Step 1: Downloading images from videos

The function **getImages()** downloads image from a single video file or multiple video files in parallel depending on the concurrency value set by the user. It takes help of another function named **extractImageFromVideo()** which takes a file name, downloads the image if there exists a video file with that given name and returns the success of the operation (with true meaning success and vice-versa).

1. For downloading a single image, you need to call **getImages()** as below:

   getImages('1538076003')

2. For downloading multiple images in parallel, you need to call **getImages()** as below:

   file_names = ['1538076003','1538076007','1538076011','1538076015']

   getImages(file_names, True)

Step 2: Detecting car at a parking spot

- **Dependencies**:

I'm using the TensorFlow implementation of the YOLO network called as Darkflow https://github.com/thtrieu/darkflow). Please follow the below instructions to install Darkflow:

1. Clone or download this repository and extract it.
2. Go inside the folder containing all the files and run the command **pip install .**
3. Download the weights (**yolo.weights**) from https://drive.google.com/drive/folders/0B1tW_VtY7onidEwyQ2FtQVplWEU
4. Create a folder named 'weights' and put the above weight file inside it.

Note: There is still no proper implementation of tiny-yolo (version3) which is the state-of-the-art in object detection.

- **Approach**:

I tried two different approaches for detecting a car at a specified parking spot which I am describing below:

1. **Recognizing from cropped image**:

If we think logically about this problem then it would appear as if it is an object recognition task rather than object detection task. This is because since we know the particular area we want to search for, we can take a cropped portion of the parking spot from the entire image, run the state-of-the-art object recognition network (ResNet50) and see if it is able to recognize a car or not. Using this approach could increase the accuracy a lot as the object recognition networks have far better performance than the networks used for object detection (detection is relatively difficult than recognition).

But all the recognition networks need input images of some specified size (224X224 in case of ResNet50) and for that I had to scale the cropped image from around 50X50 to 224X224 which resulted into poor image quality and affected the performance of the network.

2. **Detecting using reference points and threshold value**:

Over here, I'm using a similar concept like anchor boxes in the YOLO network, where I'm taking two reference points within the parking spot- one for the top left corner and other for the bottom right corner. Running the YOLO network on the entire image and calculating L2 distance between the reference points and all the bounding boxes predicted by the network. Finally, if any one of these distances is less than some threshold value than I am predicting that a car is present at the parking spot. (Refer to Image1 for the reference points that I have taken.)

To detect if a car is present or not at the mentioned parking spot, call the function **detectCar()** as below:

detectCar('1538076003.jpg')

Bonus part (analyzing rest of the parking spot):

For this task, I have numbered parking spots (leaving ones at the extreme ends as they were often not being detected by the YOLO network) from 0-7 starting from the left-hand side of the image. Hence, the parking spot specified in this assignment gets the number 4. This also works similarly to the above procedure except that it uses reference points and threshold values for every parking spot. (Refer to Image1 for the reference points that I have taken.)

To detect if a car is present or not at the second parking spot, call the function **detectCarAtSpot()** as below:

detectCarAtSpot('1538076003.jpg', 1)

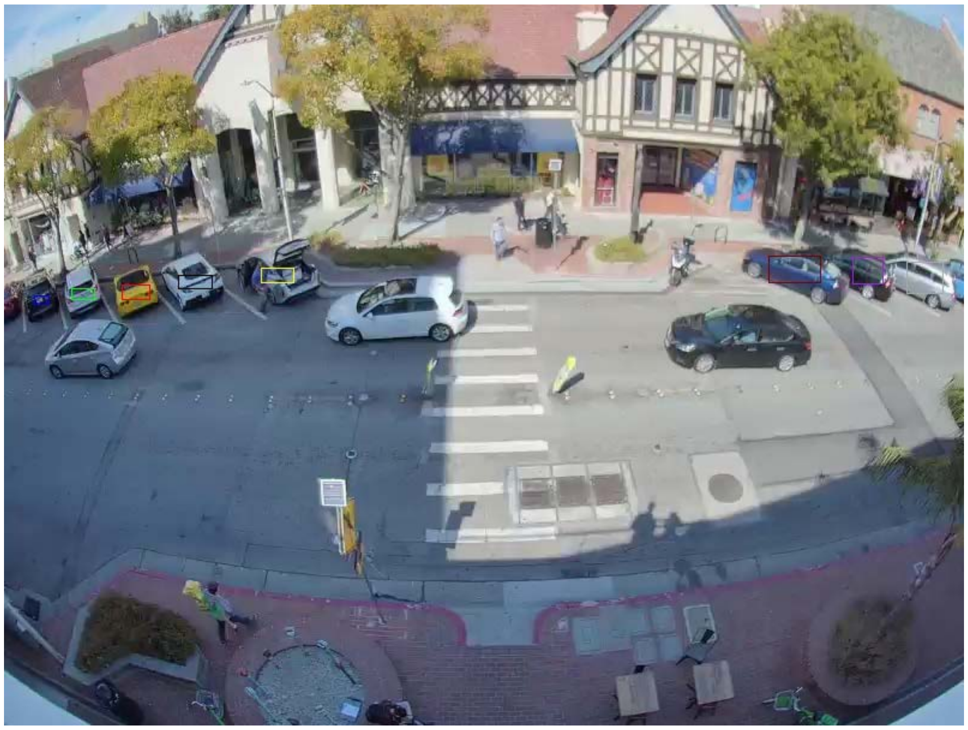

Image1: Reference points for the parking spots.

- **<u>Limitations</u>:**

  Since object detection is still one the most difficult problem in computer vision which is yet to be solved, this network has some limitations as well. It fails to detect every car in a cluttered environment, i.e., if we are looking at a huge parking area (hundreds of cars) then it won't be able to detect every car. Also, if the lightening of the image is not good (too dark or too bright) and if we have cars occluded by some other object then this also might result into failure in detection.

- **<u>Future Exploration</u>:**

  I believe that if we know specifically where we are look at, we can build a new recognition network that will take images of smaller size (equal to the cropped size) and train it to predict if a car is present or not. Moreover, this might be a good idea since we have huge amount of data to train on. Otherwise, we could use high resolution cameras so that the even after the cropped image is scaled up, it won't affect the image quality much and then we can run the state-of-the-art object recognition network on it to.

  Another approach could be to implement YOLOv3 in Tensorflow and then use the pre-trained network for doing predictions.

Step 3: Computing similarity between two cars

For finding similarity between two cars, I'm first analyzing the number of cars that were parked within the given time range and if it is greater than one (meaning the car is not there anymore but can come again in future) then I am going for template matching because we know that those cars will be rotation and scale invariant and template matching works well when these conditions hold true. So, I'm cropping out the area containing the parking spot from the images, converting the cropped image into gray scale, normalizing and computing L2 distance between them (pixel wise). Finally, if this distance is less than some threshold value then I am predicting that both of these images are of the same car.

To compare similarity between two cars, you need to call the function **compareImages()** as below:

compareImages('1538076003.jpg', '1538076007.jpg')

- **<u>Limitations & Future Exploration:</u>**

  If by finding similarity between two cars we actually mean to check if they belong to the same person or not then the above method won't work all the time. Instead what we can do to solve this problem is to get images from a high resolution camera, crop out the area containing the parking spot from the images, segment/crop out the text area, and run text classification on them. Finally, check if both the images has the same sequence of text or not. If yes, then it would mean that both of these images are of the same car.

Step 4: Put it all together!

In order to detect every car that were parked and for how long were they parked within a given time interval we need to consider four cases which are described as follow:

1. **Car detected at current time step**:

   When this is true then we need to consider if it is a new car (detected first time) or it is the car that we already saw in the previous time step. If it is a new car then will note down the current time as its starting time at the parking spot and if it is the same old car that we have already detected then we do nothing.

2. **No Car detected at current time step:**

   When this is true then we need to consider if we had a car parked at the parking spot in the previous time step or not. If yes, then we calculate its total parked time by taking difference between current time step and the time when it was detected for the first time and if no, we do nothing.

   Therefore, using the above knowledge, I'm predicting cars that were detected and for how long were they parked. Hence, in order to get this information, you need to call the function **analyzeCars()** as below:

   analyzeCars('1538076003','1538076022')

Bonus

1. **Detecting color of a car:**

   For detecting color of a car, I'm first converting the image into the HSV color scheme, cropping out a small area from the image containing the car, taking mean of the HSV values over the pixels from the cropped image and then using these average HSV values to predict different colors. (Here we are assuming the given image has a car)

   To find out color of a car, call the function **detectColor()** as below:

   detectColor('1538076003.jpg')

2. **Detecting parking spots in a random image (couldn't do it):**

   For this task, I first tried masking out white area (because the parking spot has white colored strips) from the input image then tried detecting edges and finally applied Hough transform to detect white lines. After this I tried performing pattern matching with a fixed parking sized template but since there was lot of noise in the image, made parking spot detection difficult.

   - **Future Exploration:**

     We can build an encoder-decoder network on images of parking area using convolutional neural networks where the object would be to regenerate only parking spots (just the white strips) from the original image.

Then it will detect all possible white strips (which may or may not belong to parking spot) but then we can perform a template matching over these points with a known size of the parking spot to do our predictions.

Other dependencies for the project

1. Numpy
2. OpenCV2
3. Python3