1. HOG

June 23, 2018

```
<br><br>
Histograms of Oriented Gradients (HOG)
<br><br><br>
```

Introduction

As we saw with the ORB algorithm, we can use keypoints in images to do keypoint-based matching to detect objects in images. These type of algorithms work great when you want to detect objects that have a lot of consistent internal features that are not affected by the background. For example, these algorithms work well for facial detection because faces have a lot of consistent internal features that don't get affected by the image background, such as the eyes, nose, and mouth. However, these type of algorithms don't work so well when attempting to do more general object recognition, say for example, pedestrian detection in images. The reason is that people don't have consistent internal features, like faces do, because the body shape and style of every person is different (see Fig. 1). This means that every person is going to have a different set of internal features, and so we need something that can more generally describe a person.

Fig. 1. - Pedestrians.

One option is to try to detect pedestrians by their contours instead. Detecting objects in images by their contours (boundaries) is very challenging because we have to deal with the difficulties brought about by the contrast between the background and the foreground. For example, suppose you wanted to detect a pedestrian in an image that is walking in front of a white building and she is wearing a white coat and black pants (see Fig. 2). We can see in Fig. 2, that since the background of the image is mostly white, the black pants are going to have a very high contrast, but the coat, since it is white as well, is going to have very low contrast. In this case, detecting the edges of pants is going to be easy but detecting the edges of the coat is going to be very difficult. This is where **HOG** comes in. HOG stands for **Histograms of Oriented Gradients** and it was first introduced by Navneet Dalal and Bill Triggs in 2005.

Fig. 2. - High and Low Contrast.

The HOG algorithm works by creating histograms of the distribution of gradient orientations in an image and then normalizing them in a very special way. This special normalization is what makes HOG so effective at detecting the edges of objects even in cases where the contrast is very low. These normalized histograms are put together into a feature vector, known as the HOG descriptor, that can be used to train a machine learning algorithm, such as a Support Vector Machine (SVM), to detect objects in images based on their boundaries (edges). Due to its great success and reliability, HOG has become one of the most widely used algorithms in computer vison for object detection.

In this notebook, you will learn:

- How the HOG algorithm works
- How to use OpenCV to create a HOG descriptor
- How to visualize the HOG descriptor.

1 The HOG Algorithm

As its name suggests, the HOG algorithm, is based on creating histograms from the orientation of image gradients. The HOG algorithm is implemented in a series of steps:

1. Given the image of particular object, set a detection window (region of interest) that covers the entire object in the image (see Fig. 3).

2. Calculate the magnitude and direction of the gradient for each individual pixel in the detection window.

3. Divide the detection window into connected *cells* of pixels, with all cells being of the same size (see Fig. 3). The size of the cells is a free parameter and it is usually chosen so as to match the scale of the features that want to be detected. For example, in a 64 x 128 pixel detection window, square cells 6 to 8 pixels wide are suitable for detecting human limbs.

4. Create a Histogram for each cell, by first grouping the gradient directions of all pixels in each cell into a particular number of orientation (angular) bins; and then adding up the gradient magnitudes of the gradients in each angular bin (see Fig. 3). The number of bins in the histogram is a free parameter and it is usually set to 9 angular bins.

5. Group adjacent cells into *blocks* (see Fig. 3). The number of cells in each block is a free parameter and all blocks must be of the same size. The distance between each block (known as the stride) is a free parameter but it is usually set to half the block size, in which case you will get overlapping blocks (*see video below*). The HOG algorithm has been shown empirically to work better with overlapping blocks.

6. Use the cells contained within each block to normalize the cell histograms in that block (see Fig. 3). If you have overlapping blocks this means that most cells will be normalized with respect to different blocks (*see video below*). Therefore, the same cell may have several different normalizations.

7. Collect all the normalized histograms from all the blocks into a single feature vector called the HOG descriptor.

8. Use the resulting HOG descriptors from many images of the same type of object to train a machine learning algorithm, such as an SVM, to detect those type of objects in images. For example, you could use the HOG descriptors from many images of pedestrians to train an SVM to detect pedestrians in images. The training is done with both positive a negative examples of the object you want detect in the image.

9. Once the SVM has been trained, a sliding window approach is used to try to detect and locate objects in images. Detecting an object in the image entails finding the part of the image that looks similar to the HOG pattern learned by the SVM.

Fig. 3. - HOG Diagram.
Vid. 1. - HOG Animation.

2 Why The HOG Algorithm Works

As we learned above, HOG creates histograms by adding the magnitude of the gradients in particular orientations in localized portions of the image called *cells*. By doing this we guarantee that stronger gradients will contribute more to the magnitude of their respective angular bin, while the effects of weak and randomly oriented gradients resulting from noise are minimized. In this manner the histograms tell us the dominant gradient orientation of each cell.

2.0.1 Dealing with contrast

Now, the magnitude of the dominant orientation can vary widely due to variations in local illumination and the contrast between the background and the foreground.

To account for the background-foreground contrast differences, the HOG algorithm tries to detect edges locally. In order to do this, it defines groups of cells, called **blocks**, and normalizes the histograms using this local group of cells. By normalizing locally, the HOG algorithm can detect the edges in each block very reliably; this is called **block normalization**.

In addition to using block normalization, the HOG algorithm also uses overlapping blocks to increase its performance. By using overlapping blocks, each cell contributes several independent components to the final HOG descriptor, where each component corresponds to a cell being normalized with respect to a different block. This may seem redundant but, it has been shown empirically that by normalizing each cell several times with respect to different local blocks, the performance of the HOG algorithm increases dramatically.

2.0.2 Loading Images and Importing Resources

The first step in building our HOG descriptor is to load the required packages into Python and to load our image.

We start by using OpenCV to load an image of a triangle tile. Since, the `cv2.imread()` function loads images as BGR we will convert our image to RGB so we can display it with the correct colors. As usual we will convert our BGR image to Gray Scale for analysis.

```
In [ ]: import cv2
        import numpy as np
        import matplotlib.pyplot as plt


        # Set the default figure size
        plt.rcParams['figure.figsize'] = [17.0, 7.0]

        # Load the image
        image = cv2.imread('./images/triangle_tile.jpeg')

        # Convert the original image to RGB
        original_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        # Convert the original image to gray scale
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```python
# Print the shape of the original and gray scale images
print('The original image has shape: ', original_image.shape)
print('The gray scale image has shape: ', gray_image.shape)

# Display the images
plt.subplot(121)
plt.imshow(original_image)
plt.title('Original Image')
plt.subplot(122)
plt.imshow(gray_image, cmap='gray')
plt.title('Gray Scale Image')
plt.show()
```

3 Creating The HOG Descriptor

We will be using OpenCV's `HOGDescriptor` class to create the HOG descriptor. The parameters of the HOG descriptor are setup using the `HOGDescriptor()` function. The parameters of the `HOGDescriptor()` function and their default values are given below:

`cv2.HOGDescriptor(win_size = (64, 128), block_size = (16, 16), block_stride = (8, 8), cell_size = (8, 8), nbins = 9, win_sigma = DEFAULT_WIN_SIGMA, threshold_L2hys = 0.2, gamma_correction = true, nlevels = DEFAULT_NLEVELS)`

Parameters:

- **win_size** – *Size*
  Size of detection window in pixels (*width, height*). Defines the region of interest. Must be an integer multiple of cell size.

- **block_size** – *Size*
  Block size in pixels (*width, height*). Defines how many cells are in each block. Must be an integer multiple of cell size and it must be smaller than the detection window. The smaller the block the finer detail you will get.

- **block_stride** – *Size*
  Block stride in pixels (*horizontal, vertical*). It must be an integer multiple of cell size. The `block_stride` defines the distance between adjecent blocks, for example, 8 pixels horizontally and 8 pixels vertically. Longer `block_strides` makes the algorithm run faster (because less blocks are evaluated) but the algorithm may not perform as well.

- **cell_size** – *Size*
  Cell size in pixels (*width, height*). Determines the size fo your cell. The smaller the cell the finer detail you will get.

- **nbins** – *int*
  Number of bins for the histograms. Determines the number of angular bins used to make the histograms. With more bins you capture more gradient directions. HOG uses unsigned gradients, so the angular bins will have values between 0 and 180 degrees.

- **win_sigma** – *double*
  Gaussian smoothing window parameter. The performance of the HOG algorithm can be improved by smoothing the pixels near the edges of the blocks by applying a Gaussian spatial window to each pixel before computing the histograms.

- **threshold_L2hys** – *double*
  L2-Hys (Lowe-style clipped L2 norm) normalization method shrinkage. The L2-Hys method is used to normalize the blocks and it consists of an L2-norm followed by clipping and a renormalization. The clipping limits the maximum value of the descriptor vector for each block to have the value of the given threshold (0.2 by default). After the clipping the descriptor vector is renormalized as described in *IJCV*, 60(2):91-110, 2004.

- **gamma_correction** – *bool*
  Flag to specify whether the gamma correction preprocessing is required or not. Performing gamma correction slightly increases the performance of the HOG algorithm.

- **nlevels** – *int*
  Maximum number of detection window increases.

As we can see, the `cv2.HOGDescriptor()` function supports a wide range of parameters. The first few arguments (`block_size, block_stride, cell_size`, and `nbins`) are probably the ones you are most likely to change. The other parameters can be safely left at their default values and you will get good results.

In the code below, we will use the `cv2.HOGDescriptor()` function to set the cell size, block size, block stride, and the number of bins for the histograms of the HOG descriptor. We will then use `.compute(image)` method to compute the HOG descriptor (feature vector) for the given `image`.

```
In [ ]: # Specify the parameters for our HOG descriptor

        # Cell Size in pixels (width, height). Must be smaller than the size of the detection wi
        # and must be chosen so that the resulting Block Size is smaller than the detection wind
        cell_size = (6, 6)

        # Number of cells per block in each direction (x, y). Must be chosen so that the resulti
        # Block Size is smaller than the detection window
        num_cells_per_block = (2, 2)

        # Block Size in pixels (width, height). Must be an integer multiple of Cell Size.
        # The Block Size must be smaller than the detection window
        block_size = (num_cells_per_block[0] * cell_size[0],
                      num_cells_per_block[1] * cell_size[1])

        # Calculate the number of cells that fit in our image in the x and y directions
        x_cells = gray_image.shape[1] // cell_size[0]
        y_cells = gray_image.shape[0] // cell_size[1]

        # Horizontal distance between blocks in units of Cell Size. Must be an integer and it mu
        # be set such that (x_cells - num_cells_per_block[0]) / h_stride = integer.
        h_stride = 1
```