MTRE 4490 – Machine Learning for Robot Perception
Project – Cone Detection and Position Estimation

Project goal

This project aims to identify yellow cones from a VEX Robotics competition and estimate their position relative to a corner of the field as in Figure 1. Training data is not supplied; the original image in Figure 1 is given from which examples of cones and not cones must be extracted. A support vector machine or Naïve Bayes classifier is then trained on this data and a sliding window search applied to the test image. Non-maximum suppression is used to help eliminate false positives. Finally a perspective transform is applied to obtain a "birds-eye" view from which cone positions are estimated.

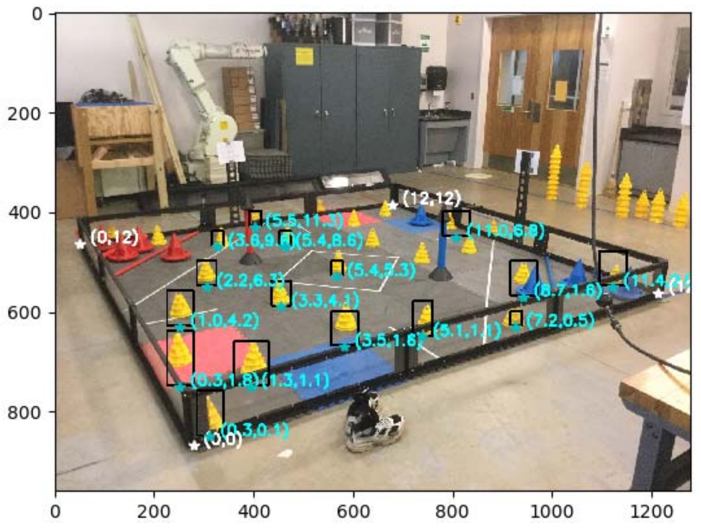

Figure 1: Results of the project where most of the yellow cones are correctly identified and their position (in feet) marked relative to the (0,0) corner of the field.

Project submission guidelines

Create a folder named LastNameProject03 containing all necessary source code files, the supplied trainingImage.jpg file, pickle files for the normalization scaler and classifier, as well as the training images organized in subfolders as desired. Source code files will include makeData.py and trainClassifier.py, and LastNameProject02.py which loads the test image, executes the sliding window search, performs non-maximum suppression, and labels cone positions using the perspective transform.

Classifier training

Create data to train the classifier by identifying areas in the supplied trainingImage.jpg file. One approach is to record where the user clicks in the image using `matplotlib.pyplot.ginput`. Use `cv2.resize` to force all training images to the same size before saving them to disk. The training images for cone and non-cone classes must be included in the project submission, as is the code `makeData.py` used to create them.

Classifier training is performed by executing another file trainClassifier.py, which uses `glob.glob` to obtain a list of training images to load. For each image, compute relevant histogram of oriented gradient (HOG) and color histogram features to feed the classifier. Normalize each feature to zero mean and unit standard deviation with `sklearn.preprocessing.StandardScaler`. Use scikit-learn to implement either a support vector machine or Naïve Bayes classifier based on the normalized features extracted from training images. Save the scaler and resulting classifier `pickle.dump` to be read by the testing script in LastNameProject03.py file.

Sliding window search on the test image

The LastNameProject03.py file should load a test image and apply the classifier using a sliding window search. In the submission, load trainingImage.jpg to demonstrate the required performance on the training image, and the instructor will manually alter the code to instead check the test image. The best performance will likely require searching at multiple scales, but keep execution time in LastNameProject03.py to a minimum, i.e. no more than one minute on an Intel Core i7-6820HQ CPU at 2.7 GHz and 16 GB RAM.

Non-maximum suppression with heat maps

Sliding window searches generally result in many overlapping detections such as in Figure 2, which ideally are combined into a single detection using the concept of non-maximum suppression (NMS). This essentially means that all but the "best" detections should be eliminated. A simple approach is to surround all overlaps into a single bounding box, such as the blue shaded area in Figure 2. Many classification algorithms return not only a bounding box but a confidence in the classification as well. Another simple approach is to suppress all but the highest confidence classification for overlapping boxes.

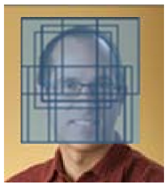

Figure 2: Sliding window search for face detection with numerous overlapping detections and a simple approach (shaded blue) for combining overlaps.

The use of thresholded heat maps is another NMS technique with the additional benefit of rejecting many false positives. The heat map is generated by beginning with a 2D array the same size as the image filled with zeros. Heat map pixel values are incremented for every bounding box containing that pixel, i.e. a pixel would have a value 5 if five bounding boxes overlap on that pixel. Pixels with high heat map values are more likely to be actual positives, whereas those with low values (such as a single, non-overlapping detection) are likely false positives. Figure 3 shows a heat map both before (left) and after applying a threshold (right) where all heat map values below a particular threshold are suppressed by setting them to zero. Remaining contiguous pixels are then grouped and enclosed with bounding boxes.

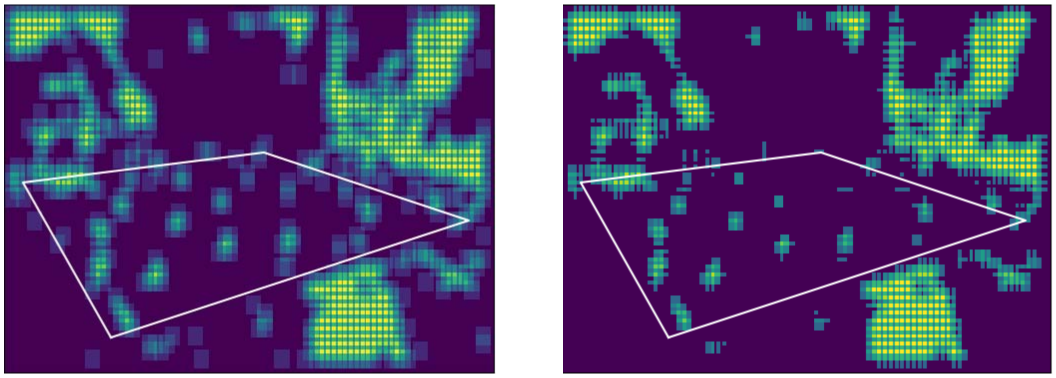

Figure 3: Heat map for sliding windows predicting a cone in its original form (left) and after threshold (right). White lines indicate region of the image containing the field.

The SciPy package, which includes Matplotlib and NumPy, contains a function `label` which takes a 2D array as input, and labels clusters of contiguous pixels. It returns a tuple where the first element is another 2D array where non-zero pixels in the input image are replaced with the cluster number. The second element in the tuple is the number of clusters. The code

```
import matplotlib.pyplot as plot
from scipy.ndimage.measurements import label
labels = label(heatMap)
plot.figure(4)
plot.clf()
plot.imshow(labels[0])
```

generates Figure 4 (left) using the heat map from Figure 3 (right). By looping through all the clusters in labels, bounding boxes are determined by enclosing all pixels of a given cluster inside a rectangle as in Figure 4 (right). Drawing rectangles is simple using `cv2.rectangle`. As an extra rejection of false positives, note that clusters too small or too large are also suppressed. Detections falling outside the field will also be eliminated in the following section.

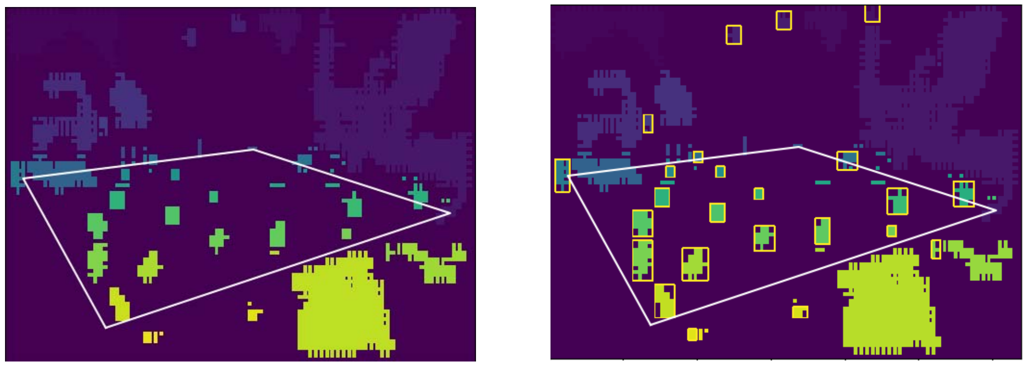
Figure 4: Heat map labeled where each cluster is designated with a different color (left) where medium-sized clusters are indicated with bounding boxes (right). White lines indicate region of the image containing the field.

Perspective transform

The pinhole camera model illustrated in Figure 5 maps point *P* in *X*,*Y*,*Z* world coordinates to point *p* in image coordinates *x*,*y* for a given camera focal length *f*. This model predicts where any point *P* in space would appear in the image, but unfortunately no one-to-one mapping exists to know from which point *P* any arbitrary *p* originated (all points on the line defined by *P* and *p* are possible). For this reason, it is impossible to know the position in space for any given image pixel without additional information.

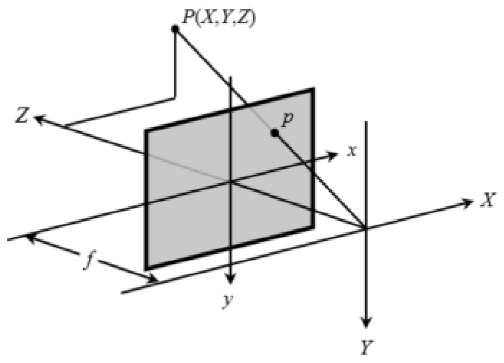

Figure 5: Pinhole camera model projecting global point *P* to image point *p*.

The perspective transform applies the pinhole camera model using four points in the image identified as representing points on a plane in global space. All image pixels are then transformed assuming they lie on this plane. The most common use of the perspective transform is to transform an image to its “birds-eye” view looking straight down on top of it. Figure 6 (left) is the original image where the four white starred points identify source points representing the plane of the ground, with Figure 6 (right) as the transform mapping the white source points to the destination magenta points. The OpenCV function `cv2.getPerspectiveTransform`[1] takes as input arguments both the source and destination points, while returning a transformation matrix which can be applied to the original image using `cv2.warpPerspective`. Request the user to click on the four corners of the field to define the white source points in Figure 6 (left) using `matplotlib.pyplot.ginput`. In Figure 6 (right) notice how the square tiles are accurately transformed, as are the white taped lines since they all lie on the plane defined by the source points. Conversely, tops of cones, the field boundary wall, and the red and blue towers are distorted since they do not lie on the plane. For this reason, the perspective transform locates the bottom of cones relatively accurately, while cone tops are not. Transforming points at the bottom center of bounding boxes will locate cones well in the plane of the field (see Figure 1). With the fact that the field is composed of 2 ft × 2 ft tiles for an overall size of 12 ft × 12 ft, the pixel location in the transformed Figure 6 (right) image translates directly to vertical and horizontal positions within the field.

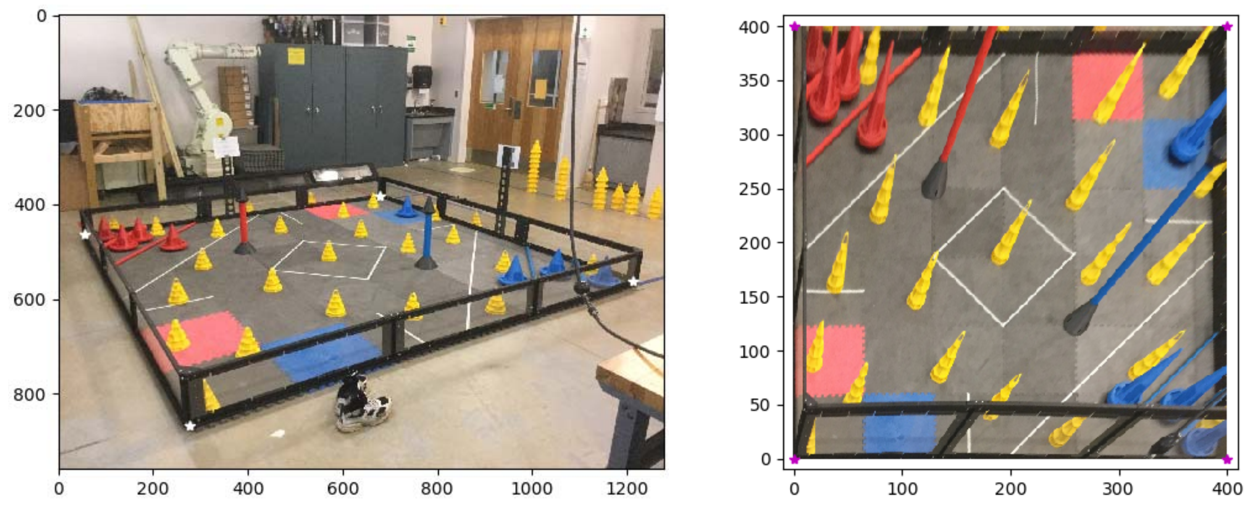

Figure 6: Perspective transform applied to the training image (left) using the white source points, resulting in the transformed image (right) with the magenta destination points.

The `cv2.warpPerspective` function transforms an entire image, but does not work on individual points or groups of points. Given the transformation matrix M, an original point (x, y) and its transformation (x', y') are related by

$$\begin{bmatrix} cx' \\ cy' \\ c \end{bmatrix} = M \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Transforming n number of points, such as the bottom centers of each bounding box in Figure 1, can be accomplished with the single matrix multiplication

$$\begin{bmatrix} c_1x_1' & c_2x_2' & \cdots & c_nx_n' \\ c_1y_1' & c_2y_2' & \cdots & c_ny_n' \\ c_1 & c_2 & \cdots & c_n \end{bmatrix} = M \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \\ 1 & 1 & \cdots & 1 \end{bmatrix}$$

Any of the transformed points (x_i', y_i') whose values fall outside the magenta destination points in Figure 6 (right) are guaranteed to be outside the field and can therefore be suppressed. Once the position of each non-suppressed bounding box is determined, `cv2.putText` can write it on the image for display as in Figure 1.

[1] http://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html

Grading rubric

- (10 points) Create training images for cones and non-cones
- (10 points) Calculate HOG and color histogram features
- (5 points) Normalize feature values to zero mean and unity standard deviation
- (5 points) Train the classifier
- (10 points) Implement sliding window search
- (10 points) Develop heat map and apply threshold
- (10 points) Draw bounding boxes on remaining detections
- (10 points) Display perspective transform of the original image
- (10 points) Display accurate positions next to bounding boxes
- (10 points) trainingImage.jpg performance: at least 10 correct detections with no false positives
- (10 points) Test image performance: at least 5 correct detections with no more than one false positive