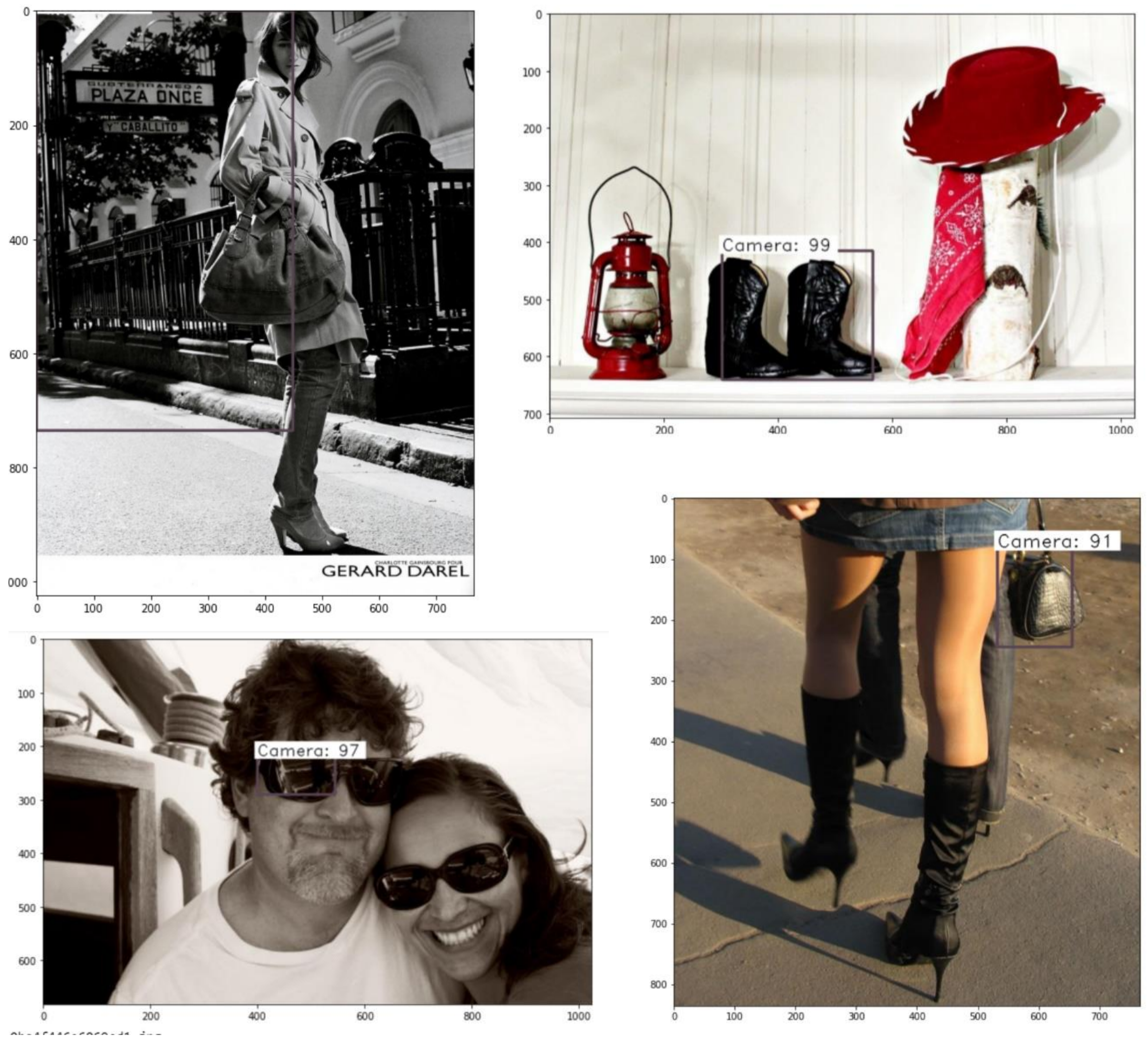


Fig 34. A collection of false detections

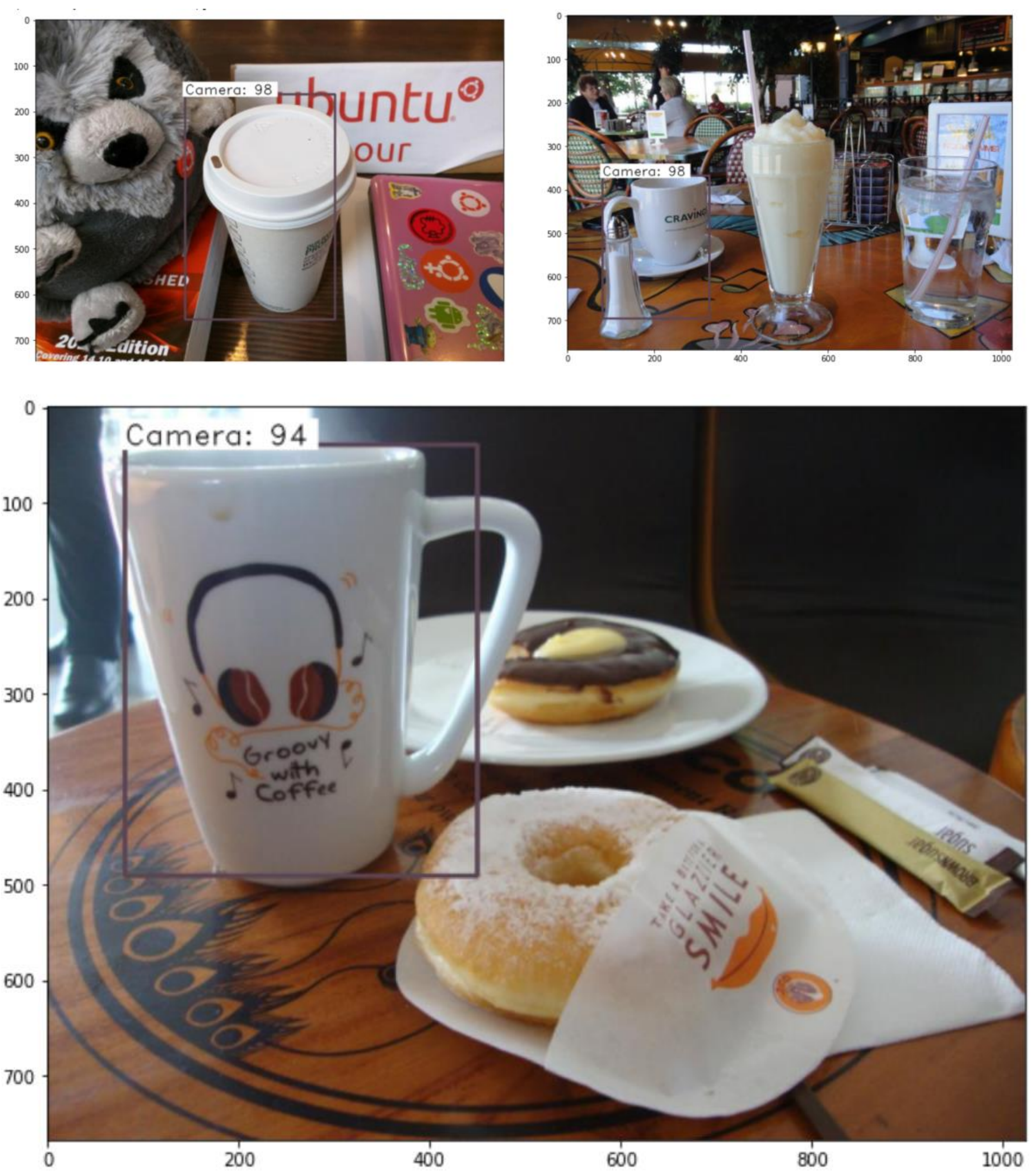
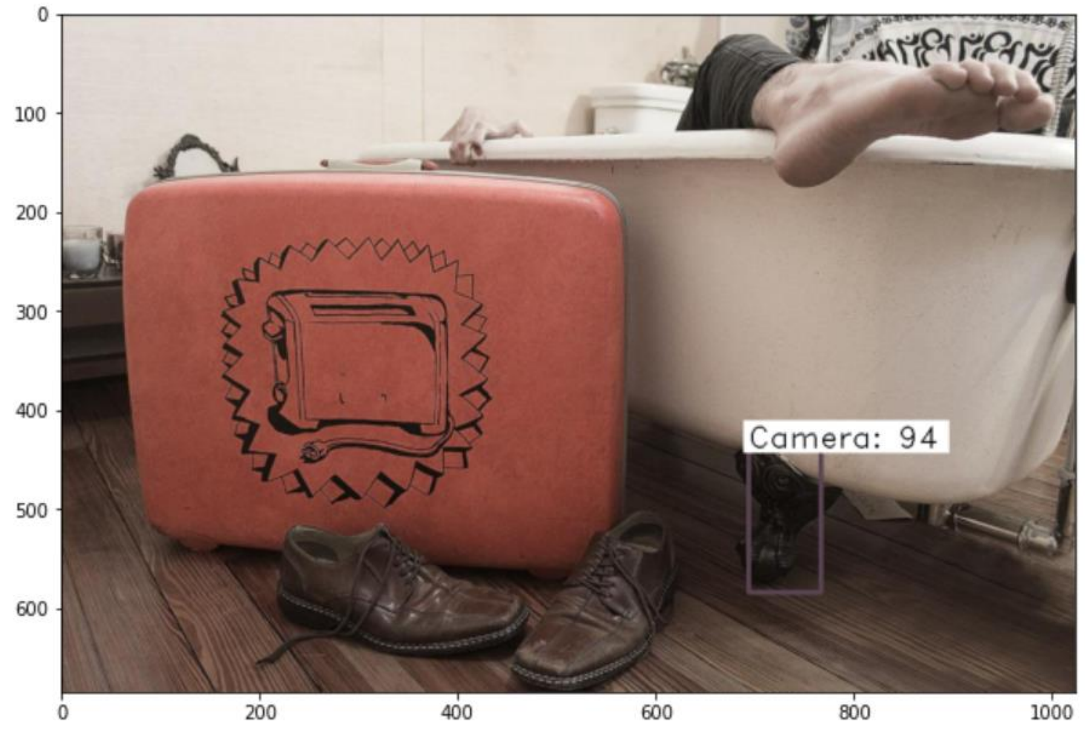


Fig 35. A collection of false detections



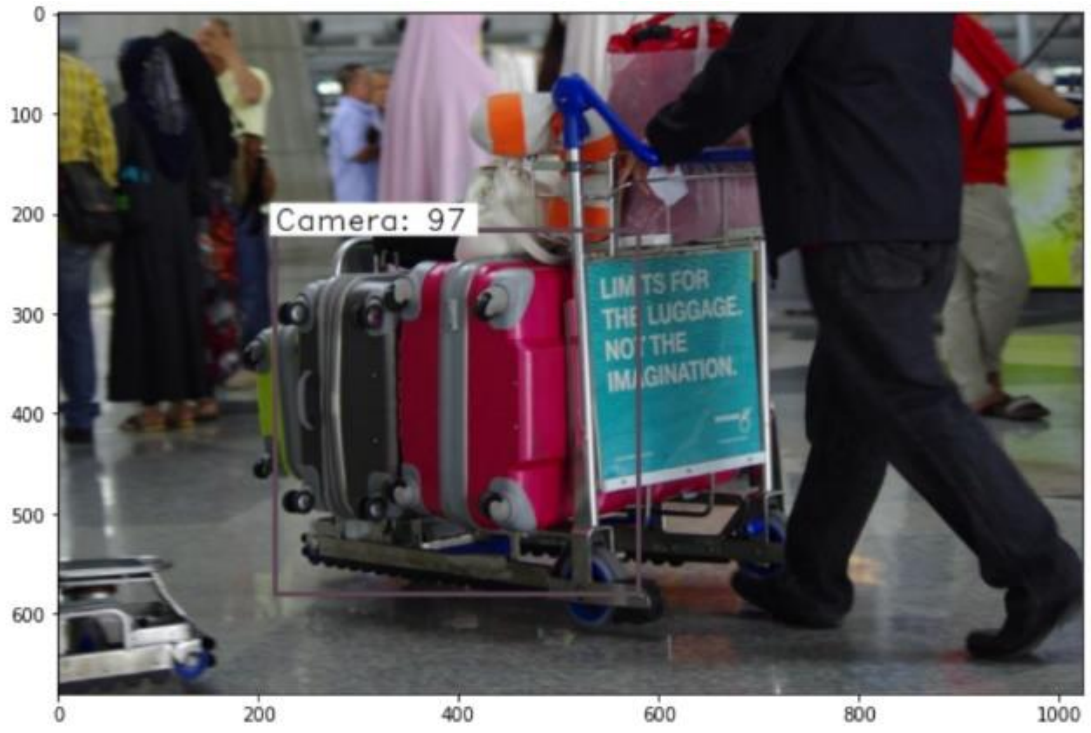


Fig 36. A collection of false detections

5.6.2 Testing on Model 5

Model 1 (using VGG16 and the new training set) is tested on this set of images. The classification threshold is set to 0.9 and the percentage of images with bounding boxes among all is 49%. Among all the testing images, two images contain lenses of camera and the lenses are detected as for Model 1. The problems of false detection in Model 1 are also applicable to this model, with similar outputs of bounding boxes of images in section 5.6.2. This model struggles a lot with sunglasses and many of them are detected as cameras, which can be seen in Fig 37. Box shape objects such as suitcases, remote controls and laptop screens are also detected. Some images are shown in Fig 38.

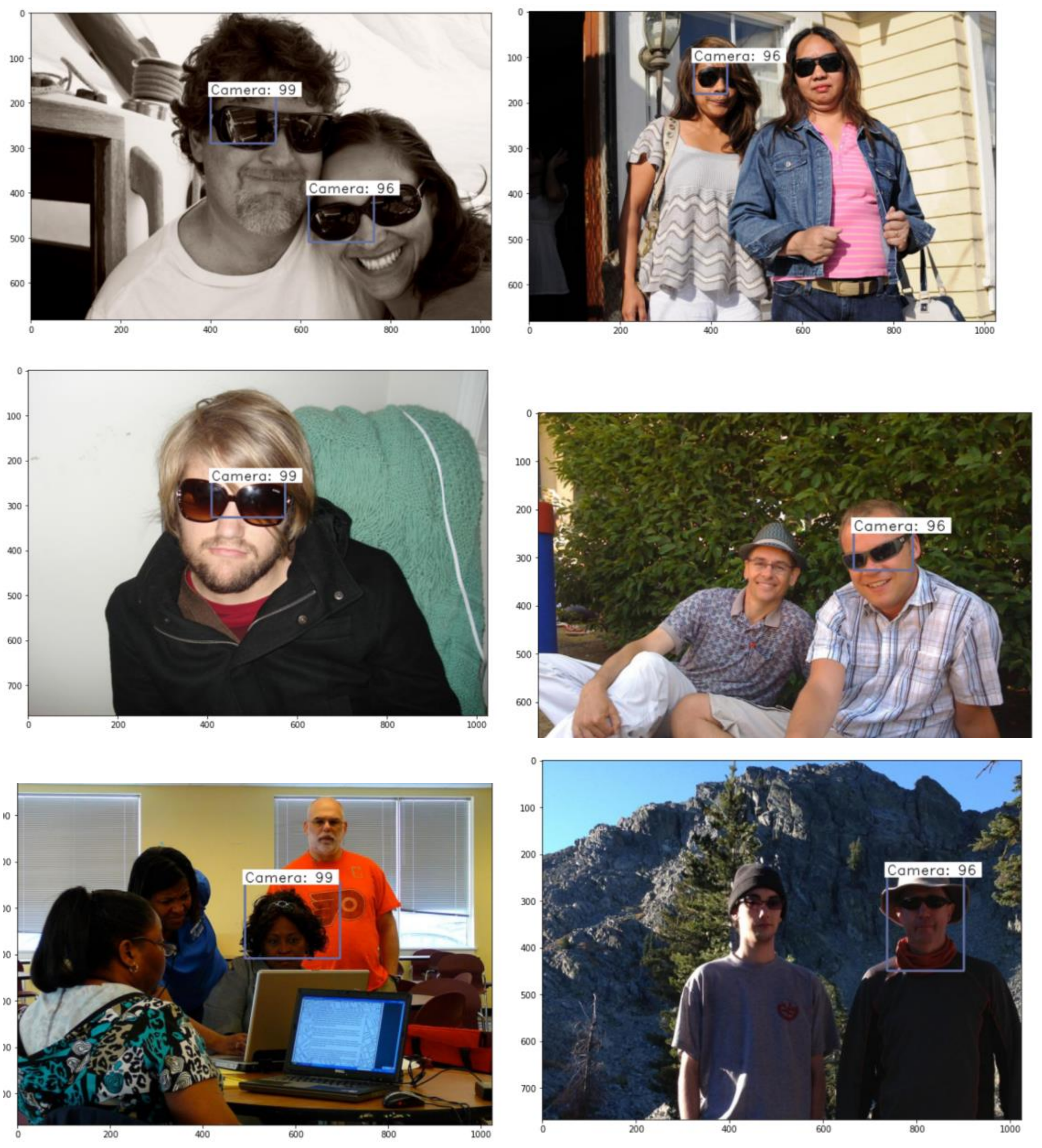


Fig 37. A collection of false detections

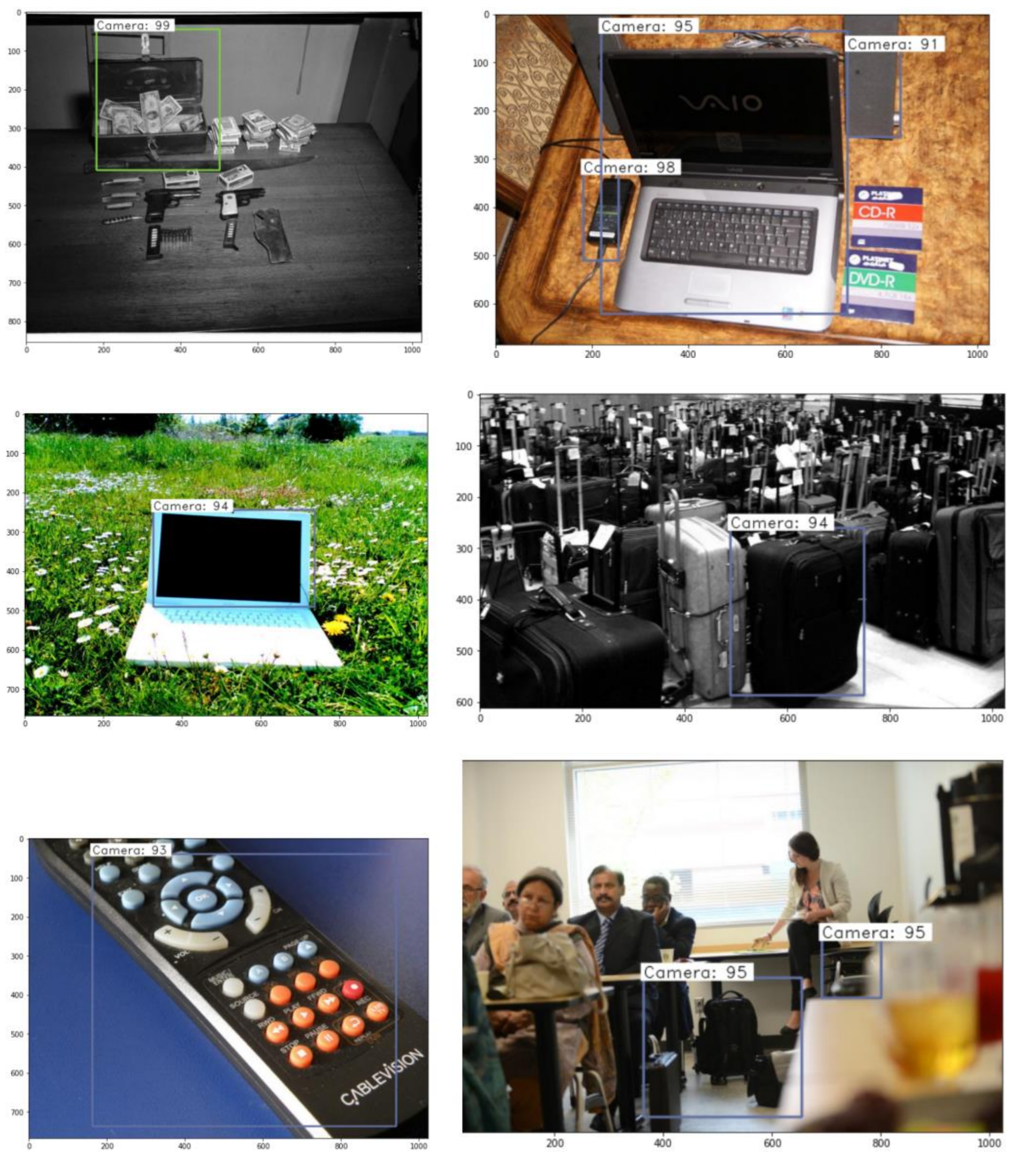


Fig 38. A collection of false detections

5.6.3 Reflection

These two models give an insight into the performance of detection in daily life. With a more proper way to train images of camera objects and smartphone objects, Model 5 may have a lower false positive rate. The problem of false detections with similar features should be analysed and solved properly in order to have accurate detection. Camera parts in training is a big issue. Training images with a whole object might help to avoid this. The detection of smartphone objects might also be an issue since many box shaped objects will have the potential to be detected.

5.7 Conclusion

Overall, the experiments of the models show the decent performance of the detection models in detecting camera objects with VGG16 network as feature extractor. The small object task and multiple object task can be handled well. The bounding boxes can also fit the objects well. A combined detection model with choices made among the tested models has the potential to achieve better performance. In real-time detection, accuracy will not be the only criterion, while knowledge from feature extractors selection, training dataset selection and training settings changes etc. will still be useful in the future detection system.

Chapter 6 Ethical preview

This part covers the potential ethical issues regarding the detection system, including technological risks and followed privacy issues and trust issues. Case analysis is involved in this process together with possible solutions.

6.1 Recording

Since the detection highly requires the use of cameras and sensors, the image processing of the environment is inevitable. Although detect-and-prevent might be an efficient active approach to protect the user's own facial image, it can risk violating the privacy of others. Therefore, the real-time visual information must be dealt with properly. With the failure of Google glass as an example, recording and displaying are allowed. Google glass users are able to start recording a video by voice control and see a live view of this recording being displayed. This drives much concern about privacy and even against laws in some countries or regions. Hololens, a mixed reality glass product developed by Microsoft also allows the recording function, while Hololens focuses more on business-related functionality and gaming. Therefore, there is a little chance that Hololens is used in public places. This helps Hololens receive less criticism than Google glass concerning privacy. Also, this shows the possibility of using image recognition in specific scenarios. For the final wearable this project is aimed for, it might still be challenging to use image recognition with cameras in daily scenes, while the detection only requires real-time image processing and there might be fewer concerns about privacy without the function of recording and displaying.

6.2 Video information storage and access

Visual information processing comes with data storage. The stored data also brings privacy concerns on the possibility of data being shared. For example, a person might be able to upload the stored data to the servers if the data is accessible. And a company might be able to access the stored data from the product it designed. These are the potential risks of making data accessible. This can cause trust issues on users and companies from other people as well.

To reduce the privacy concerns, there should be proper ways to deal with the data stored. As an example, Hololens makes developers interact with the system using a specific set of intermediate commands so that the stored data can be kept remotely. This can be meaningful when it comes to the concern of sharing. Since stored visual information of augmented reality might be useful to look back, storage can be important. However, for the purpose of detection only, the data of processed visual information might not be necessary. Therefore, storing data temporally and discarding it afterwards might also be a solution. Video data encoding or blurring human faces in stored video data can be a possible solution as well.

6.3 System security

Having a secure system is also important to reduce privacy concerns. If a system can be easily hacked, the system may face the problem of being abused on other purposes. This will threaten the privacy of other people. Also, this can also indicate the reliability of a company. Another possible consequence might be the intellectual property of a system itself. A secure system will prevent replication of the system. Therefore, it is necessary to build a secure system and have timely updates.

Chapter 7 Future work

7.1 Trade-off between accuracy and testing time

Since this project is only focusing on the performance of the detection model, time is not an issue. However, in real-time detection, the speed of detection becomes important. In this project, with the average elapsed time per image under 1 second in the testing process, the detection models prove to be still useful in about 1 frame per second in real-time detection. To have a quick reaction to the action of photographing or filming, a fast detection is needed, so that 1 frame per second may not be enough for this purpose. When making real-time detection, a trade-off between accuracy and testing time must be considered so that the detection model can have a good balance between accuracy and time.

7.2 Change feature extractor

Based on the existing structure of the detector, for the purpose of real-time detection, some more efficient feature extractors can be used such as MobileNet. MobileNet is a depth-wise separable convolutional neural network for mobile vision applications [20] It has a smaller size and fewer layers compared to VGG16. With a more efficient feature extractor, Faster R-CNN network should be able to have better trade-off between accuracy and time.

7.3 One stage detector

For real-time detection, one stage detectors such as SSD and YOLO have an advantage of testing speed. YOLO is one of the advanced one stage detectors with excellent high speed in detection. Without having to achieve very high speed, it can ensure decent accuracy as well. Therefore, together with an efficient convolution neural network, one stage detectors can be a good option for the future work.

7.4 More training images

Although data augmentation is turned on during the training process, more training data will still be a key to ensure better performance. In the testing phase, objects that have similar features result in false positives. This directly affects the detection system in daily scene detection. More training data means that there will be more resources to learn. Especially in the detection of smartphones, all smartphones have a box shape with small variations, while box shape is a very common shape of objects in daily life. Therefore, the intended detection class of the detection model needs to be refined in order to prevent false detection

7.5 Extend object detection to activity detection

A human object activity consists of complicated actions, hand gestures and interaction between objects and humans. To ensure better performance of the detection model in video sequences, action detection will be a challenging task to deal with. Time and space features might need to be extracted and analyzed in a different way to detect actions. This also requires deeper knowledge and skills than object detection. Activity detection will have an advantage of context reference. The action can be the contextual information of object and object can be the contextual information of action. This will ensure a good understanding of human object activity.

Reference

[1] T. Yamada, “Privacy Visor: Method for Preventing Face Image Detection by Using Differences in Human and Device Sensitivity,” in *Conference on Communications and Multimedia Security*, Magdeburg, CMS, 2013, pp.152-161.
[2] J. Deng, W. Dong, R. Socher, L. Li, Kai Li and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, 2009, pp. 248-255.
[3] O. Russakovsky*, J. Deng*, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg and L. Fei-Fei. (* = equal contribution) ImageNet Large Scale Visual Recognition Challenge. IJCV, 2015. http://www.image-net.org/synset?wnid=n03974915#
[4] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari. The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale. IJCV, 2020.

[5] Shorten and T. M. Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. J Big Data 6, 60 (2019). https://doi.org/10.1186/s40537-019-0197-0

[6] R. Girshick, J. Donahue, T. Darrell and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," 2014 IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, 2014, pp. 580-587, doi: 10.1109/CVPR.2014.81.

[7] R. Girshick, "Fast R-CNN," 2015 IEEE International Conference on Computer Vision (ICCV), Santiago, 2015, pp. 1440-1448.

[8] S. Ren, K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 6, pp. 1137-1149, 1 June 2017.

[9] K. He, G. Gkioxari, P. Dollár and R. Girshick, "Mask R-CNN," 2017 IEEE International Conference on Computer Vision (ICCV), Venice, 2017, pp. 2980-2988. [9]

[10] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 779-788.

[11] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, 2017, pp. 6517-6525.

[12] J. Redmon, A. Farhadi, "Yolov3: An incremental improvement" in arXiv:1804.02767, 2018, [online] Available: https://arxiv.org/abs/1804.02767.

[13] L. Liu, W. Ouyang, X. Wang et al. Deep Learning for Generic Object Detection: A Survey. Int J Comput Vis 128, 261–318 (2020). https://doi.org/10.1007/s11263-019-01247-4

[14] Z. Yao et al. "Large batch size training of neural networks with adversarial training and second-order information." ArXiv abs/1810.01021 (2018) Available: https://arxiv.org/abs/1810.01021

[15] I. Goodfellow, Y. Bengio, and A. Courville. Adaptive computation and machine learning MIT Press, (2016). ISBN:978-0-262-03561-3

[16] J. Huang et al. Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, 2017, pp. 3296-3297, doi: 10.1109/CVPR.2017.351.

[17] A. Zhang, Z. Lipton, M. Li, A. Smola. "Dive into Deep Learning" (2019)

[18] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." CoRR abs/1409.1556 (2015)

[19] K. He et al. "Deep Residual Learning for Image Recognition." 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016): 770-778.

[20] A. Howard et al. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. (2017)

[21] Sklearn.metrics.average_precision_score
.https://scikit-
learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html

[22] VGG16 – Convolutional Network for Classification and Detection.

https://neurohive.io/en/popular-networks/vgg16/

[23] Understanding and Coding a ResNet in Keras. https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33

Appendix 1 code for preprocessing

In [ ]:

```
#Code modified from https://github.com/RockyXu66/Faster_RCNN_for_Open_Images_Dataset_Keras

import cv2
from matplotlib import pyplot as plt
import numpy as np
import os
import pandas as pd
import random
from skimage import io
from shutil import copyfile
import sys
import time

import tensorflow as tf
from tensorflow.keras.preprocessing.image import load_img, img_to_array
```

In [ ]:

```
tf.__version__
```

Load data from .csv file

In [ ]:

```
images_boxable_fname = 'train-images-boxable.csv'
annotations_bbox_fname = 'oidv6-train-annotations-bbox.csv'
class_descriptions_fname = 'class-descriptions-boxable.csv'
```

In [ ]:

```
images_boxable = pd.read_csv(images_boxable_fname)
images_boxable.head()
```

In [ ]:

```
annotations_bbox = pd.read_csv(annotations_bbox_fname)
annotations_bbox.head()
```

1. **XMin, XMax, YMin, YMax**: coordinates of the box, in normalized image coordinates.
2. **IsOccluded**: Indicates that the object is occluded by another object in the image.
3. **IsTruncated**: Indicates that the object extends beyond the boundary of the image.
4. **IsGroupOf**: Indicates that the box spans a group of objects (e.g., a bed of flowers or a crowd of people). We asked annotators to use this tag for cases with more than 5 instances which are heavily occluding each other and are physically touching.
5. **IsDepiction**: Indicates that the object is a depiction (e.g., a cartoon or drawing of the object, not a real physical instance).
6. **IsInside**: Indicates a picture taken from the inside of the object (e.g., a car interior or inside of a building).

In [ ]:

```
class_descriptions = pd.read_csv(class_descriptions_fname, header=None)
class_descriptions.head()
```

Plot Bounding box

In [ ]:

```
def plot_bbox(img_id):
  img_url = images_boxable.loc[images_boxable["image_name"]==img_id + '.jpg']['image_url'].values[0
]
  img = io.imread(img_url)
  height, width, channel = img.shape
  print(f"Image: {img.shape}")
  bboxs = annotations_bbox[annotations_bbox['ImageID']==img_id]
  for index, row in bboxs.iterrows():
      xmin = row['XMin']
      xmax = row['XMax']
      ymin = row['YMin']
      ymax = row['YMax']
      xmin = int(xmin*width)
      xmax = int(xmax*width)
      ymin = int(ymin*height)
      ymax = int(ymax*height)
      label_name = row['LabelName']
      class_series = class_descriptions[class_descriptions[0]==label_name]
      class_name = class_series[1].values[0]
      print(f"Coordinates: {xmin,ymin}, {xmax,ymax}")
      cv2.rectangle(img, (xmin,ymin), (xmax,ymax), (255,0,0), 5)
      font = cv2.FONT_HERSHEY_SIMPLEX
      cv2.putText(img, class_name, (xmin,ymin-10), font, 3, (0,255,0), 5)
  plt.figure(figsize=(15,10))
  plt.title('Image with Bounding Box')
  plt.imshow(img)
  plt.axis("off")
  plt.show()
```

In [ ]:

```
least_objects _img_ids = annotations_bbox["ImageID"].value_counts().tail(50).index.values
for img_id in random.sample(list(least_objects_img_ids), 5):
  plot_bbox(img_id)
```

Get subset of the whole dataset

In [ ]:

```
class_descriptions.loc[class_descriptions[1].isin(['Camera'])]
```

In [ ]:

```
# Find the label_name for 'Camera' classes
camera_pd = class_ descriptions[class_descriptions[1]=='Camera']

label_name_camera = camera_pd[0].values[0]
```

In [ ]:

```
camera_bbox = annotations_bbox[annotations_bbox['LabelName']==label_name_camera]

print('There are %d cameras in the dataset' %(len(camera_bbox)))
```

In [ ]:

```
camera_img_id = camera_bbox['ImageID']
camera_img_id = np.unique(camera_img_id)
print('There are %d images which contain cameras' % (len(camera_img_id)))
```

In [ ]:

```
# here I've chosen 1000
images n = 1000
subcamera_img_id = random.sample(list(camera_img_id), n)

new = images_ boxable['image_name'].str.split(".", n = 1, expand =
True) subcamera_pd = images_boxable.loc[new[0].isin(subcamera_img_id)]
subcamera_pd.shape
```

In [ ]:

```
subcamera_pd.head()
```

In [ ]:

```
subcamera_dict = subcamera_pd[["image_name", "image_url"]].set_index('image_name')
["image_url"].to_dict()
```

In [ ]:

```
# download images
classes = ['Camera']
for idx, obj_type in enumerate(classes):
  n_issues = 0
  # create the directory
  if not os.path.exists(obj_type):
    os.mkdir(obj_type)
  for img_name, url in subcamera_dict.items():
    try:
      img = io.imread(url)
      saved_path = os.path.join(obj_type, img_name)
      io.imsave(saved_path, img)
    except Exception as e:
      n_issues += 1
  print(f"Images Issues: {n_issues}")
```

Dataset format for Faster-RCNN code

(fname_path, xmin, xmax, ymin, ymax, class_name)

train: 0.8 validation: 0.2

In [ ]:

```
# save images to train and test
directory train_path = 'train'
test_path = 'test'
```

In [ ]:

```
!mkdir train test
```

In [ ]:

```
random.seed(1)
```

In [ ]:

```
for i in range(len(classes)):
    all_imgs = os.listdir(classes[i])
    all_imgs = [f for f in all_imgs if not
    f.startswith('.')] random.shuffle(all_imgs)
    limit = int(n*0.8)

    train_imgs = all_imgs[:limit]
    test_imgs = all_imgs[limit:]

    # copy each classes' images to train
    directory for j in range(len(train_imgs)):
        original_path = os.path.join(classes[i],
        train_imgs[j]) new_path = os.path.join(train_path,
        train_imgs[j]) copyfile(original_path, new_path)
    # copy each classes' images to test directory
    for j in range(len(test_imgs)):
        original_path = os.path.join(classes[i], test_imgs[j])
        new_path = os.path.join(test_path, test_imgs[j])
        copyfile(original_path, new_path)
```

In [ ]:

```
label_names = [label_name_camera]

train_df = pd.DataFrame(columns=['FileName', 'XMin', 'XMax', 'YMin', 'YMax', 'ClassName'])

# Find boxes in each image and put them in a
dataframe train_imgs = os.listdir(train_path)
train_imgs = [name for name in train_imgs if not name.startswith('.')]


for i in range(len(train_imgs)):
    sys.stdout.write('Parse train_imgs ' + str(i) + '; Number of boxes: ' + str(len(train_df)) + '\
r')
    sys.stdout.flush()
    img_name = train_imgs[i]
    img_id = img_name[0:16]
    tmp _df = annotations_bbox[annotations_bbox['ImageID']==img_id]
    for index, row in tmp_df.iterrows():
        labelName = row['LabelName']
        for i in range(len(label_names)):
            if labelName == label_names[i]:
                train_df = train_df.append({'FileName': img_name,
                                            'XMin': row['XMin'],
                                            'XMax': row['XMax'],
                                            'YMin': row['YMin'],
                                            'YMax': row['YMax'],
                                            'ClassName': classes[i]},
                                           ignore_index=True)
```

In [ ]:

```
train_df.head()
```

In [ ]:

```
train_df.shape
```

In [ ]:

```
train_img_ids = train_df["FileName"].head().str.split(".").str[0].unique()
```

In [ ]:

```
for img_id in train_img_ids:
  plot_bbox(img_id)
```

In [ ]:

```
test_df = pd.DataFrame(columns=['FileName', 'XMin', 'XMax', 'YMin', 'YMax', 'ClassName'])

# find boxes in each image and put them in a
dataframe test_imgs = os.listdir(test_path)
test_imgs = [name for name in test_imgs if not name.startswith('.')]

for i in range(len(test_imgs)):
    sys.stdout.write('Parse test_imgs ' + str(i) + '; Number of boxes: ' + str(len(test_df)) + '\r'
)
    sys.stdout.flush()
    img_name = test_imgs[i]
    img_id = img_name[0:16]
    tmp _df = annotations_bbox[annotations_bbox['ImageID']==img_id]
    for index, row in tmp_df.iterrows():
        labelName = row['LabelName']
        for i in range(len(label_names)):
            if labelName == label_names[i]:
                test_df = test_df.append({'FileName': img_name,
                                            'XMin': row['XMin'],
                                            'XMax': row['XMax'],
                                            'YMin': row['YMin'],
                                            'YMax': row['YMax'],
                                      'ClassName': classes[i]},
                                     ignore_index=True)
```

In [ ]:

```
train_df.to_csv('train.csv')
test_df.to_csv('test.csv')
```

Write train.csv to annotation.txt and test.csv to test_annotation.txt

In [ ]:

```
train_df = pd.read_csv('train.csv')

# for training
with open("annotation.txt", "w+") as f:
  for idx, row in train_df.iterrows():
      img = cv2.imread('train/' + row['FileName'])
      height, width = img.shape[:2]
      x1 = int(row['XMin'] * width)
      x2 = int(row['XMax'] * width)
      y1 = int(row['YMin'] * height)
      y2 = int(row['YMax'] * height)

      google_colab_file_path = 'drive/My Drive/GP/Dataset/train/'
      fileName = os.path.join(google_colab_file_path,
      row['FileName']) className = row['ClassName']
      f.write(fileName + ',' + str(x1) + ',' + str(y1) + ',' + str(x2) + ',' + str(y2) +
',' + className + '\n')
```

In [ ]:

```
test_df = pd.read_csv('test.csv')

# for test
with open("test_annotation.txt", "w+") as f:
  for idx, row in test_df.iterrows():
      sys.stdout.write(str(idx) + '\r')
      sys.stdout.flush()
      img = cv2.imread('test/' + row['FileName'])
      height, width = img.shape[:2]
      x1 = int(row['XMin'] * width)
      x2 = int(row['XMax'] * width)
      y1 = int(row['YMin'] * height)
      y2 = int(row['YMax'] * height)

      google_colab_file_path = 'drive/My Drive/GP/Dataset/test/'
      fileName = os.path.join(google_colab_file_path,
      row['FileName']) className = row['ClassName']
      f.write(fileName + ',' + str(x1) + ',' + str(y1) + ',' + str(x2) + ',' + str(y2) +
',' + className + '\n')
```

Appendix 2 code for training (VGG)

In [ ]:

```
#Code modified from https://github.com/RockyXu66/Faster_RCNN_for_Open_Images_Dataset_Keras

from google.colab import drive
drive.mount('/content/drive')
```

Import libs

In [ ]:

```
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
```

Config setting

In [ ]:

```
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 = [64, 128, 256]

  # 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
```

Parser the data from annotation file

In [ ]:

```
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. {'Car': 2383, 'Mobile phone': 1108, 'Person': 3745}
  class_mapping: dict{key:class_name, value: idx}
   e.g. {'Car': 0, 'Mobile phone': 1, 'Person': 2}
 """
 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,
y 2, 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 background 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': in
t(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

In [ ]:

```
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.common.image_dim_ordering()
        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(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,
s elf.nb_channels))

        # 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

In [ ]:

```
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

In [ ]:

```
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 la
yer
                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='r pn_conv1')(base_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_regress')(x)
    return [x_class, x_regr, base_layers]
```

Classifier layer

In [ ]:

```
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_class_{}'.format(nb_classes))(out)
    # note: no regression target for bg class
    out _regr = TimeDistributed(Dense(4 * (nb_classes-1), activation='linear',
kernel_initializer='z ero'), name='dense_regress_{}'.format(nb_classes))(out)
    return [out_class, out_regr]
```

Calculate IoU (Intersection of Union)

In [ ]:

```
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 [ ]:

```
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
a ccording 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_anc - x1_anc))
       th = np.log((gta[bbox_num, 3] - gta[bbox_num, 2]) / (y2_anc - y1_anc))

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

       # all GT boxes should be mapped to an anchor box, so we keep 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 does 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 IOU 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 included 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_size_idx] =
     1 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 elif bbox_type == 'neutral':
      y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 elif bbox_type == 'pos':
      y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
      1 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_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[idx,2] +
n_anchr atios *
    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[idx,2] +
n_anchr atios *
    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_dx_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 negative
 # 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 [ ]:

```
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 [ ]:

```
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
a ccording 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.INTER_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,
resiz ed_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,
debug_img, num_pos
   except Exception as e:
    print(e)
    continue
```

Define loss functions for all four outputs

In [ ]:

```
lambda_rpn_regr = 1.0
lambda_rpn_class = 1.0

lambda_cls_regr = 1.0
lambda_cls_class = 1.0

epsilon = 1e-4
```

In [ ]:

```
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 neg
ative
    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 [ ]:

```
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

    # 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
large r 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]])
```

```python
        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 bbox
es
    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 [ ]:

```
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=m ax_boxes)[0]
 return result
```

Start training

In [ ]:

```
base_path = 'drive/My Drive/GP'

train_path =  'drive/My Drive/GP/Dataset/annotation.txt' # Training data (annotation file)

# 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 [ ]:

```
# 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.base_net_weights = base_weight_path
```

In [ ]:

```
#--------------------------------------------------------#
# 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) )
```

In [ ]:

```
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'.f ormat(config_output_filename))
```

In [ ]:

```
# Shuffle the images with
seed random.seed(1)
random.shuffle(train_imgs)
print('Num train samples (images) {}'.format(len(train_imgs)))
```

In [ ]:

```
# 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 [ ]:

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

In [ ]:

```
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_s tride))
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['bbo xes'][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['bbox es'][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['bbox es'][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['bbo xes'][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]+int(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.FONT_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()
```

Build the model

In [ ]:

```
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)
```

In [ ]:

```
# 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
mo dels
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
conne ct 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', 'loss_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)))
```

In [ ]:

```
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 [ ]:

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

epoch_length = 1000
num_epochs = 20
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 [ ]:

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

In [ ]:

```
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 {} previou
iterations'.format(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 settings 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.common.image_dim_ordering(),
use_regr=True, overlap_thresh=0.7, max_boxes=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 sam
les
                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=Fal
e).tolist()

                # Randomly choose (num_rois - num_pos) neg
                samples try:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois - len(selected_
os_samples), replace=False).tolist()
                except ValueError:
                  try:
                      selected_neg_samples = np.random.choice(neg_samples, C.num_rois -
len(selecte _pos_samples), replace=True).tolist()
                  except:
                    # The neg_samples is [[1 0 ]] only, therefore there's no negative
                    sample continue
                # 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 an
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.m ean(losses[:iter_num, 1])),
                                      ('final_cls', np.mean(losses[:iter_num, 2])), ('final_regr',
p.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 + l
ss_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_los
,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.')
```

In [ ]:

```
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()
```

In [ ]:

Appendix 3 code for testing (VGG)

In [ ]:

```
#Code modified from https://github.com/RockyXu66/Faster_RCNN_for_Open_Images_Dataset_Keras

from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:

```
!ls
```

Import libs

In [ ]:

```
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
```

Config setting

In [ ]:

```
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 = [64, 128, 256]

  # 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
```

Parser the data from annotation file

In [ ]:

```
def get_data(input_path):
 """Parser 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. {'Car': 2383, 'Mobile phone': 1108, 'Person': 3745}
  class_mapping: dict{key:class_name, value: idx}
   e.g. {'Car': 0, 'Mobile phone': 1, 'Person': 2}
 """
 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, 
y 2, 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 background 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': in
t(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

In [ ]:

```
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.common.image_dim_ordering()
        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(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,
s elf.nb_channels))

        # 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

In [ ]:

```
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

In [ ]:

```
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 la
yer
                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='r pn_conv1')(base_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_regress')(x)
    return [x_class, x_regr, base_layers]
```

Classifier layer

In [ ]:

```
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_class_{}'.format(nb_classes))(out)
    # note: no regression target for bg class
    out _regr = TimeDistributed(Dense(4 * (nb_classes-1), activation='linear',
kernel_initializer='z ero'), name='dense_regress_{}'.format(nb_classes))(out)
    return [out_class, out_regr]
```

Calculate IoU (Intersection of Union)

In [ ]:

```
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 [ ]:

```
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
a ccording 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_anc - x1_anc))
       th = np.log((gta[bbox_num, 3] - gta[bbox_num, 2]) / (y2_anc - y1_anc))

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

       # all GT boxes should be mapped to an anchor box, so we keep 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 does 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 IOU 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 included 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_size_idx] =
     1 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 elif bbox_type == 'neutral':
      y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 elif bbox_type == 'pos':
      y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
      1 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_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[idx,2] + n_anchr
atios *
    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[idx,2] +
n_anchr atios *
    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_dx_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 negative
 # 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 [ ]:

```
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 [ ]:

```
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
a ccording 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.INTER_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,
resiz ed_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,
debug_img, num_pos

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

In [ ]:

```
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-
pyt hon/
 # if there are no boxes, return an empty list

    # 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
large r 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
```

In [ ]:

```
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, 9) 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, 36) 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, 9) 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=m ax_boxes)[0]
 return result
```

In [ ]:

```
base_path = 'drive/My Drive/GP'

test_path = 'drive/My Drive/GP/Dataset/test_annotation.txt' # Test data (annotation file)

test_base_path = 'drive/My Drive/GP/Dataset/test' # Directory to save the test images

plt_img_path = 'drive/My Drive/GP/Dataset/plot_saved'

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

In [ ]:

```
with open(config_output_filename, 'rb') as f_in:
 C = pickle.load(f_in)

# turn off any data augmentation at test
time C.use_horizontal_flips = False
C.use_vertical_flips = False
C.rot_90 = False
```

In [ ]:

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

r_epochs = len(record_df)

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.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()
```

Test

In [ ]:

```
def format_img_size(img, C):
 """ formats the image size based on config """
 img_min_side = float(C.im_size)
 (height,width,_) = img.shape

 if width <= height:
  ratio = img_min_side/width
  new_height = int(ratio * height)
  new_width = int(img_min_side)
 else:
  ratio = img_min_side/height
  new_width = int(ratio * width)
  new_height = int(img_min_side)
 img = cv2.resize(img, (new_width, new_height),
 interpolation=cv2.INTER_CUBIC) return img, ratio
def format_img_channels(img, C):
 """ formats the image channels based on config """
 img = img[:, :, (2, 1, 0)]
 img = img.astype(np.float32)
 img[:, :, 0] -= C.img_channel_mean[0]
 img[:, :, 1] -= C.img_channel_mean[1]
 img[:, :, 2] -= C.img_channel_mean[2]
 img /= C.img_scaling_factor
 img = np.transpose(img, (2, 0, 1))
 img = np.expand_dims(img, axis=0)
 return img

def format_img(img, C):
 """ formats an image for model prediction based on config """
 img, ratio = format_img_size(img, C)
 img = format_img_channels(img, C)
 return img, ratio

# Method to transform the coordinates of the bounding box to its original
size def get_real_coordinates(ratio, x1, y1, x2, y2):

 real_x1 = int(round(x1 // ratio))
 real_y1 = int(round(y1 // ratio))
 real_x2 = int(round(x2 // ratio))
 real_y2 = int(round(y2 // ratio))

 return (real_x1, real_y1, real_x2 ,real_y2)
```

In [ ]:

```
num_features = 512

input_shape_img = (None, None, 3)
input_shape_features = (None, None, num_features)

img_input = Input(shape=input_shape_img)
roi_input = Input(shape=(C.num_rois, 4))
feature_map_input = Input(shape=input_shape_features)

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

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

classifier = classifier_layer(feature_map_input, roi_input, C.num_rois,
nb_classes=len(C.class_mapp ing))

model_rpn = Model(img_input, rpn_layers)
model_classifier_only = Model([feature_map_input, roi_input], classifier)

model_classifier = Model([feature_map_input, roi_input], classifier)

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)

model_rpn.compile(optimizer='sgd', loss='mse')
model_classifier.compile(optimizer='sgd', loss='mse')
```

In [ ]:

```
# Switch key value for class mapping
class_mapping = C.class_mapping
class_mapping = {v: k for k, v in
class_mapping.items()} print(class_mapping)
class_to_color = {class_mapping[v]: np.random.randint(0, 255, 3) for v in class_mapping}
```

In [ ]:

```
test_imgs = os.listdir(test_base_path)

imgs_path = []
for i in range(50):
 idx = np.random.randint(len(test_imgs))
 imgs_path.append(test_imgs[idx])

all_imgs = []

classes = {}
```

In [ ]:

```
# If the box classification value is less than this, we ignore this
box bbox_threshold = 0.84
for idx, img_name in enumerate(imgs_path):
    if not img_name.lower().endswith(('.bmp', '.jpeg', '.jpg', '.png', '.tif', '.tiff')):
        continue
    print(img_name)
    st = time.time()
    filepath = os.path.join(test_base_path, img_name)

    img = cv2.imread(filepath)

    X, ratio = format_img(img, C)

    X = np.transpose(X, (0, 2, 3, 1))

    # get output layer Y1, Y2 from the RPN and the feature maps F
    # Y1: y_rpn_cls
    # Y2: y_rpn_regr
    [Y1, Y2, F] = model_rpn.predict(X)

    # Get bboxes by applying NMS
    # R.shape = (300, 4)
    R = rpn_to_roi(Y1, Y2, C, K.common.image_dim_ordering(), overlap_thresh=0.7)

    # convert from (x1,y1,x2,y2) to (x,y,w,h)
    R[:, 2] -= R[:, 0]
    R[:, 3] -= R[:, 1]

    # apply the spatial pyramid pooling to the proposed
    regions bboxes = {}
    probs = {}

    for jk in range(R.shape[0]//C.num_rois + 1):
        ROIs = np.expand_dims(R[C.num_rois*jk:C.num_rois*(jk+1), :], axis=0)
        if ROIs.shape[1] == 0:
            break

        if jk == R.shape[0]//C.num_rois:
            #pad R
            curr_shape = ROIs.shape
            target_shape = (curr_shape[0],C.num_rois,curr_shape[2])
            ROIs_padded = np.zeros(target_shape).astype(ROIs.dtype)
            ROIs_padded[:, :curr_shape[1], :] = ROIs
            ROIs_padded[0, curr_shape[1]:, :] = ROIs[0, 0, :]
            ROIs = ROIs_padded

        [P_cls, P_regr] = model_classifier_only.predict([F, ROIs])

        # Calculate bboxes coordinates on resized
        image for ii in range(P_cls.shape[1]):
            # Ignore 'bg' class
         if np.max(P_cls[0, ii, :]) < bbox_threshold or np.argmax(P_cls[0, ii, :]) == (P_cls.shap
e[2] - 1):
                continue

            cls_name = class_mapping[np.argmax(P_cls[0, ii, :])]

            if cls_name not in bboxes:
                bboxes[cls_name] = []
                probs[cls_name] = []

            (x, y, w, h) = ROIs[0, ii, :]

            cls_num = np.argmax(P_cls[0, ii, :])
            try:
                (tx, ty, tw, th) = P_regr[0, ii,
                4*cls_num:4*(cls_num+1)] tx /= C.classifier_regr_std[0]
                ty /= C.classifier_regr_std[1]
                tw /= C.classifier_regr_std[2]
                th /= C.classifier_regr_std[3]
                x, y, w, h = apply_regr(x, y, w, h, tx, ty, tw, th)
            except:
                pass
            bboxes[cls_name].append([C.rpn_stride*x, C.rpn_stride*y, C.rpn_stride*(x+w), C.rpn_strid
e*(y+h)])
                probs[cls_name].append(np.max(P_cls[0, ii, :]))

    all_dets = []

    for key in bboxes:
        bbox = np.array(bboxes[key])

        new_boxes, new_probs = non_max_suppression_fast(bbox, np.array(probs[key]), overlap_thresh=
0.2)
        for jk in range(new_boxes.shape[0]):
            (x1, y1, x2, y2) = new_boxes[jk,:]

            # Calculate real coordinates on original image
            (real_x1, real_y1, real_x2, real_y2) = get_real_coordinates(ratio, x1, y1, x2, y2)

            cv2.rectangle(img,(real_x1, real_y1), (real_x2, real_y2),
(int(class_to_color[key][0]), int(class_to_color[key][1]), int(class_to_color[key][2])),4)
            textLabel = '{}: {}'.format(key,int(100*new_probs[jk]))
            all_dets.append((key,100*new_probs[jk]))

            (retval,baseLine) = cv2.getTextSize(textLabel,cv2.FONT_HERSHEY_COMPLEX,1,1)
            textOrg = (real_x1, real_y1-0)

            cv2.rectangle(img, (textOrg[0] - 5, textOrg[1]+baseLine - 5), (textOrg[0]+retval[0] +
5, textOrg[1]-retval[1] - 5), (0, 0, 0), 1)
            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, 1, (0, 0, 0), 1)

    print('Elapsed time = {}'.format(time.time() - st))
    print(all_dets)
    plt.figure(figsize=(10,10))
    #plt.grid()
    plt.imshow(cv2.cvtColor(img,cv2.COLOR_BGR2RGB))
    #plt.savefig(os.path.join(plt_img_path, img_name))
    plt.show()
```

Measure mAP

In [ ]:

```
def get_map(pred, gt, f):
 T={}
 P={}
 fx, fy = f

 for bbox in gt:
  bbox['bbox_matched'] = False

 pred_probs = np.array([s['prob'] for s in pred])
 box_idx_sorted_by_prob = np.argsort(pred_probs)[::-1]

 for box_idx in box_idx_sorted_by_prob:
  pred_box = pred[box_idx]
  pred_class = pred_box['class']
  pred_x1 = pred_box['x1']
  pred_x2 = pred_box['x2']
  pred_y1 = pred_box['y1']
  pred_y2 = pred_box['y2']
  pred_prob = pred_box['prob']
  if pred_class not in P:
   P[pred_class] = []
   T[pred_class] = []
  P[pred_class].append(pred_prob)
  found_match = False

  for gt_box in gt:
   gt_class = gt_box['class']
   gt_x1 = gt_box['x1']/fx
   gt_x2 = gt_box['x2']/fx
   gt_y1 = gt_box['y1']/fy
```

```
    gt_y2 = gt_box['y2']/fy
    gt_seen = gt_box['bbox_matched']
    if gt_class != pred_class:
      continue
    if gt_seen:
      continue
    iou_map = iou((pred_x1, pred_y1, pred_x2, pred_y2), (gt_x1, gt_y1, gt_x2, gt_y2))
    if iou_map >= 0.5:
      found_match = True
      gt_box['bbox_matched'] = True
      break
    else:
      continue

  T[pred_class].append(int(found_match))

for gt_box in gt:
  if not gt_box['bbox_matched']:# and not gt_box['difficult']:
    if gt_box['class'] not in P:
      P[gt_box['class']] = []
      T[gt_box['class']] = []

    T[gt_box['class']].append(1)
    P[gt_box['class']].append(0)

#import pdb
#pdb.set_trace()
return T, P
```

In [ ]:

```
def format_img_map(img, C):
  """Format image for mAP. Resize original image to C.im_size (300 in here)

  Args:
    img: cv2 image
    C: config

  Returns:
    img: Scaled and normalized image with expanding dimension
    fx: ratio for width scaling
    fy: ratio for height scaling
  """

  img_min_side = float(C.im_size)
  (height,width,_) = img.shape

  if width <= height:
    f = img_min_side/width
    new_height = int(f * height)
    new_width = int(img_min_side)
  else:
    f = img_min_side/height
    new_width = int(f * width)
    new_height = int(img_min_side)
  fx = width/float(new_width)
  fy = height/float(new_height)
  img = cv2.resize(img, (new_width, new_height), interpolation=cv2.INTER_CUBIC)
  # Change image channel from BGR to
  RGB img = img[:, :, (2, 1, 0)]
  img = img.astype(np.float32)
  img[:, :, 0] -= C.img_channel_mean[0]
  img[:, :, 1] -= C.img_channel_mean[1]
  img[:, :, 2] -= C.img_channel_mean[2]
  img /= C.img_scaling_factor
  # Change img shape from (height, width, channel) to (channel, height, width)
  img = np.transpose(img, (2, 0, 1))
  # Expand one dimension at axis 0
  # img shape becames (1, channel, height, width)
  img = np.expand_dims(img, axis=0)
  return img, fx, fy
```

In [ ]:

```
print(class_mapping)
```

In [ ]:

```
# This might takes a while to parser the
data test_imgs, _, _ = get_data(test_path)
```

In [ ]:

```
T={}
P={}
mAPs = []
total_time = []
FP_num = 0
for idx, img_data in enumerate(test_imgs):
    print('{}/{}'.format(idx,len(test_imgs)))
    st = time.time()
    filepath = img_data['filepath']

    img = cv2.imread(filepath)

    X, fx, fy = format_img_map(img, C)

    # Change X (img) shape from (1, channel, height, width) to (1, height, width,
    channel) X = np.transpose(X, (0, 2, 3, 1))

    # get the feature maps and output from the RPN
    [Y1, Y2, F] = model_rpn.predict(X)


    R = rpn_to_roi(Y1, Y2, C, K.common.image_dim_ordering(), overlap_thresh=0.7)

    # convert from (x1,y1,x2,y2) to (x,y,w,h)
    R[:, 2] -= R[:, 0]
    R[:, 3] -= R[:, 1]

    # apply the spatial pyramid pooling to the proposed
    regions bboxes = {}
    probs = {}

    for jk in range(R.shape[0] // C.num_rois + 1):
        ROIs = np.expand_dims(R[C.num_rois * jk:C.num_rois * (jk + 1), :], axis=0)
        if ROIs.shape[1] == 0:
            break

        if jk == R.shape[0] // C.num_rois:
            # pad R
            curr_shape = ROIs.shape
            target_shape = (curr_shape[0], C.num_rois, curr_shape[2])
            ROIs_padded = np.zeros(target_shape).astype(ROIs.dtype)
            ROIs_padded[:, :curr_shape[1], :] = ROIs
            ROIs_padded[0, curr_shape[1]:, :] = ROIs[0, 0, :]
            ROIs = ROIs_padded

        [P_cls, P_regr] = model_classifier_only.predict([F, ROIs])

        # Calculate all classes' bboxes coordinates on resized image (300, 400)
        # Drop 'bg' classes bboxes
        for ii in range(P_cls.shape[1]):

            # If class name is 'bg', continue
            if np.argmax(P_cls[0, ii, :]) == (P_cls.shape[2] - 1):
                continue

            # Get class name
            cls_name = class_mapping[np.argmax(P_cls[0, ii, :])]

            if cls_name not in bboxes:

                bboxes[cls_name] = []
                probs[cls_name] = []

            (x, y, w, h) = ROIs[0, ii, :]

            cls_num = np.argmax(P_cls[0, ii, :])
            try:
                (tx, ty, tw, th) = P_regr[0, ii, 4 * cls_num:4 * (cls_num + 1)]
                tx /= C.classifier_regr_std[0]
                ty /= C.classifier_regr_std[1]
                tw /= C.classifier_regr_std[2]
                th /= C.classifier_regr_std[3]
                x, y, w, h = roi_helpers.apply_regr(x, y, w, h, tx, ty, tw, th)
            except:
                pass
            bboxes[cls_name].append([16 * x, 16 * y, 16 * (x + w), 16 * (y + h)])
            probs[cls_name].append(np.max(P_cls[0, ii, :]))

    all_dets = []

    for key in bboxes:
        bbox = np.array(bboxes[key])

        # Apply non-max-suppression on final bboxes to get the output bounding boxe
        new_boxes, new_probs = non_max_suppression_fast(bbox, np.array(probs[key]), overlap_thresh=
0.5)
        for jk in range(new_boxes.shape[0]):
            (x1, y1, x2, y2) = new_boxes[jk, :]
            det = {'x1': x1, 'x2': x2, 'y1': y1, 'y2': y2, 'class': key, 'prob':

            new_probs[jk]} all_dets.append(det)


    print('Elapsed time = {}'.format(time.time() -
    st)) elapsed_time = time.time() - st
    total_time.append(elapsed_time)
    t, p = get_map(all_dets, img_data['bboxes'], (fx, fy))
    for key in t.keys():
        if key not in T:
            T[key] = []
            P[key] = []
        T[key].extend(t[key])
        P[key].extend(p[key])
    all_aps = []

    for key in T.keys():
        ap = average_precision_score(T[key], P[key])
        print('{} AP: {}'.format(key, ap))
        all_aps.append(ap)
    print('mAP = {}'.format(np.mean(np.array(all_aps))))
    mAPs.append(np.mean(np.array(all_aps)))
    #print(T)
    #print(P)
    if not all_dets:
      FP_num += 1
print()
print('mean average precision:', np.mean(np.array(mAPs)))
```

In [ ]:

```
mAP = [mAP for mAP in mAPs if str(mAP)!='nan']
mean_average_prec = round(np.mean(np.array(mAP)), 3)
print('After training %dk batches, the mean average precision
is %0.3f'%(len(record_df), mean_average_prec))
print('Average elapsed time:', np.average(np.array(total_time)))
print('Percentage of images that without bounding boxes among all:', FP_num/len(test_imgs))

# record_df.loc[len(record_df)-1, 'mAP'] = mean_average_prec
# record_df.to_csv(C.record_path, index=0)
# print('Save mAP to {}'.format(C.record_path))
```

In [ ]:

```
```

Appendix 4 code for training (ResNet50)

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

Import libs

In [ ]:

```
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, Add, Input, Activation, Conv2D, MaxPooling2D,
AveragePooli ng2D, Dropout, ZeroPadding2D, BatchNormalization
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.initializers import glorot_uniform
from keras.models import Model
from keras.utils import generic_utils
from keras.engine import Layer, InputSpec
from keras import initializers, regularizers
```

Config setting

In [ ]:

```
class Config:

 def __init__(self):

   # Print the process or

   not self.verbose = True

   # Name of base network

   self.network = 'resnet'

   # 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 = [64, 128, 256]

  # 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
```

Parser the data from annotation file

In [ ]:

```
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. {'Car': 2383, 'Mobile phone': 1108, 'Person': 3745}
  class_mapping: dict{key:class_name, value: idx}
   e.g. {'Car': 0, 'Mobile phone': 1, 'Person': 2}
 """
 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,
y 2, 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 background 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': in
t(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

In [ ]:

```
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.common.image_dim_ordering()
        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(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,
s elf.nb_channels))

        # 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()))
```

ResNet-50 model

In [ ]:

```
def get_img_output_length(width, height):
    # return get_output_length(width), get_output_length(height)
    def get_output_length(input_length):
        # zero_pad
        input_length += 6
        # apply 4 strided
        convolutions filter_sizes =
        [7, 3, 1, 1] stride = 2
        for filter_size in filter_sizes:
            input_length = (input_length - filter_size + stride) //
        stride return input_length
    return get_output_length(width), get_output_length(height)
```

In [ ]:

```
def identity_block(input_tensor, kernel_size, filters, stage, block, trainable=True):

    nb_filter1, nb_filter2, nb_filter3 = filters
    bn_axis = 3


    conv_name_base = 'res' + str(stage) + block + '_branch'

    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = Conv2D(nb_filter1, (1, 1), name=conv_name_base + '2a', trainable=trainable)(input_tensor)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2a')(x)
    x = Activation('relu')(x)

    x = Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same', name=conv_name_base +
'2b', trainable=trainable)(x)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2b')(x)
    x = Activation('relu')(x)

    x = Conv2D(nb_filter3, (1, 1), name=conv_name_base + '2c', trainable=trainable)(x)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2c')(x)

    x = Add()([x, input_tensor])
    x = Activation('relu')(x)
    return x


def identity_block_td(input_tensor, kernel_size, filters, stage, block, trainable=True):

    # identity block time distributed

    nb_filter1, nb_filter2, nb_filter3 = filters
    bn_axis = 3

    conv_name_base = 'res' + str(stage) + block + '_branch'

    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = TimeDistributed(Conv2D(nb_filter1, (1, 1), trainable=trainable,
kernel_initializer='normal' ), name=conv_name_base + '2a')(input_tensor)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2a')(x)
    x = Activation('relu')(x)

    x = TimeDistributed(Conv2D(nb_filter2, (kernel_size, kernel_size), trainable=trainable,
kernel_ initializer='normal',padding='same'), name=conv_name_base + '2b')(x)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2b')(x)
    x = Activation('relu')(x)

    x = TimeDistributed(Conv2D(nb_filter3, (1, 1), trainable=trainable,
kernel_initializer='normal' ), name=conv_name_base + '2c')(x)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2c')(x)

    x = Add()([x, input_tensor])
    x = Activation('relu')(x)

    return x
```

```python
def conv_block(input_tensor, kernel_size, filters, stage, block, strides=(2, 2), trainable=True):

    nb_filter1, nb_filter2, nb_filter3 = filters
    bn_axis = 3

    conv_name_base = 'res' + str(stage) + block + '_branch'

    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = Conv2D(nb_filter1, (1, 1), strides=strides, name=conv_name_base + '2a',
trainable=trainable )(input_tensor)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2a')(x)
    x = Activation('relu')(x)

    x = Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same', name=conv_name_base +
'2b', trainable=trainable)(x)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2b')(x)
    x = Activation('relu')(x)

    x = Conv2D(nb_filter3, (1, 1), name=conv_name_base + '2c', trainable=trainable)(x)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2c')(x)

    shortcut = Conv2D(nb_filter3, (1, 1), strides=strides, name=conv_name_base + '1',
trainable=tra inable)(input_tensor)
    shortcut = BatchNormalization(axis=bn_axis, name=bn_name_base + '1')(shortcut)

    x = Add()([x, shortcut])
    x = Activation('relu')(x)
    return x


def conv_block_td(input_tensor, kernel_size, filters, stage, block, input_shape, strides=(2, 2),
tr ainable=True):
    # conv block time distributed

    nb_filter1, nb_filter2, nb_filter3 = filters
    bn_axis = 3


    conv_name_base = 'res' + str(stage) + block + '_branch'

    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = TimeDistributed(Conv2D(nb_filter1, (1, 1), strides=strides, trainable=trainable,
kernel_ini tializer='normal'), input_shape=input_shape, name=conv_name_base + '2a')(input_tensor)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2a')(x)
    x = Activation('relu')(x)

    x = TimeDistributed(Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same',
trainable=tr ainable, kernel_initializer='normal'), name=conv_name_base + '2b')(x)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2b')(x)
    x = Activation('relu')(x)

    x = TimeDistributed(Conv2D(nb_filter3, (1, 1), kernel_initializer='normal'),
name=conv_name_base + '2c', trainable=trainable)(x)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2c')(x)

    shortcut = TimeDistributed(Conv2D(nb_filter3, (1, 1), strides=strides, trainable=trainable,
ker nel_initializer='normal'), name=conv_name_base + '1')(input_tensor)
    shortcut = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '1')(shortcut)

    x = Add()([x, shortcut])
    x = Activation('relu')(x)
    return x
```

In [ ]:

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

    # Determine proper input shape

    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

    x = ZeroPadding2D((3, 3))(img_input)

    x = Conv2D(64, (7, 7), strides=(2, 2), name='conv1', trainable = trainable)(x)
    x = BatchNormalization(axis=bn_axis, name='bn_conv1')(x)
    x = Activation('relu')(x)
    x = MaxPooling2D((3, 3), strides=(2, 2))(x)

    x = conv_block(x, 3, [64, 64, 256], stage=2, block='a', strides=(1, 1), trainable = trainable)
    x = identity_block(x, 3, [64, 64, 256], stage=2, block='b', trainable = trainable)
    x = identity_block(x, 3, [64, 64, 256], stage=2, block='c', trainable = trainable)

    x = conv_block(x, 3, [128, 128, 512], stage=3, block='a', trainable = trainable)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='b', trainable = trainable)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='c', trainable = trainable)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='d', trainable = trainable)

    x = conv_block(x, 3, [256, 256, 1024], stage=4, block='a', trainable = trainable)
    x = identity_block(x, 3, [256, 256, 1024], stage=4, block='b', trainable = trainable)
    x = identity_block(x, 3, [256, 256, 1024], stage=4, block='c', trainable = trainable)
    x = identity_block(x, 3, [256, 256, 1024], stage=4, block='d', trainable = trainable)
    x = identity_block(x, 3, [256, 256, 1024], stage=4, block='e', trainable = trainable)
    x = identity_block(x, 3, [256, 256, 1024], stage=4, block='f', trainable = trainable)

    return x
```

RPN layer

In [ ]:

```
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 la
yer
                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: resnet 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: resnet in here
    """
    x = Conv2D(512, (3, 3), padding='same', activation='relu', kernel_initializer='normal',
name='r pn_conv1')(base_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_regress')(x)

    return [x_class, x_regr, base_layers]
```

Classifier layer

In [ ]:

```
def classifier_layers(x, input_shape, trainable=False):

    # compile times on theano tend to be very high, so we use smaller ROI pooling regions
to workaround
```

```
    # (hence a smaller stride in the region that follows the ROI pool)
    x = conv_block_td(x, 3, [512, 512, 2048], stage=5, block='a', input_shape=input_shape,
strides=( 2, 2), trainable=trainable)
    x = identity_block_td(x, 3, [512, 512, 2048], stage=5, block='b', trainable=trainable)
    x = identity_block_td(x, 3, [512, 512, 2048], stage=5, block='c', trainable=trainable)
    x = TimeDistributed(AveragePooling2D((7, 7)), name='avg_pool')(x)

    return x
```

In [ ]:

```
def classifier(base_layers, input_rois, num_rois, nb_classes = 21, trainable=False):

    # compile times on theano tend to be very high, so we use smaller ROI pooling regions
to workaround

    pooling_regions = 14
    input_shape = (num_rois,14,14,1024)

    out_roi_pool = RoiPoolingConv(pooling_regions, num_rois)([base_layers, input_rois])
    out = classifier_layers(out_roi_pool, input_shape=input_shape, trainable=True)

    out = TimeDistributed(Flatten())(out)

    out_class = TimeDistributed(Dense(nb_classes, activation='softmax',
kernel_initializer='zero'), name='dense_class_{}'.format(nb_classes))(out)
    # note: no regression target for bg class
    out _regr = TimeDistributed(Dense(4 * (nb_classes-1), activation='linear',
kernel_initializer='z ero'), name='dense_regress_{}'.format(nb_classes))(out)
    return [out_class, out_regr]
```

Calculate IoU (Intersection of Union)

In [ ]:

```
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 [ ]:

```
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
a ccording 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_anc - x1_anc))
       th = np.log((gta[bbox_num, 3] - gta[bbox_num, 2]) / (y2_anc - y1_anc))

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

       # all GT boxes should be mapped to an anchor box, so we keep 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 does 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 IOU 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 included 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_size_idx] =
     1 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 elif bbox_type == 'neutral':
      y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 elif bbox_type == 'pos':
      y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
      1 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_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[idx,2] + n_anchr
atios *
    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[idx,2] +
n_anchr atios *
    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_dx_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 negative
 # 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 [ ]:

```
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 [ ]:

```
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
a ccording 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.INTER_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,
resiz ed_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,
debug_img, num_pos

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

Define loss functions for all four outputs

In [ ]:

```
lambda_rpn_regr = 1.0
lambda_rpn_class = 1.0

lambda_cls_regr = 1.0
lambda_cls_class = 1.0
```

```
epsilon = 1e-4
```

In [ ]:

```
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 neg
ative
    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 [ ]:

```
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

    # 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
large r 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 bbox
es
    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 [ ]:

```
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=m ax_boxes)[0]
 return result
```

Start training

In [ ]:

```
base_path = 'drive/My Drive/GP'

train_path =  'drive/My Drive/GP/Dataset/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_resnet50.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/resnet50_weights_tf_dim_ordering_tf_kernels.h5')

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

In [ ]:

```
# 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 [ ]:

```
#--------------------------------------------------------#
# 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) )
```

In [ ]:

```
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'.f ormat(config_output_filename))
```

In [ ]:

```
# Shuffle the images with
seed random.seed(1)
random.shuffle(train_imgs)
print('Num train samples (images) {}'.format(len(train_imgs)))
```

In [ ]:

```
# 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 [ ]:

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

In [ ]:

```
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_s tride))
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['bbo xes'][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['bbox es'][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['bbox es'][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['bbo xes'][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]+int(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.FONT_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()
```

Build the model

In [ ]:

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

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

# define the base network (can be VGG, Resnet50, Inception, etc)

shared_layers = nn_base(img_input, trainable=True)
```

In [ ]:

```
# 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(shared_layers, roi_input, C.num_rois, nb_classes=len(classes_count),

traina ble = True)

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
mo dels
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
conne ct 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', 'loss_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)))
```

In [ ]:

```
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')
```

Traning setting

In [ ]:

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

epoch_length = 1000
num_epochs = 100
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 [ ]:

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

Start

In [ ]:

```
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 {} previou
iterations'.format(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 settings 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.common.image_dim_ordering(),
use_regr=True, overlap_thresh=0.7, max_boxes=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 sam
les
                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=Fal
e).tolist()

                # Randomly choose (num_rois - num_pos) neg
                samples try:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois - len(selected_
os_samples), replace=False).tolist()
                except:
                    selected_neg_samples = np.random.choice(neg_samples, C.num_rois -
len(selected_ os_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 an
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.m ean(losses[:iter_num, 1])),
                                      ('final_cls', np.mean(losses[:iter_num, 2])), ('final_regr',
p.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 + l
ss_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_los
,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.')
```

Graph

In [ ]:

```
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()
```

In [ ]:

```

```

Appendix 5 code for testing (ResNet50)

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:

```
!ls
```

Import libs

In [ ]:

```
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, Add, Input, Activation, Conv2D, MaxPooling2D,
AveragePooli ng2D, Dropout, ZeroPadding2D, BatchNormalization
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.initializers import glorot_uniform
from keras.models import Model
from keras.utils import generic_utils
from keras.engine import Layer, InputSpec
from keras import initializers, regularizers
```

Config setting

In [ ]:

```
class Config:

 def __init__(self):

   # Print the process or
   not self.verbose = True
   # Name of base network
   self.network = 'resnet'
 # 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 = [64, 128, 256]

  # 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
```

Parser the data from annotation file

In [ ]:

```
def get_data(input_path):
 """Parser 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. {'Car': 2383, 'Mobile phone': 1108, 'Person': 3745}
  class_mapping: dict{key:class_name, value: idx}
   e.g. {'Car': 0, 'Mobile phone': 1, 'Person': 2}
 """
 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,
y 2, 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 background 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': in
t(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

In [ ]:

```
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.common.image_dim_ordering()
        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(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,
s elf.nb_channels))

        # 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()))
```

ResNet-50 model

In [ ]:

```
def get_img_output_length(width, height):
    # return get_output_length(width), get_output_length(height)
    def get_output_length(input_length):
        # zero_pad
        input_length += 6
        # apply 4 strided
        convolutions filter_sizes =
        [7, 3, 1, 1] stride = 2
        for filter_size in filter_sizes:
            input_length = (input_length - filter_size + stride) //
        stride return input_length
    return get_output_length(width), get_output_length(height)
```

In [ ]:

```
def identity_block(input_tensor, kernel_size, filters, stage, block, trainable=True):

    nb_filter1, nb_filter2, nb_filter3 = filters
    bn_axis = 3


    conv_name_base = 'res' + str(stage) + block + '_branch'

    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = Conv2D(nb_filter1, (1, 1), name=conv_name_base + '2a', trainable=trainable)(input_tensor)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2a')(x)
    x = Activation('relu')(x)

    x = Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same', name=conv_name_base +
'2b', trainable=trainable)(x)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2b')(x)
    x = Activation('relu')(x)

    x = Conv2D(nb_filter3, (1, 1), name=conv_name_base + '2c', trainable=trainable)(x)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2c')(x)

    x = Add()([x, input_tensor])
    x = Activation('relu')(x)
    return x


def identity_block_td(input_tensor, kernel_size, filters, stage, block, trainable=True):

    # identity block time distributed

    nb_filter1, nb_filter2, nb_filter3 = filters
    bn_axis = 3

    conv_name_base = 'res' + str(stage) + block + '_branch'

    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = TimeDistributed(Conv2D(nb_filter1, (1, 1), trainable=trainable,
kernel_initializer='normal' ), name=conv_name_base + '2a')(input_tensor)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2a')(x)
    x = Activation('relu')(x)

    x = TimeDistributed(Conv2D(nb_filter2, (kernel_size, kernel_size), trainable=trainable,
kernel_ initializer='normal',padding='same'), name=conv_name_base + '2b')(x)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2b')(x)
    x = Activation('relu')(x)

    x = TimeDistributed(Conv2D(nb_filter3, (1, 1), trainable=trainable,
kernel_initializer='normal' ), name=conv_name_base + '2c')(x)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2c')(x)
    x = Add()([x, input_tensor])
    x = Activation('relu')(x)

    return x

def conv_block(input_tensor, kernel_size, filters, stage, block, strides=(2, 2), trainable=True):

    nb_filter1, nb_filter2, nb_filter3 = filters
    bn_axis = 3

    conv_name_base = 'res' + str(stage) + block + '_branch'

    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = Conv2D(nb_filter1, (1, 1), strides=strides, name=conv_name_base + '2a',
trainable=trainable )(input_tensor)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2a')(x)
    x = Activation('relu')(x)

    x = Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same', name=conv_name_base +
'2b', trainable=trainable)(x)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2b')(x)
    x = Activation('relu')(x)

    x = Conv2D(nb_filter3, (1, 1), name=conv_name_base + '2c', trainable=trainable)(x)
    x = BatchNormalization(axis=bn_axis, name=bn_name_base + '2c')(x)

    shortcut = Conv2D(nb_filter3, (1, 1), strides=strides, name=conv_name_base + '1',
trainable=tra inable)(input_tensor)
    shortcut = BatchNormalization(axis=bn_axis, name=bn_name_base + '1')(shortcut)

    x = Add()([x, shortcut])
    x = Activation('relu')(x)
    return x


def conv_block_td(input_tensor, kernel_size, filters, stage, block, input_shape, strides=(2, 2),
tr ainable=True):
    # conv block time distributed

    nb_filter1, nb_filter2, nb_filter3 = filters
    bn_axis = 3


    conv_name_base = 'res' + str(stage) + block + '_branch'

    bn_name_base = 'bn' + str(stage) + block + '_branch'

    x = TimeDistributed(Conv2D(nb_filter1, (1, 1), strides=strides, trainable=trainable,
kernel_ini tializer='normal'), input_shape=input_shape, name=conv_name_base + '2a')(input_tensor)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2a')(x)
    x = Activation('relu')(x)

    x = TimeDistributed(Conv2D(nb_filter2, (kernel_size, kernel_size), padding='same',
trainable=tr ainable, kernel_initializer='normal'), name=conv_name_base + '2b')(x)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2b')(x)
    x = Activation('relu')(x)

    x = TimeDistributed(Conv2D(nb_filter3, (1, 1), kernel_initializer='normal'),
name=conv_name_base + '2c', trainable=trainable)(x)
    x = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '2c')(x)

    shortcut = TimeDistributed(Conv2D(nb_filter3, (1, 1), strides=strides, trainable=trainable,
ker nel_initializer='normal'), name=conv_name_base + '1')(input_tensor)
    shortcut = TimeDistributed(BatchNormalization(axis=bn_axis), name=bn_name_base + '1')(shortcut)

    x = Add()([x, shortcut])
    x = Activation('relu')(x)
    return x
```

In [ ]:

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

    # Determine proper input shape

    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

    x = ZeroPadding2D((3, 3))(img_input)

    x = Conv2D(64, (7, 7), strides=(2, 2), name='conv1', trainable = trainable)(x)
    x = BatchNormalization(axis=bn_axis, name='bn_conv1')(x)
    x = Activation('relu')(x)
    x = MaxPooling2D((3, 3), strides=(2, 2))(x)

    x = conv_block(x, 3, [64, 64, 256], stage=2, block='a', strides=(1, 1), trainable = trainable)
    x = identity_block(x, 3, [64, 64, 256], stage=2, block='b', trainable = trainable)
    x = identity_block(x, 3, [64, 64, 256], stage=2, block='c', trainable = trainable)

    x = conv_block(x, 3, [128, 128, 512], stage=3, block='a', trainable = trainable)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='b', trainable = trainable)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='c', trainable = trainable)
    x = identity_block(x, 3, [128, 128, 512], stage=3, block='d', trainable = trainable)

    x = conv_block(x, 3, [256, 256, 1024], stage=4, block='a', trainable = trainable)
    x = identity_block(x, 3, [256, 256, 1024], stage=4, block='b', trainable = trainable)
    x = identity_block(x, 3, [256, 256, 1024], stage=4, block='c', trainable = trainable)
    x = identity_block(x, 3, [256, 256, 1024], stage=4, block='d', trainable = trainable)
    x = identity_block(x, 3, [256, 256, 1024], stage=4, block='e', trainable = trainable)
    x = identity_block(x, 3, [256, 256, 1024], stage=4, block='f', trainable = trainable)

    return x
```

RPN layer

In [ ]:

```
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 la
yer
                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: resnet 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: resnet in here
    """
    x = Conv2D(512, (3, 3), padding='same', activation='relu', kernel_initializer='normal',
name='r pn_conv1')(base_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_regress')(x)
    return [x_class, x_regr, base_layers]
```

Classifier layer

In [ ]:

```
def classifier_layers(x, input_shape, trainable=False):

    # compile times on theano tend to be very high, so we use smaller ROI pooling regions
to workaround
    # (hence a smaller stride in the region that follows the ROI pool)
    x = conv_block_td(x, 3, [512, 512, 2048], stage=5, block='a', input_shape=input_shape,
strides=( 2, 2), trainable=trainable)
    x = identity_block_td(x, 3, [512, 512, 2048], stage=5, block='b', trainable=trainable)
    x = identity_block_td(x, 3, [512, 512, 2048], stage=5, block='c', trainable=trainable)
    x = TimeDistributed(AveragePooling2D((7, 7)), name='avg_pool')(x)

    return x
```

In [ ]:

```
def classifier(base_layers, input_rois, num_rois, nb_classes = 21, trainable=False):

    # compile times on theano tend to be very high, so we use smaller ROI pooling regions
to workaround

    pooling_regions = 14
    input_shape = (num_rois,14,14,1024)

    out_roi_pool = RoiPoolingConv(pooling_regions, num_rois)([base_layers, input_rois])
    out = classifier_layers(out_roi_pool, input_shape=input_shape, trainable=True)

    out = TimeDistributed(Flatten())(out)

    out_class = TimeDistributed(Dense(nb_classes, activation='softmax',
kernel_initializer='zero'), name='dense_class_{}'.format(nb_classes))(out)
    # note: no regression target for bg class
    out _regr = TimeDistributed(Dense(4 * (nb_classes-1), activation='linear',
kernel_initializer='z ero'), name='dense_regress_{}'.format(nb_classes))(out)
    return [out_class, out_regr]
```

Calculate IoU (Intersection of Union)

In [ ]:

```
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 [ ]:

```
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
a ccording 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_anc - x1_anc))
       th = np.log((gta[bbox_num, 3] - gta[bbox_num, 2]) / (y2_anc - y1_anc))

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

       # all GT boxes should be mapped to an anchor box, so we keep 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 does 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 IOU 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 included 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_size_idx] =
     1 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 elif bbox_type == 'neutral':
      y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
     0 elif bbox_type == 'pos':
      y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_idx] =
      1 y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios * anchor_size_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[idx,2] + n_anchr
atios *
    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[idx,2] +
n_anchr atios *
    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_dx_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 negative
 # 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 [ ]:

```
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 [ ]:

```
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 a ccording 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.INTER_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, resiz ed_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, 
debug_img, num_pos

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

In [ ]:

```
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-pyt hon/
 # if there are no boxes, return an empty list

    # 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
large r 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
```

In [ ]:

```
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, 9) 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, 36) 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, 9) 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=m ax_boxes)[0]
 return result
```

In [ ]:

```
base_path = 'drive/My Drive/GP'

test_path = 'drive/My Drive/GP/Dataset/test_annotation.txt' # Test data (annotation file)

test_base_path = 'drive/My Drive/GP/Dataset/test' # Directory to save the test images

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

In [ ]:

```
with open(config_output_filename, 'rb') as f_in:
 C = pickle.load(f_in)

# turn off any data augmentation at test
time C.use_horizontal_flips = False
C.use_vertical_flips = False
C.rot_90 = False
```

In [ ]:

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

r_epochs = len(record_df)

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.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()
```

Test

In [ ]:

```
def format_img_size(img, C):
 """ formats the image size based on config """
 img_min_side = float(C.im_size)
 (height,width,_) = img.shape

 if width <= height:
  ratio = img_min_side/width
  new_height = int(ratio * height)
  new_width = int(img_min_side)
 else:
  ratio = img_min_side/height
  new_width = int(ratio * width)
  new_height = int(img_min_side)
 img = cv2.resize(img, (new_width, new_height),

 interpolation=cv2.INTER_CUBIC) return img, ratio
def format_img_channels(img, C):
 """ formats the image channels based on config """
 img = img[:, :, (2, 1, 0)]
 img = img.astype(np.float32)
 img[:, :, 0] -= C.img_channel_mean[0]
 img[:, :, 1] -= C.img_channel_mean[1]
 img[:, :, 2] -= C.img_channel_mean[2]
 img /= C.img_scaling_factor
 img = np.transpose(img, (2, 0, 1))
 img = np.expand_dims(img, axis=0)
 return img

def format_img(img, C):
 """ formats an image for model prediction based on config """
 img, ratio = format_img_size(img, C)
 img = format_img_channels(img, C)
 return img, ratio

# Method to transform the coordinates of the bounding box to its original

size def get_real_coordinates(ratio, x1, y1, x2, y2):

 real_x1 = int(round(x1 // ratio))
 real_y1 = int(round(y1 // ratio))
 real_x2 = int(round(x2 // ratio))
 real_y2 = int(round(y2 // ratio))

 return (real_x1, real_y1, real_x2 ,real_y2)
```

In [ ]:

```
# Feature num:
# VGG: 512
# Resnet50: 1024
 # Resnet101: 1024

num_features = 1024

input_shape_img = (None, None, 3)

input_shape_features = (None, None, num_features)

img_input = Input(shape=input_shape_img)
```

```
roi_input = Input(shape=(C.num_rois, 4))
feature_map_input = Input(shape=input_shape_features)

# define the base network (can be VGG, Resnet50, Inception, etc)
shared_layers = nn_base(img_input, trainable=True)
# define the RPN, built on the base layers
num_anchors = len(C.anchor_box_scales) * len(C.anchor_box_ratios)
rpn_layers = rpn_layer(shared_layers, num_anchors)

classifier = classifier(feature_map_input, roi_input, C.num_rois,
nb_classes=len(C.class_mapping), trainable = True)
model_rpn = Model(img_input, rpn_layers)
model_classifier_only = Model([feature_map_input, roi_input], classifier)

model_classifier = Model([feature_map_input, roi_input], classifier)

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)

model_rpn.compile(optimizer='sgd', loss='mse')
model_classifier.compile(optimizer='sgd', loss='mse')
```

In [ ]:

```
# Switch key value for class mapping
class_mapping = C.class_mapping
class_mapping = {v: k for k, v in
class_mapping.items()} print(class_mapping)
class_to_color = {class_mapping[v]: np.random.randint(0, 255, 3) for v in class_mapping}
```

In [ ]:

```
test_img_num = 30
test_imgs = os.listdir(test_base_path)

imgs_path = []
for i in range(test_img_num):
 idx = np.random.randint(len(test_imgs))
 imgs_path.append(test_imgs[idx])

all_imgs = []

classes = {}
```

In [ ]:

```
# If the box classification value is less than this, we ignore this
box bbox_threshold = 0.5
for idx, img_name in enumerate(imgs_path):
    if not img_name.lower().endswith(('.bmp', '.jpeg', '.jpg', '.png', '.tif', '.tiff')):
        continue
    print(img_name)
    st = time.time()
    filepath = os.path.join(test_base_path, img_name)

    img = cv2.imread(filepath)

    X, ratio = format_img(img, C)

    X = np.transpose(X, (0, 2, 3, 1))

    # get output layer Y1, Y2 from the RPN and the feature maps F
    # Y1: y_rpn_cls
    # Y2: y_rpn_regr
    [Y1, Y2, F] = model_rpn.predict(X)

    # Get bboxes by applying NMS
    # R.shape = (300, 4)
    R = rpn_to_roi(Y1, Y2, C, K.common.image_dim_ordering(), overlap_thresh=0.7)
    # convert from (x1,y1,x2,y2) to (x,y,w,h)
    R[:, 2] -= R[:, 0]
    R[:, 3] -= R[:, 1]

    # apply the spatial pyramid pooling to the proposed
    regions bboxes = {}
    probs = {}

    for jk in range(R.shape[0]//C.num_rois + 1):
        ROIs = np.expand_dims(R[C.num_rois*jk:C.num_rois*(jk+1), :], axis=0)
        if ROIs.shape[1] == 0:
            break

        if jk == R.shape[0]//C.num_rois:
            #pad R
            curr_shape = ROIs.shape
            target_shape = (curr_shape[0],C.num_rois,curr_shape[2])
            ROIs_padded = np.zeros(target_shape).astype(ROIs.dtype)
            ROIs_padded[:, :curr_shape[1], :] = ROIs
            ROIs_padded[0, curr_shape[1]:, :] = ROIs[0, 0, :]
            ROIs = ROIs_padded

        [P_cls, P_regr] = model_classifier_only.predict([F, ROIs])

        # Calculate bboxes coordinates on resized
        image for ii in range(P_cls.shape[1]):
            # Ignore 'bg' class
         if np.max(P_cls[0, ii, :]) < bbox_threshold or np.argmax(P_cls[0, ii, :]) == (P_cls.shap
e[2] - 1):
                continue

            cls_name = class_mapping[np.argmax(P_cls[0, ii, :])]

            if cls_name not in bboxes:
                bboxes[cls_name] = []
                probs[cls_name] = []

            (x, y, w, h) = ROIs[0, ii, :]

            cls_num = np.argmax(P_cls[0, ii, :])
            try:
                (tx, ty, tw, th) = P_regr[0, ii,
                4*cls_num:4*(cls_num+1)] tx /= C.classifier_regr_std[0]
                ty /= C.classifier_regr_std[1]
                tw /= C.classifier_regr_std[2]
                th /= C.classifier_regr_std[3]
                x, y, w, h = apply_regr(x, y, w, h, tx, ty, tw, th)
            except:
                pass
         bboxes[cls_name].append([C.rpn_stride*x, C.rpn_stride*y, C.rpn_stride*(x+w), C.rpn_strid
e*(y+h)])
            probs[cls_name].append(np.max(P_cls[0, ii, :]))

    all_dets = []

    for key in bboxes:
        bbox = np.array(bboxes[key])

        new_boxes, new_probs = non_max_suppression_fast(bbox, np.array(probs[key]), overlap_thresh=
0.2)
        for jk in range(new_boxes.shape[0]):
            (x1, y1, x2, y2) = new_boxes[jk,:]

            # Calculate real coordinates on original image
            (real_x1, real_y1, real_x2, real_y2) = get_real_coordinates(ratio, x1, y1, x2, y2)

            cv2.rectangle(img,(real_x1, real_y1), (real_x2, real_y2),
(int(class_to_color[key][0]), int(class_to_color[key][1]), int(class_to_color[key][2])),4)
            textLabel = '{}: {}'.format(key,int(100*new_probs[jk]))
            all_dets.append((key,100*new_probs[jk]))

            (retval,baseLine) = cv2.getTextSize(textLabel,cv2.FONT_HERSHEY_COMPLEX,1,1)
            textOrg = (real_x1, real_y1-0)

            cv2.rectangle(img, (textOrg[0] - 5, textOrg[1]+baseLine - 5), (textOrg[0]+retval[0] +
5, textOrg[1]-retval[1] - 5), (0, 0, 0), 1)
```

```
            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, 1, (0, 0, 0), 1)

    print('Elapsed time = {}'.format(time.time() - st))

    print(all_dets)
    plt.figure(figsize=(10,10))
    #plt.grid()
    plt.imshow(cv2.cvtColor(img,cv2.COLOR_BGR2RGB))
    plt.show()
```

Measure mAP

In [ ]:

```
def get_map(pred, gt, f):
 T={}
 P={}
 fx, fy = f

 for bbox in gt:
  bbox['bbox_matched'] = False

 pred_probs = np.array([s['prob'] for s in pred])
 box_idx_sorted_by_prob = np.argsort(pred_probs)[::-1]

 for box_idx in box_idx_sorted_by_prob:
  pred_box = pred[box_idx]
  pred_class = pred_box['class']
  pred_x1 = pred_box['x1']
  pred_x2 = pred_box['x2']
  pred_y1 = pred_box['y1']
  pred_y2 = pred_box['y2']
  pred_prob = pred_box['prob']
  if pred_class not in P:
   P[pred_class] = []
   T[pred_class] = []
  P[pred_class].append(pred_prob)
  found_match = False

  for gt_box in gt:
   gt_class = gt_box['class']
   gt_x1 = gt_box['x1']/fx
   gt_x2 = gt_box['x2']/fx
   gt_y1 = gt_box['y1']/fy
   gt_y2 = gt_box['y2']/fy
   gt_seen = gt_box['bbox_matched']
   if gt_class != pred_class:
    continue
   if gt_seen:
    continue
   iou_map = iou((pred_x1, pred_y1, pred_x2, pred_y2), (gt_x1, gt_y1, gt_x2, gt_y2))
   if iou_map >= 0.5:
    found_match = True
    gt_box['bbox_matched'] = True
    break
   else:
    continue

  T[pred_class].append(int(found_match))

 for gt_box in gt:
  if not gt_box['bbox_matched']:# and not gt_box['difficult']:
   if gt_box['class'] not in P:
    P[gt_box['class']] = []
    T[gt_box['class']] = []

   T[gt_box['class']].append(1)
   P[gt_box['class']].append(0)

 #import pdb
 #pdb.set_trace()
 return T, P
```

In [ ]:

```
def format_img_map(img, C):
 """Format image for mAP. Resize original image to C.im_size (300 in here)

 Args:
  img: cv2 image
  C: config

 Returns:
  img: Scaled and normalized image with expanding dimension
  fx: ratio for width scaling
  fy: ratio for height scaling
 """

 img_min_side = float(C.im_size)
 (height,width,_) = img.shape

 if width <= height:
  f = img_min_side/width
  new_height = int(f * height)
  new_width = int(img_min_side)
 else:
  f = img_min_side/height
  new_width = int(f * width)
  new_height = int(img_min_side)
 fx = width/float(new_width)
 fy = height/float(new_height)
 img = cv2.resize(img, (new_width, new_height), interpolation=cv2.INTER_CUBIC)
 # Change image channel from BGR to
 RGB img = img[:, :, (2, 1, 0)]
 img = img.astype(np.float32)
 img[:, :, 0] -= C.img_channel_mean[0]
 img[:, :, 1] -= C.img_channel_mean[1]
 img[:, :, 2] -= C.img_channel_mean[2]
 img /= C.img_scaling_factor
 # Change img shape from (height, width, channel) to (channel, height, width)
 img = np.transpose(img, (2, 0, 1))
 # Expand one dimension at axis 0
 # img shape becames (1, channel, height, width)
 img = np.expand_dims(img, axis=0)
 return img, fx, fy
```

In [ ]:

```
print(class_mapping)
```

In [ ]:

```
# This might takes a while to parser the

data test_imgs, _, _ = get_data(test_path)
```

In [ ]:

```
T={}
P={}
mAPs = []
total_time = []
FP_num = 0
for idx, img_data in enumerate(test_imgs):
    print('{}/{}'.format(idx,len(test_imgs)))
    st = time.time()
    filepath = img_data['filepath']

    img = cv2.imread(filepath)

    X, fx, fy = format_img_map(img, C)

    # Change X (img) shape from (1, channel, height, width) to (1, height, width,

    channel) X = np.transpose(X, (0, 2, 3, 1))

    # get the feature maps and output from the RPN
    [Y1, Y2, F] = model_rpn.predict(X)


    R = rpn_to_roi(Y1, Y2, C, K.common.image_dim_ordering(), overlap_thresh=0.7)

    # convert from (x1,y1,x2,y2) to (x,y,w,h)
    R[:, 2] -= R[:, 0]
    R[:, 3] -= R[:, 1]

    # apply the spatial pyramid pooling to the proposed
    regions bboxes = {}
    probs = {}

    for jk in range(R.shape[0] // C.num_rois + 1):
        ROIs = np.expand_dims(R[C.num_rois * jk:C.num_rois * (jk + 1), :], axis=0)
        if ROIs.shape[1] == 0:
            break

        if jk == R.shape[0] // C.num_rois:
            # pad R
            curr_shape = ROIs.shape
            target_shape = (curr_shape[0], C.num_rois, curr_shape[2])
            ROIs_padded = np.zeros(target_shape).astype(ROIs.dtype)
            ROIs_padded[:, :curr_shape[1], :] = ROIs
            ROIs_padded[0, curr_shape[1]:, :] = ROIs[0, 0, :]
            ROIs = ROIs_padded

        [P_cls, P_regr] = model_classifier_only.predict([F, ROIs])

        # Calculate all classes' bboxes coordinates on resized image (300, 400)
        # Drop 'bg' classes bboxes
        for ii in range(P_cls.shape[1]):

            # If class name is 'bg', continue
            if np.argmax(P_cls[0, ii, :]) == (P_cls.shape[2] - 1):
                continue

            # Get class name
            cls_name = class_mapping[np.argmax(P_cls[0, ii, :])]

            if cls_name not in bboxes:
                bboxes[cls_name] = []
                probs[cls_name] = []

            (x, y, w, h) = ROIs[0, ii, :]

            cls_num = np.argmax(P_cls[0, ii, :])
            try:
                (tx, ty, tw, th) = P_regr[0, ii, 4 * cls_num:4 * (cls_num + 1)]
                tx /= C.classifier_regr_std[0]
                ty /= C.classifier_regr_std[1]
                tw /= C.classifier_regr_std[2]
                th /= C.classifier_regr_std[3]
                x, y, w, h = roi_helpers.apply_regr(x, y, w, h, tx, ty, tw, th)
            except:
                pass
            bboxes[cls_name].append([16 * x, 16 * y, 16 * (x + w), 16 * (y + h)])
            probs[cls_name].append(np.max(P_cls[0, ii, :]))

    all_dets = []

    for key in bboxes:
        bbox = np.array(bboxes[key])

        # Apply non-max-suppression on final bboxes to get the output bounding boxe
        new_boxes, new_probs = non_max_suppression_fast(bbox, np.array(probs[key]), overlap_thresh=
0.5)
        for jk in range(new_boxes.shape[0]):
            (x1, y1, x2, y2) = new_boxes[jk, :]
            det = {'x1': x1, 'x2': x2, 'y1': y1, 'y2': y2, 'class': key, 'prob':
            new_probs[jk]} all_dets.append(det)

    print('Elapsed time = {}'.format(time.time() -
    st)) elapsed_time = time.time() - st
    total_time.append(elapsed_time)
    t, p = get_map(all_dets, img_data['bboxes'], (fx, fy))
    for key in t.keys():
        if key not in T:
            T[key] = []
            P[key] = []
        T[key].extend(t[key])
        P[key].extend(p[key])
    all_aps = []

    for key in T.keys():
        ap = average_precision_score(T[key], P[key])
        print('{} AP: {}'.format(key, ap))
        all_aps.append(ap)
    print('mAP = {}'.format(np.mean(np.array(all_aps))))
    mAPs.append(np.mean(np.array(all_aps)))
    #print(T)
    #print(P)
    if not all_dets:
      FP_num += 1
print()
print('mean average precision:', np.mean(np.array(mAPs)))
```

In [ ]:

```
mAP = [mAP for mAP in mAPs if str(mAP)!='nan']
mean_average_prec = round(np.mean(np.array(mAP)), 3)
print('After training %dk batches, the mean average precision
is %0.3f'%(len(record_df), mean_average_prec))
print('Average elapsed time:', np.average(np.array(total_time)))
print('Percentage of images that without bounding boxes among all:', FP_num/len(test_imgs))

# record_df.loc[len(record_df)-1, 'mAP'] = mean_average_prec
# record_df.to_csv(C.record_path, index=0)
# print('Save mAP to {}'.format(C.record_path))
```

In [ ]: