EECS 442 Final Project on Fast Pedestrian Detection

Haoyu Liu, Huan Lu, Sheng Liu
University of Michigan, Ann Arbor
1301 Beal Avenue, Ann Arbor, MI 48109-2122
haoyuliu@umich.edu, hiker@umich.edu, liusheng@umich.edu

Abstract

This project is aimed at designing a relatively fast upright pedestrian detection system used for applications like driverless car. Instead of using traditional sliding window for such a task, which is understandably slow, we tried to exploit segmentation for preprocessing images. After getting possible detection windows, we chose to use Histogram of Gradient (HoG) vector to represent each window. For the classification step, we decided to use a simple neural network.

1. Introduction

In this project, we are trying to implement an object detection system, specifically, our goal is to design a pedestrian detection system that can be used in larger applications like driverless car. Such higher level usage of our system determines the specific goal of our project, which is basically: assuming the input image to be the street view image taken from the front camera of a vehicle, process such input to detect locations where upright pedestrians is standing or walking. In other words, our project is aimed at finding upright pedestrians in an image. Previously, most of relevant systems are implemented by using a sliding window to scan the whole image, and then feed every single window to the classifier to determine whether such window contains a pedestrian. This procedure has the advantage of being secure: it could hardly miss a pedestrian in the image, however, one of the most obvious drawback of this system is that it is slow, especially when we use modern classification systems like Convolutional Neural Network, the cost of classifying an image is relatively high. Therefore, in this project, considering the real-time property of the applications that use this system, we want to explore some methods so that we can speed up such system by removing sliding window. For the scope of this project, we only detect pedestrian in static images.

2. Approach

2.1. Segmentation

One of the most important part of this project lies in detecting pedestrians without using sliding window; this leads us to think about what is the special property of the region in an image that contains pedestrian. Our intuition and answer to this question is: pedestrian in an image will usually introduce inconsistency so that it will create edges. Based on this answer, the first module of our system would be segmentation.

By using segmentation, we hope we can find all possible positions where a pedestrian could be. For illustration, we use the following example image in the rest of our description.

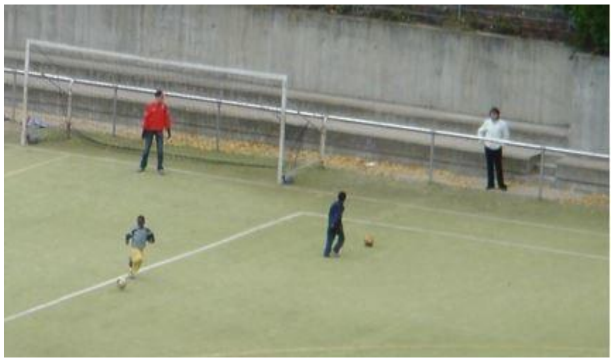
Figure 1: Example image.

First of all, because of the inconsistency, we perform edge detection on an image (transform original and possibly colored image to grayscale). After the edge detection, we will get a binary image, in which the pixel values of edges will be 1 and the other part will be 0.

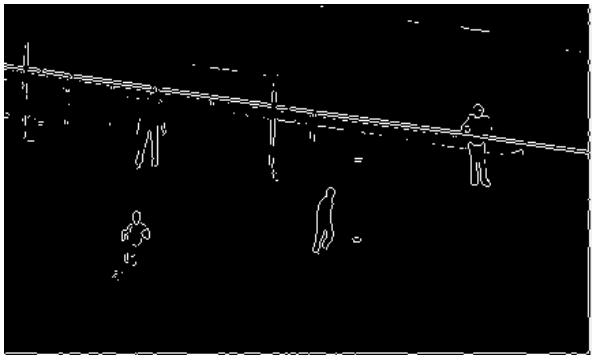
Figure 2: Image after edge detection.

For the next step, initially we consider the most ideal case, in which the pedestrian is standing in front of a white wall, after the edge detection, the contour of that pedestrian will be shown, and such contour will be closed, i.e. it will circle out a specific region in the image, and that region would exactly be a pedestrian. For example, see the contour at the center of Figure 2. In order to utilize the closing property, we will perform a binary image operation called "imfill", which will fill a closed region in a binary image with white pixel values. In this ideal case, the resulting binary image will give us the exact detection of a pedestrian: white pixel region in the binary image denotes the pedestrian location in the original image. To mark such region in the original image, we decide to use the centroid of the white region (also a connected component) to locate the coordinates of this region and then form a fixed sized rectangular window around that center location, and finally extract that window from original image and feed this window into the classifier; if the classifier returns positive then we create a rectangle on that location in the output image. Finally, the output of our system will mark all pedestrians in the image.

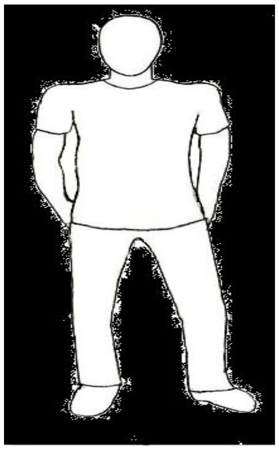
Figure 3: Ideal example.

This is the basic idea of the segmentation. However, in reality, we cannot assume the edges will be automatically closed after the edge detection step. To address this problem, before hole-filling, we choose to perform a dilation step, which will essentially enlarge the boundaries of white pixels in the image. The intuition for this step is that we hope it could help us connect the edges to form closed region for us to better fill the holes in the image.

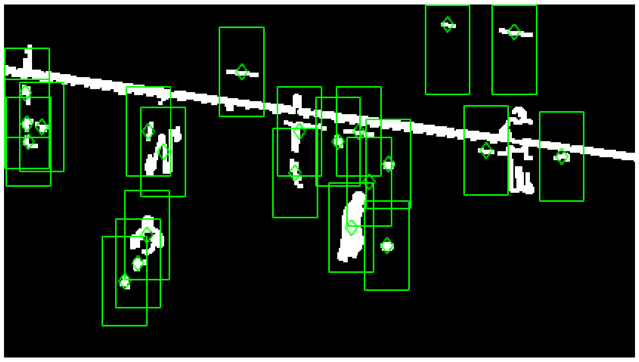
Figure 4: Image after dilation and filling, green diamonds representing the centroids of each component and green rectangle representing possible detection window.

Unfortunately, this method will bring another problem, it will possibly connect many components together such that the centroid of that new large component (after filling the hole) may not accurately represent the location of the pedestrian, such that we might lose a detection (As shown in Figure 4, the detection for the person on the right part of the image is inaccurate, because of its connection to the long pole in the figure). To fix this problem, we introduce another function that will help us split over-large component, such that the centroid will represent the location of a component more accurately. The code for this function is fairly simple: for the whole binary image after dilation, we set rows and columns with some certain intervals to be zeros. As a result, the components in the image will be broken into many relatively small components.

Another optimization is from the following observation: as can be seen from Figure 4, if we form windows around these centroids of components, the windows will have many overlapping. Thus, if the centroids are close enough, the windows corresponding to them will be basically the same. In general, we do not want to waste time to do classification on similar windows, so we choose to add a step similar to 'non-maximum suppression': for each window formed around a centroid, if such window contains other centroids, we compare the 'area'(which is the same as the number of white pixels in this window) of this window and that of the windows corresponding to the centroids in this window; if the area is not the maximum (or we made a relaxation that if it is not the top two largest area), we simply discard this centroid in the output. After this step, we can obviously and reasonably reduce the number of windows thus increase the efficiency of the system.

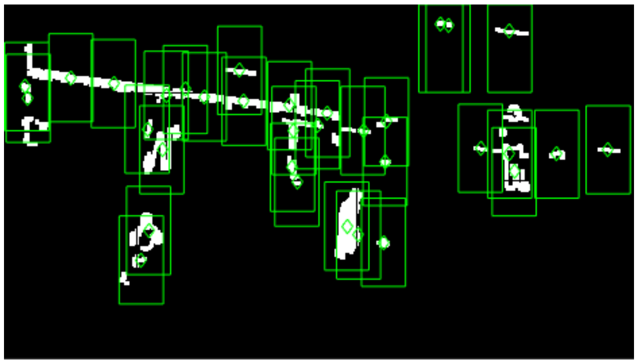
Figure 5: 'Chopped' Image and detections.

As can be seen from Figure 5, after the 'chopping' and non-maximum suppression step, the number of components increased, and the influence of large components will no longer exist, which can be demonstrated by the detection of the pedestrian on the right part of the image.

Then, we mark the regions in the original image using rectangles and send these rectangular windows to the next module.

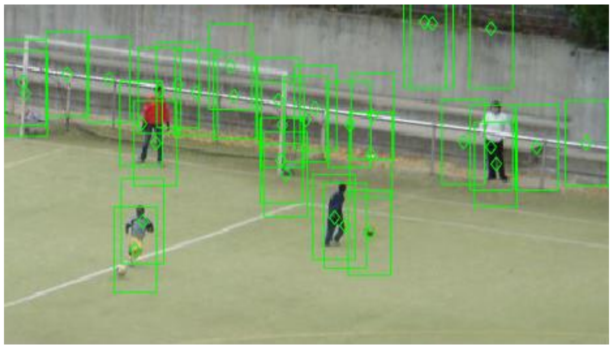
Figure 6: Possible detections on original image.

The above analysis looks reasonable but it is basically targeted to the given feature of that example image. In particular, one of the most important assumptions is that the pedestrian in that image is in similar scale, so that the sizes look the same. However, this might not be true, especially for front camera of a car. Consider the following image from the Full Image Pedestrian Dataset.

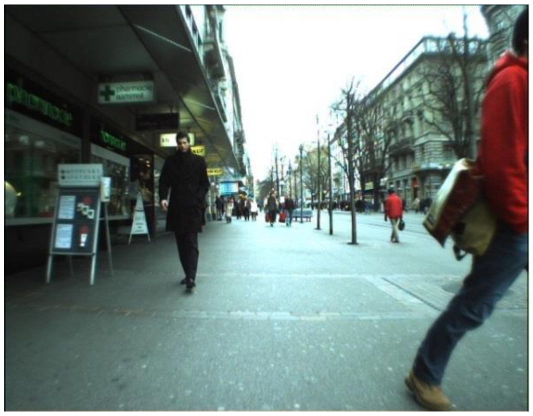
Figure 7: Another example image of pedestrian detection.

In this image, we could find pedestrians at multiple scales, so that if we stick to use a fixed window size to detect them, it might miss some cases.

The solution for this problem is straightforward, we could do a resize. The key choice here is: resize the window or resize the image. According to our analysis, if we only use larger windows on the original detected centroids, the result will not improve significantly because the detection points remain the same. However, we noticed that for larger pedestrian in an image like Figure 7, it is hard for the segmentation module to make a closed contour because of the relatively high resolution of the figure makes the contour more complex. Therefore, we should resize the original image to reduce such resolution and make our modules more applicable. Then, in our implementation, instead of finding pedestrians at only one scale, we add another resizing branch: in order to detect possible pedestrian at a larger scale, we resize the original image to a smaller size, which is similar to the size of the windows we use, and then perform the same sequence of operations as above. Finally mark larger windows at the original image, which is illustrated by Figure 8. For this project, we did the resizing in a reasonable range.

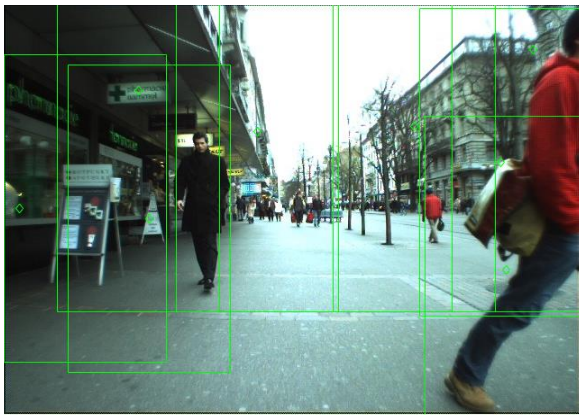
Figure 8: Possible detections made by the resizing branch.

With these detections, we have to determine whether the contents in these windows are pedestrians. Initially the idea for this process is to do classification for the binary image, which is essentially classifying the contour. In some case, this kind of makes sense since the contour can be very informative; however, after our previous steps, especially after the dilation step, the components in binary image can be something like a whole chunk with an arbitrary shape which would make the classification very ineffective. Having acknowledged the drawback of binary image, it would be easy for us to utilize the original colored image. In this project, the object to be classified is the window in the original image.

2.2. Feature description

The next module of our system is an intermediate module. In order to classify the window more effectively and efficiently. We borrowed the idea from N. Dalal and B. Triggs to use Histogram of Gradient (HoG) descriptor to describe each window. For fix-sized window, the HoG descriptor will give a vector of fixed length. In the final step, we send this vector into the classifier. The reason for this intermediate step is that first, HoG descriptor has been proved to be a good descriptor for pedestrian detection, and second, instead of putting a complete window into the classifier, HoG can reduce the dimension thus reduce the computational complexity for our system, making our system even faster.

The HoG represents the gradient majority in each part of the image. So the feature vector can very effectively represent the shape of objects in an image. To compute the HoG feature vector, first compute the gradient along the x direction and y direction, by convolving with $[-1 \quad 0 \quad 1]$

and $\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$ respectively. Then we can compute the gradient by taking $angle = \arctan(\frac{dy}{dx})$.After we got the gradient for each pixel, we calculate the histogram of gradient in each cell. Cell is a sub-area of the image. We divide an image into different cells. We compared different cell size and finally choose size 8×8 because it produces the best result. Because the angle is a continuous value, we need to set a range for each value in the histogram. We first convert the angle $[180°, 360°)$ into$[0°, 180°)$, now we have the angle value from$[180°, 360°)$. Then we divide the value into several binnings. We choose binning of 9, so each binning has range of 20°. Then for all the pixels in one cell, we add up the magnitude of gradient values for each binning direction. Now we have the histogram for one cell. After repeating this step for all the cells, we applied another step to normalize the magnitude. Because the lighting condition could be very different in different part of the image, so we normalize the magnitude of histogram in a certain block. A block contains several cells, we first convert the block into a column vector, compute the 2-norm of this vector, then normalize the vector in the block with this 2-norm value. We choose the block size of 2×2. We use overlapped block in our implementation, so two neighbor blocks will have some cells in common. The overlap stride is set to half the size of block size, which means each two neighbor block will have a half overlapped with another. That is good because one cell will be normalized by different background value generated by different block. So this kind of value is more robust. Finally, after normalizing all the blocks, we convert the blocks into a single column vector as the output, which is used as HoG feature vector. The figure below shows a visualization of our HoG vector in each cell. The value displayed in the figure is not normalized.

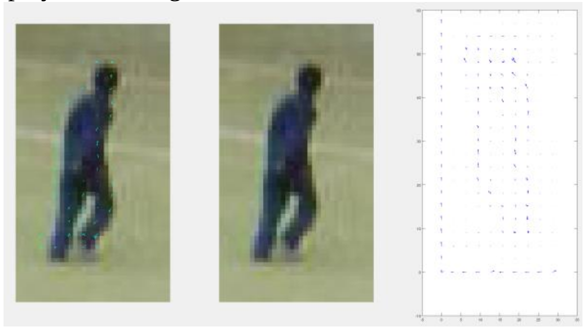
Figure 9: HoG feature with given image

2.3. Classification

2.3.1 Neural Network

The final module of our system is the classifier. Having recognized the power of neural network, we are very willing to utilizing such power. Nevertheless, considering the limit of the size of the dataset, we also have to limit the scale of our network. In our project, we decided to use a simple neural network with 1 linear layer.

We have tried different window size hoping to find the one best capturing the common appearance of pedestrian. For each window size, to which we first resize the training images, then get the HoG features of the processed image. Next step, we build up a neuron network taking the HoG features as inputs, output the label 2 if it's a pedestrian, 1 otherwise. For different window size, we can get different lowest cross-entropy loss, so we need to set different threshold for stopping training process. In general, the larger window size provides larger HoG feature vector, and yields lower cross-entropy loss as well, but due to the small number of training data we have, it's more likely to get overfitted model with larger window size. See the following result for different window size.

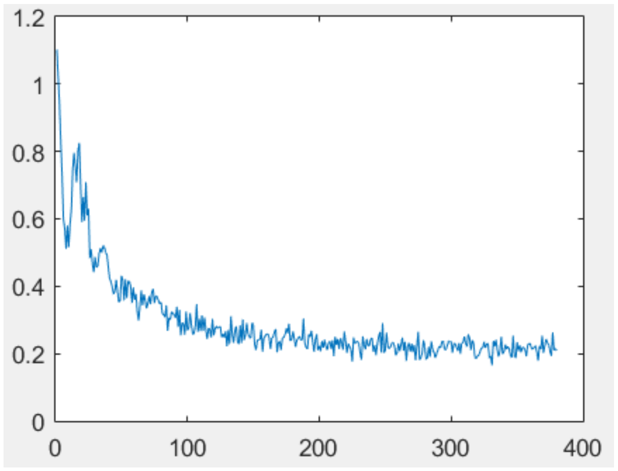

Figure 10: Training loss for window size 74*34

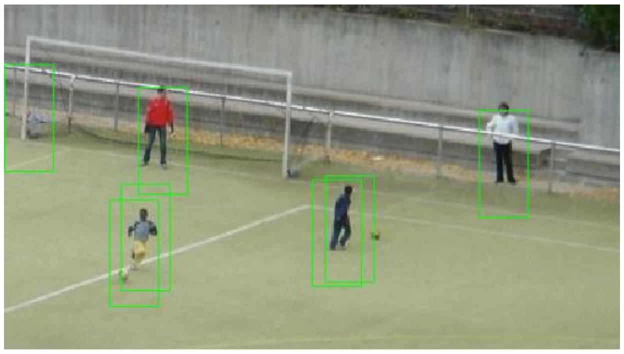
Figure 11: Result for window size 74*34

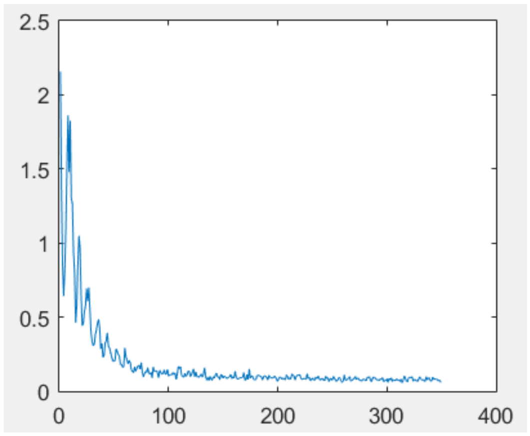

Figure 12: Training loss for window size 160*90

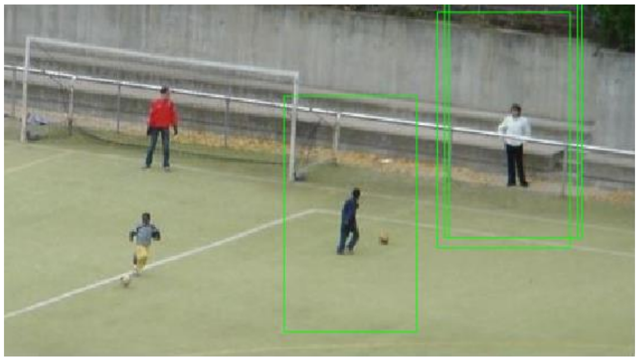
Figure 13: Result for window size 160*90

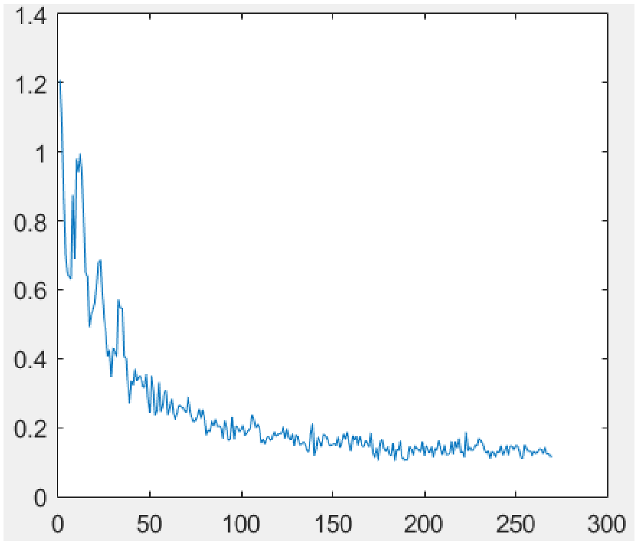

Figure 14: Training loss for window size 160*90

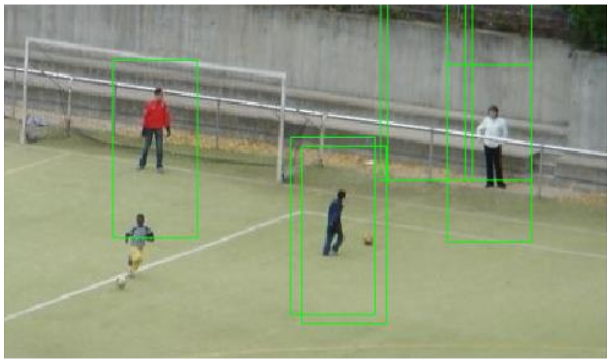
Figure 15: Result for window size 120*60

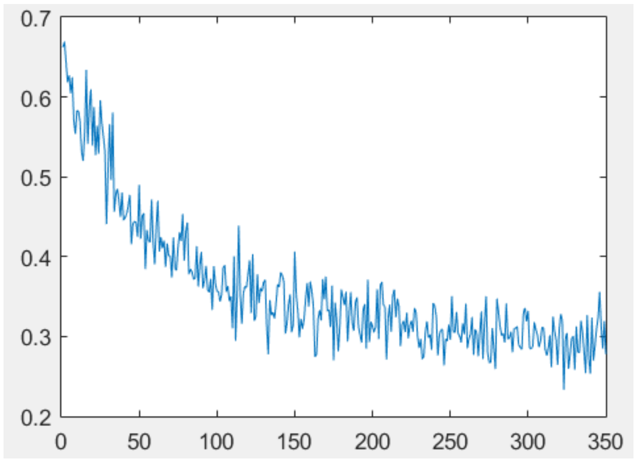

Figure 16: Training loss for window size 60*30

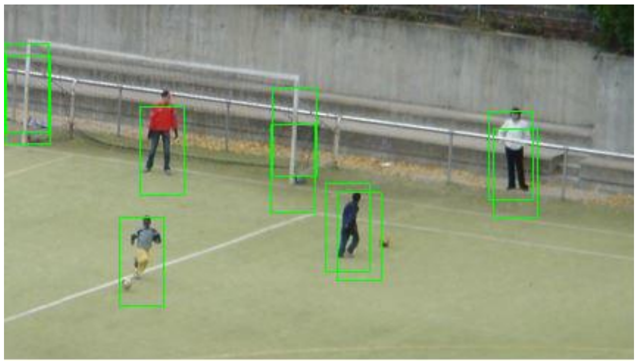
Figure 17: Result for window size 60*30

After comparison among many different window sizes, we choose the size 74*34.

For this windows size, the HoG feature vector is 432*1. The structure of the neuron network is: flatten, linear, softmax.

2.3.2 Support Vector Machine

Another classifier we used is called Support Vector Machine, or SVM in short. We first extract the HoG feature vector for all the training data, and use these labeled data to train our SVM. Then we use the classifier to classify each output window into human or non-human. We use grid search algorithm to select the optimal parameter for SVM, and finally got the parameter $\sigma = 4$ and $C = 1024$ for the best performance, with error rate only 2.7%.

3. Implementation

As from the previous section, our project can be divided into three modules.

The first module is getting possible detections by segmentation. For this module, everything except the edge detection function (which we used the MATLAB version) is our own work, including building the structure, image dilation, splitting component, non-maximum suppression, image resizing and marking.

The second module is HoG feature extraction, we wrote our own version of HoG feature extractor, but it gives only a 70% accuracy in the experiment, so we finally decided to use the MATLAB build in function for better performance.

The last module is the classification, we try different structures of neuron network and choose different window size to train the network. After finding the most applicable one, we apply it to classify images and output the target image. We implemented the SVM by ourselves, with one function SMO borrowed from Diego Andres Alvarez[5] .

4. Experiments

4.1. Experiment on neural network

The first experiment we did is to continue the usage of our example image. In this experiment, we did not scale the image so that the detection is on a fixed scale. And for detection, we only use a window size of 74*34.

As from the example used above, we tested our system using that image and we have the resulting image as follows, in which the rectangle denotes a detected pedestrian.

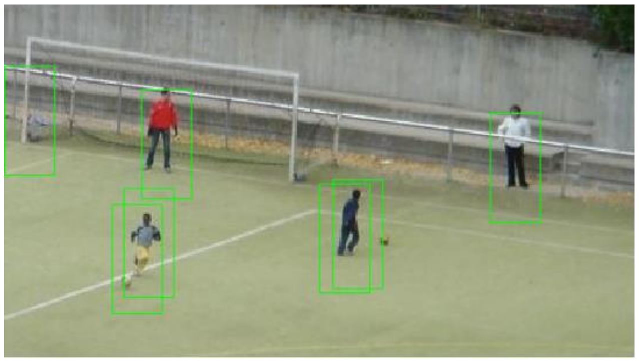
Figure 18: Result for fixed scale

In the second set of experiments, we introduced the scaling of images. Other than keep the original image size, we scale the image to have the height as 1, 4/3, 5/3, 2, 7/3, 8/3, 3 times of the window height, and use detect pedestrian for all these images, in this way, we can almost find all pedestrian in the image even though they have different size in original image. In the following figures, we can see that two of them can sufficiently contain all of the pedestrians.

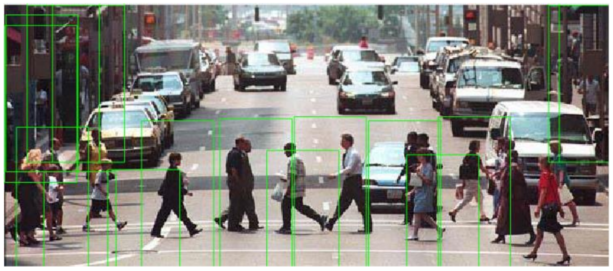
Figure 19: Result for scaling algorithm (1)

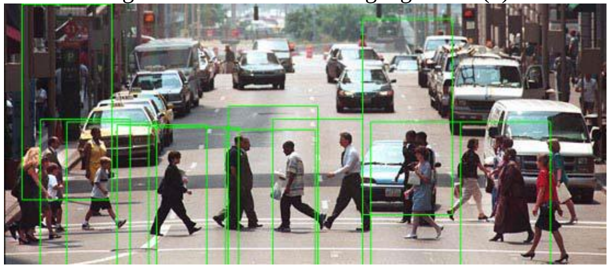
Figure 20: Result for scaling algorithm (2)

4.2. Experiment on SVM

In our SVM version implementation. The result is slight better. We can see from the output image Figure 21 below, all four humans in this image are successfully identified and no false positive result. But in another image Figure 22, only hidden human in the corners are detected, those obvious walking humans are not successfully detected.

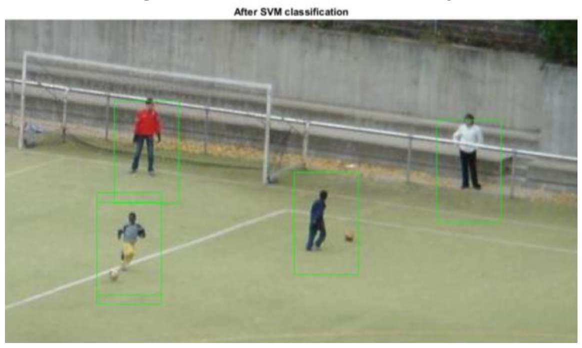

Figure 21: SVM detected soccer image

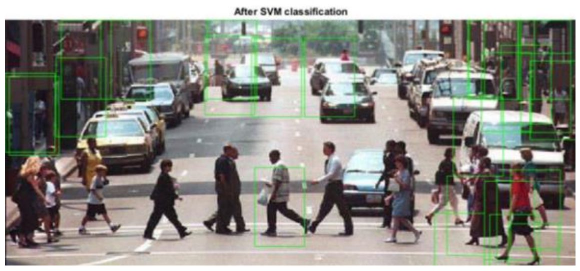

Figure 21: SVM detected street image

One possible explanation to this result is that SVM is very sensitive to the scale of human in the input window. Because SVM treat every single element in the input vector as an individual, so the shape of human is learned by the classifier. But the shape is a fixed scale compared to the input window. So when the detected human is larger than the window size, SVM can not successfully identify the human part object.

5. Conclusion

As can be seen from the experiments, generally our system can accurately detect pedestrians in the image. One obvious problem of our system is that it has some unreasonable false positives, meaning that the classifier tends to mark non-pedestrian as human, and this happens frequently. It is easy to figure out such a problem can be tracked down to the classifier(s). Due to the lack of a generalized and large-sized pedestrian dataset, it is hard to train our network or SVM sufficiently, which will finally result in the inaccuracy of the classification. By 'generalized', we mean the property of the training images: they are usually in the same 'mode' such that the training will 'trap' our classifier to such a mode so that it cannot generalize well.

Also, though we have eliminated the step of sliding window, the binary segmentation step cannot always perform as we hoped it to, i.e. segment a complete pedestrian out. Usually, it could only get part of a pedestrian, which will make the classification harder. And for the resizing step, we only scale the image by a factor and use the previous fix-sized window to do detection, which cannot cover the case in which the pedestrian figure may have a different size.

A better approach is to use dynamic window size. So for each object we detected in the input image, we first find the border of this object, then form a window which covers the entire part of this object. In this way, the window passed to the classifier is more generalized. So we could get better result when classifying those images.

In this project, we designed a complete system for pedestrian detection. During the design phase, we considered and solved many unseen problems. One of the most important lessons we have learned from this process is how to correctly use the information from an image for a vision system. For example, in order to extract possible pedestrians from an image, we can use the gradient information in the image(i.e. edges), rather than naively using a sliding window over the original colored image, which can reduce the complexity; in order to utilize the gradient information, we choose to perform operations on binary images, which is much simpler and faster; on the other hand, in order to classify whether a window contains a pedestrian, binary image can no longer provide sufficient information but the original image can. The methods we use for different modules are alternating between original image and binary image based on demands from higher level objectives. In other words, if you want simpler information such as edges, it is enough to use only grayscale or binary image, however, if you want to determine whether a sub-image contains a human, you have to resort to original colored image for more information. Having this understanding will be very helpful for designing vision system.

Other take-away from this project are understanding some operations on binary image, learning HoG features, SVM classifier and enhancing the application of neural networks.

References

[1] Stephen Gould, Tianshi Gao, Daphne Koller. Region-based Segmentation and Object Detection, NIPS, 2009.

[2] Navneet Dalal and Bill Triggs. Histograms of Oriented Gradients for Human Detection. CVPR, 2005

[3] Dataset from Cornell University, CS 4670 Project 5 Object Detection. http://www.cs.cornell.edu/courses/cs4670/2013fa/projects/p5/index.html

[4] Olivier Chapelle. Support Vector Machines for Image Classification. 1998

[5] Diego Andres Alvarez, Implementation of the Sequential Minimal Optimization training algorithm for Vapnik's Support Vector Machine, 2002