Spring 2022 Introduction to Artificial Intelligence

Report of Homework #1

Student name: 劉子齊 Jonathan
Student ID: 0716304

A. Load and prepare your dataset

```python
# Begin your code (Part 1)
# raise NotImplementedError("To be implemented")

dataset = []

for foldername in os.listdir(dataPath):
  if(foldername == "car"):
    label = 1
  elif(foldername == "non-car"):
    label = 0

  for filename in os.listdir(os.path.join(dataPath, foldername)):
      img = cv2.imread(os.path.join(dataPath, foldername, filename), 0)
      img = cv2.resize(img, (36, 16))
      # img = np.rot90(img, 0)
      tp_data = tuple()
      tp_data = (img, label)
      dataset.append(tp_data)

# End your code (Part 1)
```

For the first part, I applied two for loops in my program. The first for loop is for the two folders, which are folder with car images and with non-car images. After accessing the folder, the second for loop help up to go through all the images. To get the required format of the input, after reading an image by cv2, I resize it into (36, 16) first. After resizing it, I construct a new tuple names “tp_data” then append the numpy array of the image and its classification label into a list named “dataset”, which is also the return list of the loadImages function.

B. Implement Adaboost algorithm

For the second part of HW #1, I will give a brief introduction to the algorithms applied in HW #1, which are the Viola-Jones object detection algorithm, and the Adaboost algorithm for feature selection.

As a brief introduction to Viola-Jones detector, it conducts the following four steps:

- Calculating Integral Image

- Calculating Haar like features
- AdaBoost Learning Algorithm
- Cascade Filter

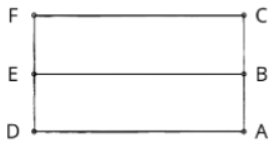
For the first step, since I have to do some calculation to obtain the Haar features. By normal calculation techniques, sure I can still get the features, but I will have to calculate the average of a given region multiple times and the time complexity of these operations are O(n^2).

By integral image approach, I can make the time complexity to achieve O(1).which is compute the integral image, which can be found in the following figure.

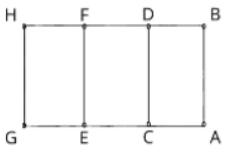
```
print("Computing integral images")
posNum, negNum = 0, 0
iis, labels = [], []
for i in range(len(dataset)):
    iis.append(utils.integralImage(dataset[i][0]))
    labels.append(dataset[i][1])
    if dataset[i][1] == 1:
        posNum += 1
    else:
        negNum += 1
```

In the second step, by using integral images I can achieve constant time evaluation of Haar features, which have the following three types:

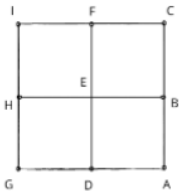
- Edge Features or 2 Rectangular Features
- Line Features or 3 Rectangular Features
- Diagonal Features or 4 Rectangular Features



2 Rectangle Feature



3 Rectangle Feature



4 Rectangle Feature

The following is the corresponding function of this step:

```
def buildFeatures(self, imageShape):
    """
    Builds the possible features given an image shape.
      Parameters:
        imageShape: A tuple of form (height, width).
      Returns:
        A numpy array of HaarFeature class.
    """
```

As there will be so many features, all of them will not include parking spots in it. From the integral image, I will get two possible things: features containing cars and features containing no cars. I need only those features which contains cars.

In HW #1, this job will be done by Adaboost. It will help to sample cars from rest of the image parts using weak classifiers and cascade. The overall process used is ensemble method. A weighted arrangement of all these features are used in evaluating and deciding any given slot has car or not, which will eliminate all redundant features.

However, before I start to train the weak classifiers, since the value of each feature for the images never changes, I should apply all the features before training. Applying the features before training also allowed us to pre-select features. In the following figure, I used "SelectPercentile" from "sklearn.feature_selection" library to pre-select features, to speed up training in the later steps.

```
print("Applying features to dataset")
featureVals = self.applyFeatures(features, iis)
print("Selecting best features")
indices = SelectPercentile(f_classif, percentile=10).fit(featureVals.T, labels).get_support(indices=True)
```

After applying the features, Viola-Jones uses a series of weak classifiers and combines them according to their weights to obtain a final strong classifier. Each weak classifier looks at a single feature, I'll call it "f" in the following equation. Each weak classifier has a threshold "θ", and a polarity "p", to determine the classification of a training example according to the following equation:

$$h(x, f, p, \theta) = \begin{cases} 1 & \text{if } pf(x) < p\theta, \\ 0 & \text{otherwise} \end{cases}$$

When training the weak classifiers, each feature is a sum of the positive rectangle regions with the sum of the negative rectangle regions subtracted from them. In the following code, I combined the training part and the part of selecting the best weak classifier in the same function.

Each time a new weak classifier is selected as the best one, all the weak classifiers must be retrained since the training examples are weighted differently after each round.

Since this is a computationally expensive process, I have first sorted the weights according to the feature value that they correspond to. Then I iterated through the array of weights and computed the error if the threshold was chosen to be that feature.

To select the best weak classifier in each round, I iterated through all the classifiers and calculated the average weighted error of each one on the training dataset, and chose the classifier with the lowest error. Eventually, I obtain the best weak classifier and the best error.

```
#training the weak classifier

total_pos, total_neg = 0, 0
for w, label in zip(weights, labels):
    if label == 1:
        total_pos += w
    else:
        total_neg += w

classifiers = []
total_features = featureVals.shape[0]

for index, feature in enumerate(featureVals):
    if len(classifiers) % 1000 == 0 and len(classifiers) != 0:
        print("Trained %d classifiers out of %d" % (len(classifiers), total_features))

    applied_feature = sorted(zip(weights, feature, labels), key=lambda x: x[1])

    pos_seen, neg_seen = 0, 0
    pos_weights, neg_weights = 0, 0
    min_error, best_feature, best_threshold, best_polarity = float('inf'), None, None, None

    for w, f, label in applied_feature:
        error = min(neg_weights + total_pos - pos_weights, pos_weights + total_neg - neg_weights)
        if error < min_error:
            min_error = error
            best_feature = features[index]
            best_threshold = f
            best_polarity = 1 if pos_seen > neg_seen else -1

        if label == 1:
            pos_seen += 1
            pos_weights += w
        else:
            neg_seen += 1
            neg_weights += w

    clf = WeakClassifier(best_feature, best_threshold, best_polarity)
    classifiers.append(clf)

bestClf, bestError = None, float('inf')

for clf in classifiers:
    error = 0

    for data, label_i, w in zip(iis, labels, weights):
        correctness = abs(clf.classify(data) - label_i)
        error += w * correctness

    error = error / len(iis)
    if error < bestError:
        bestClf, bestError = clf, error
```

For the last step, I tested through all the data through the trained classifier and get the result to evaluate and as a standard for further tuning.

C. Additional experiments

Class Name	T = 1	T = 2	T = 3	T = 4	T = 5	T = 6	T = 7	T = 8	T = 9	T = 10
Train False Positive Rate	3.33%	58%	0.67%	42.33%	0.33%	38.67%	0.33%	31%	0.67%	17.67%
Train False Negative Rate	19%	2.67%	6.67%	2.33%	4%	10%	2.67%	1%	4.33%	0.67%
Train Accuracy	88.83%	69.67%	96.33%	77.67%	97.83%	80.16%	98.50%	84%	97.50%	90.83%
Train False Positive Rate	3.33%	56%	3%	38.67%	4%	32.67%	4.67%	23.33%	3.67%	11%
Train False Negative Rate	16%	4%	12.33%	7.33%	9.33%	4.67%	9.33%	8.33%	10%	6.67%
Train Accuracy	90.33%	70%	92.33%	77%	93.33%	81.33%	93%	84.16%	93.16%	91.16%

By the experiment result above, I find the result of the Ts which are multiples of two seems to be a bit strange to me. Hence, I print out the weights and the weights inside the train function of adaboost.py when “weights = weights / np.linalg.norm(weights)”.

I found that when T is an even number, there will be several weights that have not been updated, which will be equivalent to the normalized weights in the previous iteration.

Therefore, I suppose that there are just a few pictures that will cause this result when the accuracy rate is 0, and this phenomenon may be corrected by updating the weights again, which leads to a lower accuracy only when T is an even number.

D. Detect car

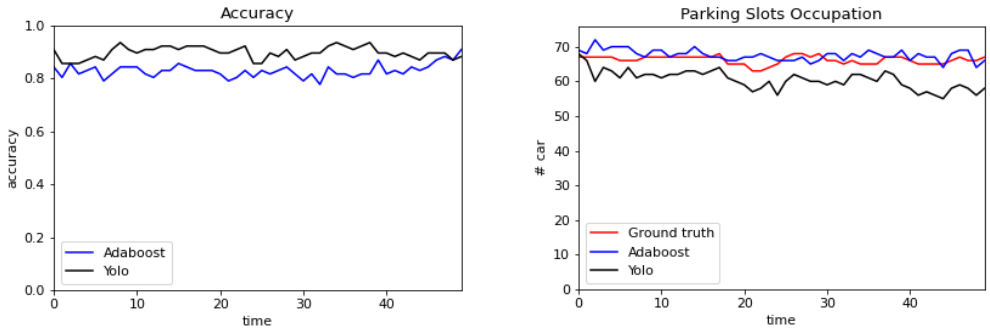
```
with open(dataPath) as f:
  lines = f.readlines()

cap = cv2.VideoCapture("data/detect/video.gif")

now_at = 0
final_result = []


while(cap.isOpened()):
  ret, frame = cap.read()

  tmp_classify_result = []

  if ret == True:

    now_at += 1
    print("we are now at frame no.", now_at)

    for i in range(int(lines[0])):
      tmp_object = list(map(int, lines[i+1].split(" ")))
      tmp = np.fromiter(tmp_object, dtype=np.int)

      cropped_frame = crop(tmp[0], tmp[1], tmp[2], tmp[3], tmp[4], tmp[5], tmp[6], tmp[7], frame)
      # cropped_frame = np.rot90(cropped_frame)
      cropped_frame = cv2.resize(cropped_frame, (36, 16))
      cropped_frame = cv2.cvtColor(cropped_frame, cv2.COLOR_BGR2GRAY)

      classify_result = clf.classify(cropped_frame)
```

```
        if classify_result == 1:
            tmp_classify_result.append(1)

            if now_at == 1:
                draw = np.array([[tmp[4], tmp[5]], [tmp[6], tmp[7]], [tmp[2], tmp[3]], [tmp[0], tmp[1]]])
                cv2.polylines(frame, [draw], True, (0,255,0), 2)
        else:
            tmp_classify_result.append(0)

    final_result.append(tmp_classify_result)
    if now_at == 1:
        frame = frame[:, :, [2,1,0]]
        plt.imshow(frame)
        plt.show()

  else:
    file = open("Adaboost_pred.txt", "w+")
    for frame_result in final_result:
        for i in range(len(frame_result)):
            file.write(str(frame_result[i]))
            if i != (len(frame_result)-1):
                file.write(" ")
        file.write("\n")
    file.close()
    break

cap.release()
```

For the fourth part, I get the data of the gif file frame by frame by cv2.VideoCapture(). In each frame, I crop the parking spaces by the crop() function. After resizing and transferring it into grey scale, I classify it by the clf.classify() function.

After classifying it, I appended the result in a list, to have the result saved into the Adaboost_pred.txt file for the comparison in the next part. Besides, if now_at is 1, which means I am at the first frame, I will draw the bounding boxes on those spaces classified with a car and plot it out by the cv2.polylines() function.

E. Difference between Adaboost and Yolov5



After testing both Adaboost and Yolov5, we can easily tell that on average, Yolov5 got better accuracy than Adaboost. Besides, we can also find the result of the parking slots occupation of the two models and the actual ground truth. Telling that Adaboost is more likely to make positive predictions, which makes it to have higher false positive rate. On the other side, Yolov5 also got higher false negative rate.