Elective in AI - Module 1
-
Project Report

Damiano Zappia
ID number: 1870958
-
Prof. Fiora Pirri

October 19, 2020

1 Introduction

The aim of this project is to implement and utilize the Keras RetinaNet, a one stage object detector that analising images is capable of recognising objects, actions etc inside them.
Thus before to begin presenting the work done in this project, a brief focus on RetinaNet is due.
All the code developed for this project, together with images, results, and dataset's files, are available at the following GitHub link:
https://github.com/damianozappia/ElectiveInAI-Module1-project

1.1 Keras Retinanet

Object detectors are divided into 2 different categories: one stage object detectors and two stage object detectors.
The highest accuracy ones are nowadays the latters, where the classifier is applied to a sparse set of candidate object locations, detected in the first stage.
On the other hand, the one stage detectors are applied over a regular and dense sampling of possible object locations, with the advantage of being faster and simpler, but with the con of being less accurate.
Keras Retinanet focuses precisely on this aspect, tring to face the main problem of the one stage detectors, the extreme foreground - background class imbalance that is encountered during the training of dense detectors. Indeed, Keras Rerinanet is actually a dense detector, differing from the others in the loss that it uses in order to detect objects inside images.
The loss implemented by the network is a new kind of function called **Focal Loss, that I will discuss below**.

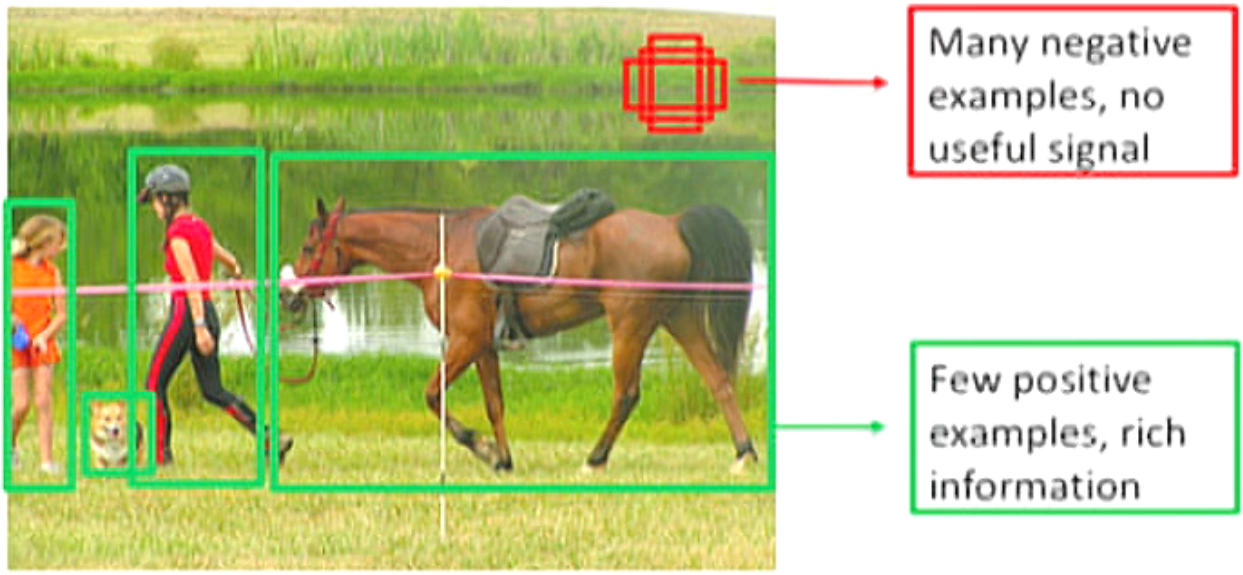
1.1.1 Two stage vs One stage detectors

Two stages object detectors aim to decrease the large number of negative examples in each image, that result from sliding on the image some dense and predefined windows, called *anchors*, doing this by first determining regions where most likely objects appear, the so called Regions of Interests (*RoIs*), and then processing these *RoIs* with a detection network that outputs the object's detection result by visualizing bounding boxes with associated level of confidence inside the image.
Finally in all the state of the art two-stages detectors a non-maxima suppresion method is applied in order to remove duplicates or highly-overlapping results.

One-stage detectors instead, have the aim to predict the detection result directly from anchors, without any proposal elimination stage, directly after extracting the features from image. This, as previously said, has the advantage of speed, but at the cost of sacrificing accuracy.
Here comes the challenge in Keras Retinanet, try to match or surpass the accuracy of two stages detectors while keeping the same speed or even more.

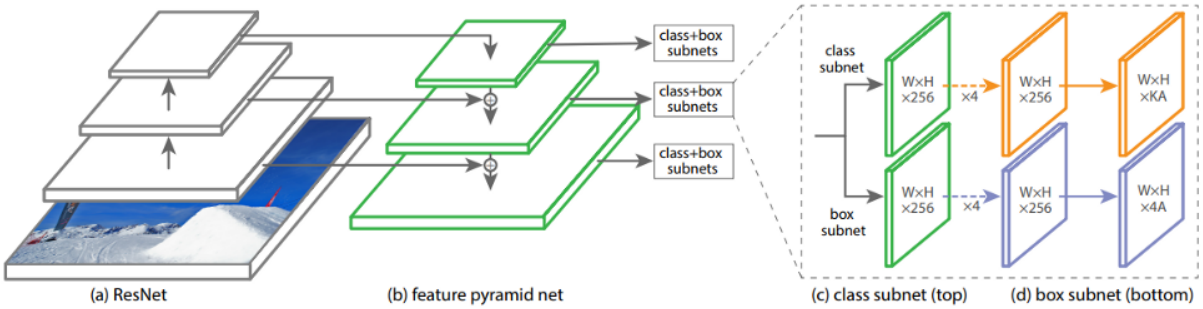
1.1.2 Class imbalance

The main problem to overcome, is the class imbalance that is present in each image. With class imbalance, is intended a problem that occurs when a certain class is over represented. Especially in the "foreground-background imbalance" it happens that the background instances significantly outnumber the positive ones (figure below).



Indeed, usually detectors evaluate about 10^4 to 10^5 locations in each image, but only few of them contain actually objects. This has the effect of causing inefficient training as most locations are easy negatives and thus don't contribute to useful signal, and overhelming training with a lot of samples that should not be there, leading to a degenerate model.
The proposed loss by RetinaNet, *focal loss*, naturally handles this class imbalance problem.

Figure 1: RetinaNet Detector Architecture



1.1.3 Focal Loss

The focal loss function designed for RetinaNet, address the one stage object detection scenario, and fights the high class imbalance. It is introduced starting from the cross entropy loss for binary classification, and modifies it in order to down-weight the easy examples and focus the training on hard examples. Specifically, a modulating factor is added to the cross entropy loss, $(1-p_t)^\gamma$, where γ is a the tunable *focusing* parameter, and p_t is defined as:

$$p_t = \begin{cases} p & \text{if y} = 1 \\ 1-p & \text{otherwise} \end{cases}$$

With p representing the model's estimated probability for the class with label $y = 1$.
The focal loss is thus defined as:

$$FL(p_t) = -(1-p_t)^\gamma log(p_t)$$

With this formulation, when an example is misclassified and p_t is small, the modulating factor is near one and the loss results unaffected. While if $p_t \to 1$, the factor goes to zero and the loss for well classified examples is down-weighted. The focusing parameter γ, adjust smoothly the rate at which easy examples are downweighted. There fore, when $\gamma = 0$, Focal Loss is equivalent to Cross Entropy loss , and raising γ the effect of the modulating factor is increased.

1.1.4 RetinaNet Detector

The Retinanet is a single unified network, composed by a backbone network, and two specific subnetworks.
The backbone network has the task to compute a convolutional feature map over the image, then the first subnetwork performs a convolutional object classification on the output given by the backbone, and the second subnetwork performs a convolutional bounding box regression.

2 Project Development

The aim of the project was to use the implementation of Keras RetinaNet by Fyzir, to satisfy two different assignments:

- Develop a python notebook to train it on one of the well known and large datasets provided in the github page project, obtaining the weights of the network
- Test the network on some of the training videos composing the AVA actions dataset by Google, and finding the error.

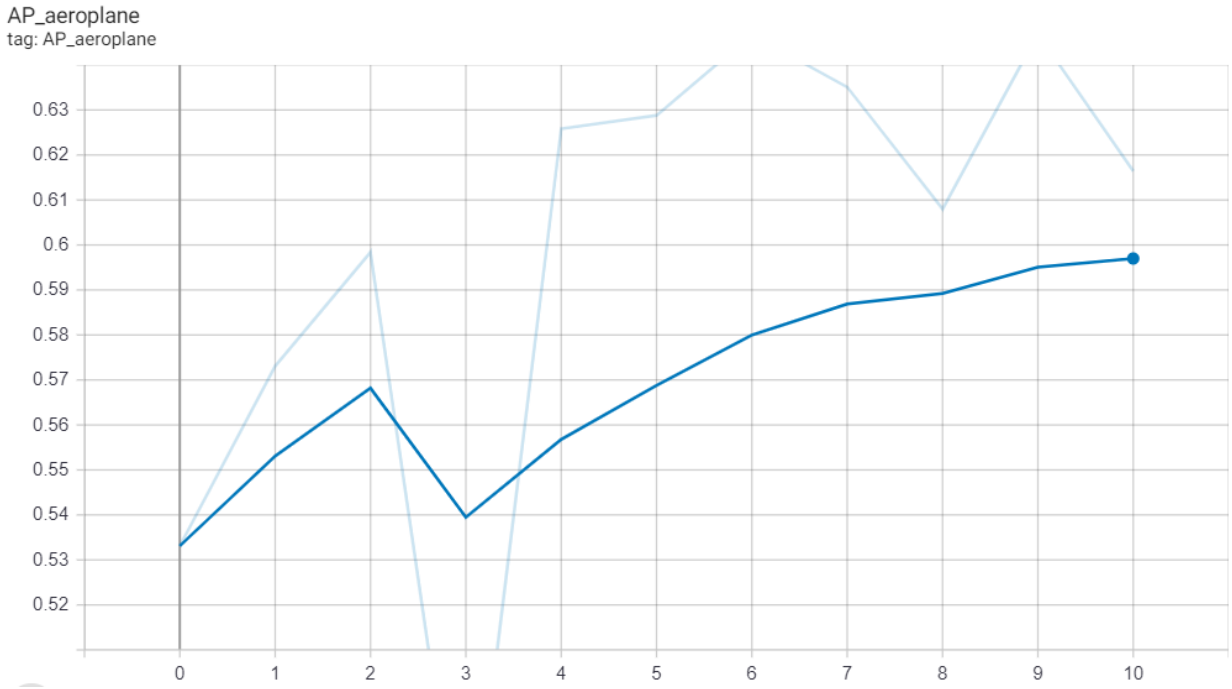
Thus I will now explain and analyze the two different assignments separately, the code I have developed, the training procedure, and the results obtained. I will show also some examples of utilization of the network to make inference.

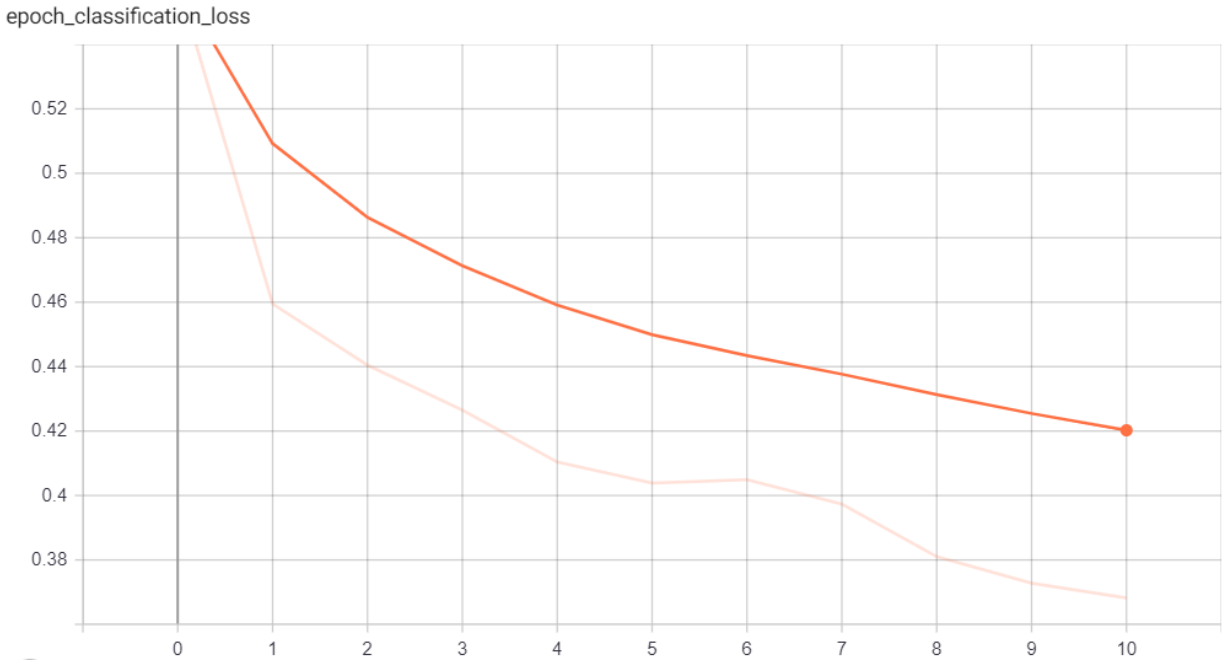
2.1 First Assignment

As starting point I developed a python notebook using the Google Colab platform, with the purpose of making the code of Keras RetinaNet working. For this task I first cloned the repository on the notebook, and after installing all the required packages, I chose to train the network on the Pascal VOC dataset.
It is a public available dataset very commonly used in object recognition challenges, and is composed by 20 classes, with high quality images, having complete annotation for objects.
For my development I used two different versions of Pascal VOC dataset, specifically the 2007 version and the 2012 version, that is more recent and thus bigger, having more images.
For both versions I also did not started from scratch but I decided to use pre-trained weights on the COCO datsets with resnet50 backbone. This because colab offers limited time to train the model, and also it is a good choice to have already a starting point, that allows to the network to begin training already with "some experience" and ability to better accomplish the task.
The command line to train this procedure, after importing the dataset, is the following:

```
! python keras_retinanet/bin/train.py --random-transform --weights {
   PRETRAINED_MODEL} --epochs 30  pascal ./VOCdevkit/VOC2012
```

At the end of the training I obtained 2 different .h5 models, one for each dataset, that contains inside the current weights that the network has at that reached point.
Below I reported two images of training results for the Network on the Pascal VOC 2012 dataset. It is possible to see how just **after 10 epochs**, that required though more than 8 hours of training, the results are showing positive behaviour:





Unfortunately given the very restricted time available to train the model, I did not reached an high number of trained epochs, but the progress of the training leaves with good hopes in case of further execution.

3 Second Assignment

The second assignment has been a lot more trivial. This because now the dataset from where to take the images, with annotations and all the relevant and linked information was in a format quite different from the one requested by the Keras RetinaNet in order to work properly.
More specifically, the AVA dataset is composed by many rows in csv format, where for each row a YouTube video ID is specified, from where to take the image, with the related informations. In particular, as reported on the AVA website:

" Each row contains an annotation for one person performing an action in an interval, where that annotation is associated with the middle frame. Different persons and multiple action labels are described in separate rows.

The format of a row is the following: video id, middle frame timestamp, person box, action id, person id

- video id: YouTube identifier
- middle frame timestamp: in seconds from the start of the YouTube.
- person box: top-left (*x1, y1*) and bottom-right (*x2,y2*) normalized with respect to frame size, where (0.0, 0.0) corresponds to the top left, and (1.0, 1.0) corresponds to bottom right.
- action id: identifier of an action class
- person id: a unique integer allowing this box to be linked to other boxes depicting the same person in adjacent frames of this video.

"

This format of the data, required a lot of pre-processing in order to extract the images and all their linked informations. Indeed this task has been solved developing a code that operates a series of instructions allowing progressively to extract the data and put them in the required format. Let's analyze the procedure developed in its steps:

3.1 Classes CSV creation

The first thing I've done is to prepare the classes csv file. The classes file for AVA dataset is in the form of a *pbtxt* file, with every action that has the following form:

```
label {
  name: "bend/bow (at the waist)"
  label_id: 1
  label_type: PERSON_MOVEMENT
}
...
```

Starting from it I need to obtain a file containing for each row the class name, followed by its id number. Thus I wrote the following python script in order to do the conversion:

```
import csv
result1 = ""
result2 = ""

with open('ava_action_list_v2.2.pbtxt') as f:
    txt = f.readlines()

for i in range(len(txt)):
  if ("name" in txt[i]) and ("label_id" in txt[i+1]) :
    #get the label id
    aux = txt[i+1].split(" ")
    result1 = aux[3]
    result1 = result1.replace('\n', '')
    #print("result 1 is: ", result1)

    #get the name
    aux = txt[i].split(' "')
    result2 = aux[1]
    result2 = result2.replace('"', '')
    #result2 = result2.replace('\n', '')

    #print("result 2 is: ", result2)
    with open('ava_actions.csv', mode='a') as annotation_file:
              annotation_file = csv.writer(annotation_file, delimiter=',')
              annotation_file.writerow([result2,result1])
```

At the end the code returns the classes file in the correct form.

3.2 Annotation file creation

The next step required to create the annotations csv file, starting from a file that contains for each row the informations reported above.
I decided, for the motivations explained below, to not take all the rows from the original train file, but to take only fiver rows for each different video, with a total of 1155 rows.
For this task the code developed is quite longer and can be divided in parts.

3.2.1 Video downloading

Firstly, I used a python package called *pytube* in order to extract each video reported in the train file, using its ID. To do this I created a for loop that for each row inside the train file takes the first element (the video ID), adds it to the https youtube default url, and search for its availability.
If the video is available, the best resolution is selected, and then the video is downloaded.
All this is resumed with these key lines of code:

```

filename = 'AVA_train_annotations.csv'
data = pd.read_csv(filename, header=None, sep=',', skiprows=count)

for row in range(len(data)):
```

```
6
7   print("fields of row  ", row, " are: \n", data.loc[row], "\n")
8
9   IMAGE_NAME = data.loc[row,0] + '.jpg'
10  VIDEO_URL = 'https://www.youtube.com/watch?v='+str(data.loc[row,0])
11  KEYFRAME_TIME = data.loc[row,1]
12  X_min = float(data.loc[row,2])
13  Y_min = float(data.loc[row,3])
14  X_MAX = float(data.loc[row,4])
15  Y_MAX = float(data.loc[row,5])
16  ACTION_ID = int(data.loc[row,6])
17
18  video = YouTube(VIDEO_URL)
19
20  ...
21
22  video.streams.filter(file_extension = "mp4").all()
23
24  stream = video.streams.get_highest_resolution()
25  stream.download(filename='video')
```

The problem here is that the code does an high number of video download request to the YouTube server, and for this reason after a while the YouTube requests are blocked. To solve this issue I did the followings: first I put the video extraction code in a try-except block, in this way if the next video in the train file is already downloaded, it is kept and we avoid useless requests to the server; second, each time the requests exceed anyway the maximum allowed number, I created a new Colab notebook, that results like a new device for the server, and thus is allowed to download videos. This procedure is thus not easy or quick, and was repeated several times until the images and data extraction is completed.

3.2.2 Images extraction

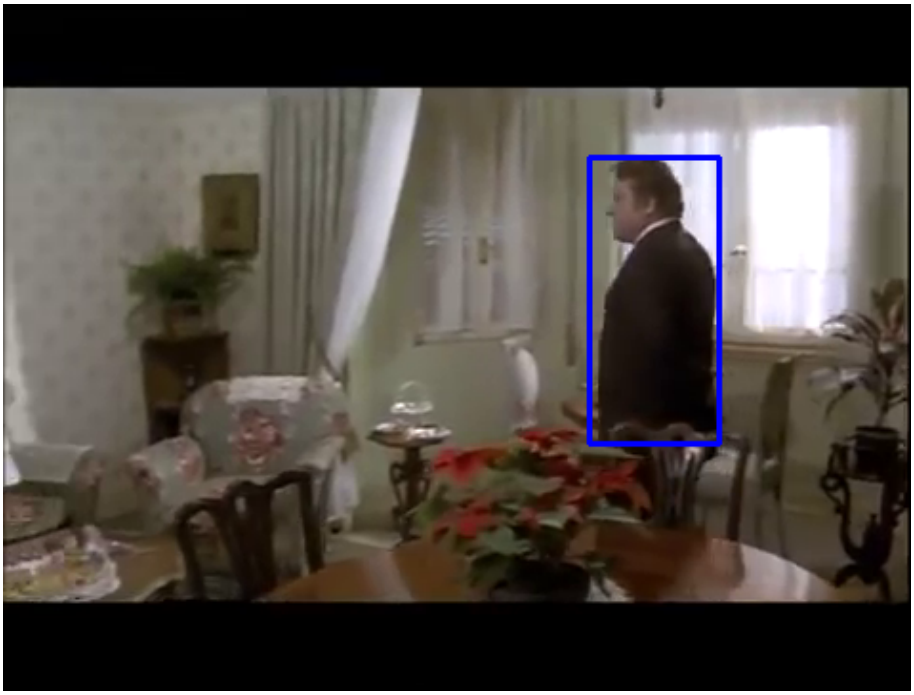
For each video extracted, there are a lists of five actions reported in 5 different images (or better, frames) belonging to the video. The procedure here has been to retrieve all the information for each image, that as expected are:

- Timestamp in the video
- x min coordinate of the box
- y min coordinate of the box
- x max coordinate of the box
- y max coordinate of the box
- related action

These information are read in the train file, and then are saved in 6 variables, that at each iteration of the loop are used to extract the image and save it in the dataset, together with the coordinates of the bounding box in the image, and the action represented.
The code that does what explained is the following:

```
1
2   TIME = float(KEYFRAME_TIME) * 1000
3   print("time is :", TIME, "\n")
4
5   path = 'dataset'
6   cap = cv2.VideoCapture('video.mp4')
7   cap.set(cv2.CAP_PROP_POS_MSEC,TIME)      # Go to the specified milli sec
     . position
8   ret,frame = cap.read()                   # Retrieves the frame at the
     specified second
9   cv2.imwrite(os.path.join(path , IMAGE_NAME), frame)
10
11
12  X_min = X_min * dimensions[1]
13  X_MAX = X_MAX * dimensions[1]
14  Y_min = Y_min * dimensions[0]
15  Y_MAX = Y_MAX * dimensions[0]
16
17  X_min = int(round(X_min, 0))
18  X_MAX = int(round(X_MAX, 0))
19  Y_min = int(round(Y_min, 0))
20  Y_MAX = int(round(Y_MAX, 0))
```

Also, in order to increase the interpretability of the procedure while the code was executing, I decided to use the coordinates information of the bounding box inside the image, to print each extracted image in terminal, with the respective bounding box. An example is showed in the below picture.



This allows the user to better visualize what is happening inside the process.

4 Network Training

Once that all the necessary information has been extracted and put in the required form, I wrote a new notebook responsible for utilizing the network on this new custom dataset. I decided to put the commands for cloning the repository and installing the required packages inside colab without any reference path to external directories, like for example Google Drive. This because the execution from Colab itself is faster and the steps to clone and install the packages, even if has to be done at every new session startup, are very quick and require only few seconds.
Once the code is ready to be used, I import all the packages and specify the path from where to take the dataset (in this case external to colab, either from local machine or drive).
Once all is ready to start, in order to begin the network train I need the following command line:

```
!keras_retinanet/bin/train.py --freeze-backbone --weights {
    PRETRAINED_MODEL} --epochs 50 csv AVA_annotations_v2.csv ava_actions_v3
    .csv
```

There are many options available to use the train.py file. In this case I have :

- –freeze-backbone: used to freeze training of backbone layers, particularly useful with small datasets, like in my case
- –weights: indicates that I used a ResNet50 model, pretrained on Coco dataset, as starting point, instead of beginning from scratch.
- – epochs: used to specify the number of epochs that we want the model is trained

Other options include the possibility to apply some data augmentation with random transformations to images, specify the folder where to save the trained models, and so on and so forth.

Once the train is complete, I use the network to make some inference in images, using a code that takes a number as reference threshold value, indicating the value of confidence above which the model can output the result. To do this, the training model is converted into an inference model, using the scrips from the Keras RetinaNet code. Once the inference model is ready, it can do inference on whatever image the user gives in input to it. The produced result is something similar to the following picture.

5 Results

In the following images, I show some results obtained during the train and usage of the network. In general the results are not particularly good, but there are many explainable reasons for that.

- **First**, the dataset obtained with the images extraction procedure, is small. This because even if the overall number of train annotations in the original AVA dataset is huge, I extracted from that a lower number of lines, because for each line the procedure of extracting the related image is as explained not so quick.

- **Second**, many of the videos reported in the AVA train dataset are no more available on YouTube, and this of course decreases the number of samples extracted that can be used in the training.

- **Third**, the quality of these video is in general not so good, causing images extracted to have low resolution, and in general low level of detail, due to the video to which they belong, that often is some old movie. Thus the quality of the dataset is mediocre.

- **Fourth**, some bounding boxes of the dataset's images, are not representing an emblematic action, or are sometimes misleading, labeling for example with *sit* a situation

in which the subject could very likely be *standing*, and viceversa.

- **Fourth**, the limited amount of time available to train the network, does not allow to reach very good results.

6 Conclusions

To conclude, we can say that the aim of the network is in any case satisfied, because the Keras RetinaNet is easy to set up, and has a quick train and utilization procedure. Also the work and results make believable the idea that with the right improvements on the problematic sides o the procedure, the network can perform very well.
These thoughts are also confirmed by the fact that the Keras Retinanet implementation by Fyzir, was used by the winner of the Kaggle competition "RSNA Pneumonia Detection Challenge".

References

[1] Website: https://github.com/fizyr/keras-retinanet

[2] AVA dataset: http://research.google.com/ava/

[3] Website: Sik-Ho Tsang, "Review: RetinaNet — Focal Loss (Object Detection)", https://towardsdatascience.com/review-retinanet-focal-loss-object-detection-38fba6afabe4

[4] Lin, Goyal, Girshick, Kaiming He, Dollár, "Focal Loss for Dense Object Detection", https://arxiv.org/abs/1708.02002

[5] Oksuz, Can Cam, Kalkan, Akbas, "Imbalance Problems in Object Detection: A Review", https://arxiv.org/abs/1909.00169