ECSE 415 - Final Project Report Fall 2020

Counting People in a Shopping Mall

McGill

Group 06

Hanwen Wang 260778557

Yichen Wang 260761601

Yudong Zhou 260721223

Yifan He 260763870

1. Description of Person Detector (part 1):

For the person detection in the first part of this project, we use the Facebook Detectron2 package as our existing person detection technique. Detectron 2 comes to the rescue if we want to train an object detection model in a snap with a custom dataset. All the models present in the model zoo of the Detectron 2 library are pre-trained on COCO Dataset. We just need to fine-tune our custom dataset on the pre-trained model.With the use of this library, we further obtain bounding boxes around people in the mall images. After several testings, we find that there are many factors which may influence the detection results, such as reflections and dummies. To reduce error and further increase arruacry, we change the value of threshold. Figure 1.1 and 1.2 below illustrate different detection outputs when the threshold values are set to be 0.2 and 0.3 respectively.

Figure 1.1 output when T = 0.2

Figure 1.2 output when T = 0.3

Finally, we choose the threshold value as 0.5. With the use of this value, dummies are included while reflections are excluded as required, and the output meets the expectation.

```
# Inference with a keypoint detection model
cfg = get_cfg()      # get a fresh new config
cfg.merge_from_file(model_zoo.get_config_file("COCO-Keypoints/keypoint_rcnn_R_50_FPN_3x.yaml"))
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5    # set threshold for this model
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Keypoints/keypoint_rcnn_R_50_FPN_3x.yaml")
predictor = DefaultPredictor(cfg)
```

Figure 1.3 code for testing of the threshold value

2. Description of Person Detector (part 2):

In this part, we implement our own model for detecting people in an image. Results from part 1, which are generated using the Facebook Detectron 2 library, will be utilized. The first step to build a predicting model for detecting people in an image is to generate a training set including many positive and negative examples so we could train the SVM model.

For positive examples, a cropping function is written to get those required images. It mainly consists of finding coordinates of the four corners of the detection box from the output instances produced by the results of Part 1. After that, we need to resize all the cropped positive examples to the same size (64*128 in our case) because it will be easier to calculate the HoG feature for comparison. The function is shown in fig 2.1 below.

```
# function to crop positive examples in one image
def crop_positive_examples(img, outputs):
  crop_imgs = []
  crop_imgs_cor = [] # coordinates of positive samples
  out_cpu = outputs["instances"].to("cpu")
  num = out_cpu.pred_boxes.tensor.shape[0] # number of instances
  for i in range(num):
    x1 = int(out_cpu.pred_boxes[i].tensor[0][0])
    y1 = int(out_cpu.pred_boxes[i].tensor[0][1])
    x2 = int(out_cpu.pred_boxes[i].tensor[0][2])
    y2 = int(out_cpu.pred_boxes[i].tensor[0][3])
    crop_img = img[y1:y2, x1:x2]
    # resize the image to width: 64, height: 128
    crop_img = cv2.resize(crop_img,(64,128))
    C = [x1, y1, x2, y2]
    crop_imgs.append(crop_img)
    crop_imgs_cor.append(C)
  return crop_imgs, crop_imgs_cor
```

Fig. 2.1. Positive Example Cropping function

For negative examples, it is a little complicated as the images must not contain any person as it may affect our training model. Therefore, an additional method that ensures random patch selection without any person included is required. What we do is to check whether the randomly selected rectangle patch has any overlap with the positive samples we have just detected using the previous corner coordinates. The function is shown in Fig. 2.2.

```
# function to detect rectangle overlap,
# C1 & C2 are coordinates in form (x1, y1, x2, y2)
def has_overlap(C1, C2):
  if (C1[0]>=C2[2]) or (C1[2]<=C2[0]) or (C1[3]<=C2[1]) or (C1[1]>=C2[3]):
    return False
  else:
    return True
```

Fig. 2.2. Overlap detection function

Afterwards, HoG features should be extracted from our positive and negative examples so the SVM model can learn from the features. The HoG function is imported from skimage library. After that, all the features are transformed into numpy arrays and concatenated into one single training set. The labels are indicated as 1 for positive images and 0 for negative images. For SVM, we choose to use linear SVM and fit all the positive and negative images to the classifier to ensure better accuracy.

We tried several values for the number of positive and negative examples. Finally we choose all the positive features in only the first 100 images, which has a number of 2148. And correspondingly we choose 21 negative features in each image, which has a number of 2100 in total. In this way, we can save much time to generate this model while still keeping the train set large enough. And the number of two features are almost balanced, which avoids the problem that the model is more sensitive to one compared to the other.

Next, we define a function named ‘sliding_window’, which takes the image, window size and step size as inputs, where the step size is defined by the change in x and y each time. The function starts from the top-left corner, ends at the bottom-right corner, and finally returns the coordinates and the patch images resized with (64,128), which are the values of width and the height respectively.

Finally, we define a function named ‘detector’, which collects the detected patches by calling the ‘sliding_window’ function and evaluating the windows. The ‘detector’ function scales the original image and feeds downscaled images into the ‘sliding_window’ function, thus achieving relatively varying window sizes. After using hog again to get feature value for each window sized image, we predict its label and set a threshold on confidence score to further filter out some errors. If its predicted label is 1 and its confidence score is greater than the value we set, it is appended to the detection list. After that, we perform non maximum suppression to remove duplicates. Then all the remaining boxes are considered as valid person detections and are returned.

The choice of window size is determined under the consideration between detection accuracy and time cost. If the window size is too large, it will be hard for the detector to catch positive patches that are relatively small. If the window size is too small, since we need to scale down the image for multiple times to achieve varying window sizes, it will require too many layers of downscaling to detect large patches, which is time-consuming. Therefore, we need to find a window size that can detect small patches and at the same time can prevent the running time from being too large. The step size is also an important factor of the performance and the running speed. A smaller step size will result in a slower running speed, while a larger step size may cause the missing detections of some small patches. After several testings, we set window size to (30,60) and step size to (15,30) so that the detection works with acceptable accuracy and time cost.

3. Description of the duplicate detection removal and person counting:

To remove duplicate detections, we import the non_max_suppression algorithm from imutils.object_detection. In non-maximum suppression, it basically takes out the box with the maximum confidence score from all the inputs and compares it with all the remaining boxes. If the intersection over union between these two boxes is greater than threshold value, the box is removed from the remaining boxes. This whole process is repeated until no box remains. Then all the boxes taken out are the final boxes shown in the image. As a result, in our code implementation, we first get all detected boxes for three different sizes of image and stored in a list named detections. After that we perform the non-maximum suppression algorithm on detections to remove duplicate boxes. At last, we use len(detections) to count the elements stored in detections list, which is the number of detections for one image.

4. Evaluation of Person Detector:

To evaluate the performance of our person detector, an Intersection-over-union (IoU) performance metric is used. The main function of this metric is to calculate the accuracy of an object detector on a particular dataset by computing the area of overlap between the predicted bounding box and the ground-truth bounding box. A function is written to calculate the IoU between two images. The code is shown below in Figure 4.1.

```
# This is a function used to calculate IoU of two patches
def bb_intersection_over_union(boxA, boxB):
  # determine the (x, y)-coordinates of the intersection rectangle
  xA = max(boxA[0], boxB[0])
  yA = max(boxA[1], boxB[1])
  xB = min(boxA[2], boxB[2])
  yB = min(boxA[3], boxB[3])
  # compute the area of intersection rectangle
  interArea = max(0, xB - xA + 1) * max(0, yB - yA + 1)
  # compute the area of both the prediction and ground-truth
  # rectangles
  boxAArea = (boxA[2] - boxA[0] + 1) * (boxA[3] - boxA[1] + 1)
  boxBArea = (boxB[2] - boxB[0] + 1) * (boxB[3] - boxB[1] + 1)
  # compute the intersection over union by taking the intersection
  # area and dividing it by the sum of prediction + ground-truth
  # areas - the interesection area
  iou = interArea / float(boxAArea + boxBArea - interArea)
  # return the intersection over union value
  return iou
```

Fig. 4.1. IoU calculation between two boxes

Since there can be many detected boxes in each image, we need to calculate the IoUs for all pairs of predicted boxes and ground-truth boxes, which requires a matching mechanism to find the corresponding predicted box for a ground-truth box. We choose to use the distance between centers of the boxes as an indicator and a ground-truth box will form a pair with its nearest predicted box to calculate the IoU. If the distance from the nearest predicted box to the current ground-truth box is greater than a threshold, then this ground-truth box will be considered undetected, thus resulting in an IoU value of zero. This mechanism is implemented in a function called 'cal_IoU'.

By calculating the mean IoU value, we obtain an average of 0.2002258067365. The result is not very satisfying, as a good object detector often achieves an IoU value of 0.5 or higher. But it also indicates that we have much room for improvement.

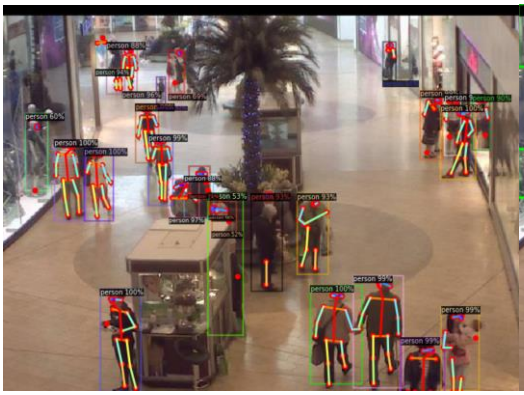

Fig. 4.2. Result from part 1

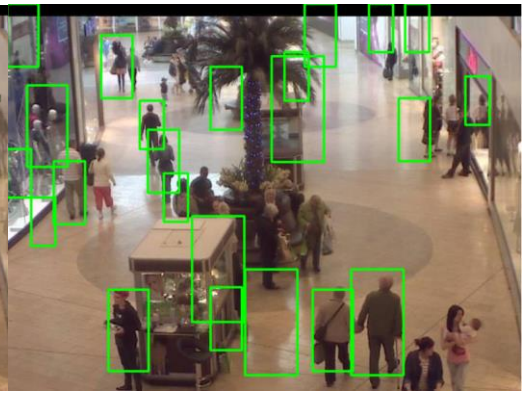

Fig. 4.3. Result from part 2

Figure 4.2 and 4.3 are the detection results from the same image in part 1 and part 2 respectively. As can be seen, there are more missing detections (false negatives) and erroneous detections (false positives) in part 2 compared to part 1.

5. Evaluation of Person Counting:

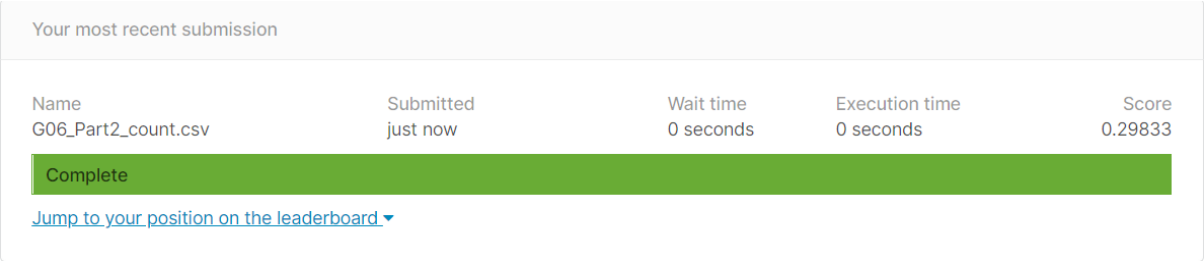

Fig. 5.1. Kaggle submission

Figure 5.1 is the screenshot after we submitted the part 2 result onto Kaggle. As can be seen, the score is 0.29833, which is a better performance than uniform random values. However, although we submitted our csv file onto Kaggle before Dec 14th, the submission was considered late and we cannot find our rank on the website. The rank may be 15, since our

score is between the current 14th and 15th places on the leaderboard, which is shown in Figure 5.2.

Fig. 5.2. Kaggle rank on Dec 14

6. Discussion

Lessons learned during this project

In this project, we reviewed the Histogram of Oriented Gradients and Support Vector Machine, studied Non-Maximum Suppression algorithm and Intersection over Union. We also learnt that the optimal parameters for HoG were found to be four 8x8 pixels cells per block with 9 histogram channels.

Improvement upon our results

Many things can be done to improve our result, because multiple variables and parameters can affect the result. First is the threshold for the model to detect positive examples. Currently we set it to 0.5, however if we increase this value, only people will be considered as a positive example; if we decrease this value, mannequins and reflections of people in the windows will also be considered as positive examples. Second is the number of positive examples and negative examples. In our project, we take them only from the first 100 images out of the total 2000 images to save execution time. If we do not consider the speed problem, taking all positive examples and the same number of negative examples from all 2000 images to form a SVM model should be more accurate. Third is the sliding window problem. Our logic is to keep the window size and step size the same, scale down the image by a factor of 1.3 for 3 times to save running time. If we want to improve accuracy, more window sizes, smaller step size and more iterations with smaller downscale value should be used. In this way, the image can be processed with more details and there can be a higher probability of detecting the human with a right box scale. In addition, the threshold for clf.decision_function and non_maximum_supression will also change the final result, they need to be adjusted together with all factors mentioned above.

Better ways to estimate how many people there are in an image

Faster R-CNN is one alternative approach. It is much faster than its predecessors, which are R-CNN and Fast R-CNN [1]. It basically has the following steps [2]. Generating a convolutional feature map of the input image through a convolutional network, using Region Proposal Network to predict proposals, reshaping those proposals to form a fixed-sized

feature map using a RoI pooling layer and training classification probability and bounding box regression using Softmax Loss and Smooth L1 Loss algorithms. Because its speed is very fast, it can even be used for real time object detection.

YOLO is the other alternative approach. It is used widely because its biggest advantage is that it can achieve high accuracy while also being able to run in real-time [3]. Unlike the R-CNN family which uses regions to localize objects, YOLO considers the entire image and predicts the bounding box coordinates and class probabilities. It is kind of similar to our project, but it is way faster than what we implemented. It also uses the sliding window and non-maximum suppression algorithms. But for the input image, it will be passed through several convolutional layers and two maxpool layers before generating a vector output. This approach has drawbacks like lower mean average precision and localization errors, but they are solved in higher versions of YOLO.

Difficulties we faced in the project

As mentioned above, many factors affect the final detections in an image, including the positive threshold, the size of the training set, the window size and step size and so on. Since running detection on a large number of images (2000) is very time-consuming, we cannot directly see the result when we change some parameters. Consequently, we cannot figure out the best combination of values to achieve a relatively accurate result of detections. We spend time mostly on eliminating wrong boxes in the predicted image, however, we are not able to achieve a better result mainly because of the time budget. This is the biggest difficulty we faced in the project.

The suitability of different types of features

The aim of the HOG is to describe an image by a set of local oriented gradients histogram. These histograms represent occurrences of specific gradient orientation in a local part of images. HOG is usually used to detect a specific object from images. Especially, HOG shows good performance for human detection. However, it has a disadvantage that is very sensitive to image rotation. Therefore, HOG is not a good choice for classification of textures or objects which can often be detected as rotated images. But in our case, since images are collected from a surveillance camera, there is no rotation problem.

Compared with HOG and LBP, Haar-like features have advantages in its calculation speed. Due to the use of integral images, a Haar-like feature of any size can be calculated in constant time. A Haar-like feature considers adjacent rectangular regions at a specific location in a detection window, sums up the pixel intensities in each region and calculates the difference between these sums. But it is useful only if there is a strong intensity difference, such as in face recognition, where eyes and noses have different intensity values. Hence, when applying Haar features on human detection, it may not be that effective.

The local binary pattern (LBP) is an image operator which transforms an image into an array or image of integer labels describing small-scale appearance of the image. However, LBP

operators have a disadvantage where they produce rather long histograms, which slow down the recognition speed especially on large-scale databases. We have already reduced the size of the training data just to improve the speed, therefore LBP is not suitable in our case.

7. Group Work

Hanwen Wang

He searched and made plans for this project. Following the project procedure, he took part in implementing the general logic of each part. He focused mainly on debugging and finding appropriate values for some decisive parameters which can affect the final output.

Yichen Wang

He contributed to the model testing and report writing. He focused mainly on the discussion about improvement of the code and description of the model.

Yudong Zhou

He contributed to code implementation. By completing the functions such as crop_positive_examples, crop_negative_examples and cal_IoU, he made sure that the main logic could work as expected. He also explored the effect of some parameters.

Yifan He

He tested the model and wrote the report. He is mainly responsible for searching ways to improve the code and describing the algorithms written by the software team.

8. Reference

[1] R. Gandhi. “R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms.” towards data science. https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e (accessed Dec. 14, 2020)

[2] “Faster-rcnn原理及相应概念解释.” cnblogs.com. https://www.cnblogs.com/dudumiaomiao/p/6560841.html (accessed Dec. 14, 2020)

[3] alibabazhouyu. “Yolo算法详解.” CSDN. https://blog.csdn.net/alibabazhouyu/article/details/81150479 (accessed Dec. 14, 2020)

[4] Biomisa.org, 2020. [Online]. Available: http://biomisa.org/uploads/2016/10/Lect-15.pdf. Accessed: Dec. 14, 2020