Stefan Breuers <breuers@vision.rwth-aachen.de>
Wolfgang Mehner <mehner@vision.rwth-aachen.de>

Exercise 4: Histograms, Recognition, Interest Points

due **before** 2016-12-19

Important information regarding the exercises:

- The exercise is not mandatory.
- There will be no corrections.
- Nevertheless, we encourage you to work on the exercises and present your solutions in the exercise class. For this regard the submission rules.
- In the archive for this exercise you will find the functions `apply.m` that should be used for displaying your results. You should also use it to test your implementation and see if the results make sense. Answers are to be submitted within `answers.m` Do **not** modify the apply files in any way.
- Please do **not** include the data files in your submission!
- If applicable submit your code solution as a zip/tar.gz file named `mn1_mn2_mn3.{zip/tar.gz}` with your **matriculation numbers** (`mn`).
- Submit your solutions via the L^{2}P system.

Question 1: Histogram Representations ..... $(\Sigma = 0)$

(a) The **`hist`** function of Matlab has two modes: When called without specifying a return value, it displays a histogram of the vector given as an argument. When called with a specified return value, it returns the histogram instead of displaying it.

```
hist(image);                    % Displays the histogram of image
image_histogram = hist(image); % Fills image_histogram with the
    histogram of image
```

Read an image and convert it to gray values. Reshape the 2D image array of size (N,M) to a 1D vector of size (N*M,1). Calculate the histogram of gray values using the **`hist`** function and display it using the **`bar`** function. Compare the histograms for different numbers of bins. Explain your observations.
Implement a function called `myhist` which takes a filename and a number of bins as an input and returns a 1D histogram vector of gray values. Use the **`hist`** function of Matlab for the histogram computation. Normalize the histogram such that its integral (sum) is equal to one.

```
function histogram = myhist(filename, bins)
```

Repeat the histogram computations using this new function. Does the result look the same?

(b) Implement a new function `myhist2` which takes the same input as the previous one but returns a 3D histogram of RGB values.

```
function histogram = myhist2(filename, bins)
```

Optionally, remove the bins corresponding to the black color (background) before normalization.

Visualize the RGB histograms of the images `sunset.png` and `terrain.png` using the provided function `plot_color_histogram` and describe what you see. Which number of bins gives the best impression of the color distribution?

(c) In order to compare two histograms, you need a suitable distance measure. Here, the squared Euclidean distance is a common method.

Implement a function `hist_dist_euclidean` which takes as an input two histograms `h1` and `h2` and returns the squared Euclidean distance between them:

$$\text{dist}(h_1, h_2) = \sqrt{\sum_{i=1}^{D} (h_1(i) - h_2(i))^2}$$

```
function dist = hist_dist_euclidean(h1, h2)
```

Compute the distances between images `model/obj1__0.png` and `model/obj91__0.png`, `model/obj1__0.png` and `model/obj94__0.png`, which distance is smaller and why?

(d) Implement a new function `myhist3` which takes the same input as the previous one but returns a 2D histogram of `r`, `g` values, where $\mathbf{r} = \frac{R}{R+G+B}$ and $\mathbf{g} = \frac{G}{R+G+B}$.

```
function histogram = myhist3(filename, bins)
```

Apply this function to images of a red, a green and a blue object and visualize your results using **`imagesc`**:

```
h_vec = myhist3('model/obj4__0.png', 20);
histogram = reshape(h_vec, 20, 20);
imagesc(histogram);
colormap gray;
```

Do the histograms look correct?

(e) Implement a new function `myhist4` which takes the same input as the previous one but returns a 2D histogram of `dx`, `dy` values, which are Gaussian derivatives of the image in x and y directions. Note that derivatives can have negative values. To correct that, add a constant value to the derivatives (e.g.,if your image contains values between 0 and 255, then the values of the derivatives will be in the range $[-255, 255]$.

```
function histogram = myhist4(filename, bins)
```

Visualize the gradient histograms of some images using **`imagesc`**. For getting a better visual impression, you should logarithmize the histograms:

```
h_vec = myhist4('model/obj4__0.png', 20);
histogram = reshape(h_vec, 20, 20);
imagesc(log(histogram));
colormap gray;
```

Do the histograms look correct?

Question 2: Sliding Window Detection ..... $(\Sigma = 0)$

In this exercise we will implement a simple HOG-like car detector with the help of MATLAB's built-in SVM function.

At first, get the training and test images from the Illinois dataset[1] and save both folders (`TrainImages/` and `TestImages/`) in the parent folder (where `apply.m` is located). This way we can load all images in a folder structure

```
positive_images = dir('TrainImages/pos*.pgm');
negative_images = dir('TrainImages/neg*.pgm');
```

and access single images using

```
img = positive_images(1);
img_data = imread(['TrainImages/', img.name]);
```

To change parameters and choose test images for the following code, change the provided `parameters.m`, not the `apply.m` script.

(a) First we want to implement a simple HOG-like descriptor `calculate_hoglike.m` which takes an image and computes the corresponding HOG-like representation. The general declaration looks like this:

```
function hog = calculate_hoglike(img, cellsize, n_bins, sigma,
    offset_lr, offset_tb, interpolation)
```

where `img` is the grayscale image, `cellsize` is the size of each HOG-like bin in both dimensions, `n_bins` is the number of bins for the angles, `sigma` the scale of the Gaussian used to compute the image gradients and `offset_lr`, `offset_tb` margins in pixels to counteract border artifacts. The boolean flag `interpolation` is used in Subquestion d. The output should be a three dimensional matrix. The first two dimensions are the y and x indices of the HOG cell. The third dimension describes the bins of the HOG descriptor. When the dimensions of the images are not a multiple of the `cellsize`, discard the remaining pixels to the right and to the bottom of the image.

First use the provided `imgradient.m` to get the gradient direction and magnitude of the image. Then cutoff the border according to the offset parameters to compensate for border artifacts.

Since the HOG descriptor is a cell based descriptor, we have to select for each pixel the cell (or with interpolation multiple) cells where the gradient is added. A simple version of this function is provided

```
function [cells_x, cells_y, weights] = get_cell(image_height,
    image_width, cellsize, x, y)
```

where `image_height`, `image_width` are the image dimensions, `cellsize` is the size of the HOG cell and `x` and `y` are the coordinates of the current pixel. It returns row vectors of the x and y coordinates and the weights for assigning the gradient to the corresponding cells. When the correct HOG cell and angle cell are selected, the product of weight and gradient magnitude is added to this cell. Each cell has to be independently L_1 normalized to 1. Note that the original HOG descriptor uses a more elaborated two-stage normalization scheme.

[1] http://cogcomp.cs.illinois.edu/Data/Car/

The results are visualized by `render_hogimage.m`. Describe what you see. Can you spot any striking HOG-cells in the positive and negative example?

(b) We now want to train a SVM classifier in the HOG-like feature space and actually apply it on a new test image. Note that the `apply.m` script already provides the part for training the SVM:

```
samples = [positive_samples; negative_samples];
targets = [ repmat(p.car,size(positive_samples,1),1) ; repmat(p.
    no_car, size(negative_samples,1),1)];
SVMStruct = fitcsvm(samples,targets);
```

Your task is to implement the function

```
function scores = object_hypothesis(svm, hog_img, ny_bins, nx_bins
    )
```

which takes as input the trained SVM model `svm`, the HOG-like representation of the test image `hog_img` and the corresponding number of bins. These should be used to slide a window over the corresponding HOG-image and compute the SVM-score for each window to return the full score matrix `scores`. For this purpose you have to utilize the in-built function

```
[label,score] = predict(SVMModel,newX);
```

in an appropriate manner. Check the documentation for a helpful example.

Once the score matrix is computed you still need to detect the objects. To do so, implement the function

```
function [y,x, score] = detect_objects(hypotheses, threshold)
```

which takes the score matrix `hypotheses` and a threshold to return position and scores of detected object. The `apply.m` script takes care of drawing bounding boxes around the responses. How does the result look?

(c) A detector like our HOG-like descriptor usually gives multiple responses for the same target. To compensate for that non-maximum-suppression needs to be applied. Here, we want to implement a simple 8-neigborhood search.

```
function result = nms(score)
```

This function takes the current score matrix and returns a result matrix of the same size with the non-maximum-suppression applied. It looks at every pixel of the score matrix and keeps it, if it is the maximum in its 8-neighborhood, or is set to 0 otherwise. Set `use_nms` in `parameters.m` to true enable the non-maximum-suppression. Does the result look better? Can you find examples where it fails? Can you imagine a downside of non-maximum-suppression?

(d) Extend the simple assignment scheme of pixels to HOG cells with a bilinear interpolation. The bilinear interpolation will distribute the gradient magnitude over multiple neighboring cells. This will result in a more robust descriptor.

Figure 1 explains the weighting scheme which will be applied. The areas of the colored rectangles are calculated. The weight for assigning the gradient to a given cell is the

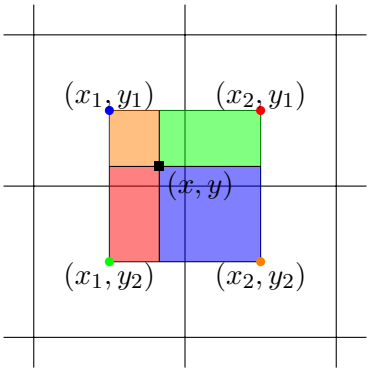

Figure 1: Bilinear Interpolation of HOG features.

area of the opposing surface divided by the sum of all areas. For example the weight for assigning the pixel gradient to the red cell is

$$w_{\text{red}} = \frac{A_{\text{red}}}{A_{\text{red}} + A_{\text{blue}} + A_{\text{green}} + A_{\text{orange}}}.$$

The formula for bilinear interpolation states

$$\begin{aligned}
h(x_1, y_1) &= w \cdot \left(1 - \frac{x - x_1}{b_x}\right)\left(1 - \frac{y - y_1}{b_y}\right) \\
h(x_1, y_2) &= w \cdot \left(1 - \frac{x - x_1}{b_x}\right)\left(\frac{y - y_1}{b_y}\right) \\
h(x_2, y_1) &= w \cdot \left(\frac{x - x_1}{b_x}\right)\left(1 - \frac{y - y_1}{b_y}\right) \\
h(x_2, y_2) &= w \cdot \left(\frac{x - x_1}{b_x}\right)\left(\frac{y - y_1}{b_y}\right)
\end{aligned}$$

Remember that $b_x = x_2 - x_1$ and $b_y = y_2 - y_1$. Implement this method by implementing the function

```
function [cells_x, cells_y, weights] = get_cell_weights(
    image_height, image_width, cellsize, x, y)
```

with the same signature as described in Exercise a.

Modify the function `calculate_hoglike` to make use of this improvement when the flag `interpolation` is set. Set `use_interpolation` in `parameters.m` to true to use interpolation. You might need to clear the workspace to retrain your classifier. Try out your interpolated HOG-like detector with and without non-maximum-supression. What do you observe?

Question 3: Hessian Detector ..... $(\boldsymbol{\Sigma} = \mathbf{0})$

In this exercise, we will implement a simple Hessian detector. This detector operates on the second-derivative matrix $\mathbf{H}$ (called the "Hessian" matrix)

$$\mathbf{H} = \begin{bmatrix} D_{xx}(x, y; \sigma) & D_{xy}(x, y; \sigma) \\ D_{xy}(x, y; \sigma) & D_{yy}(x, y; \sigma) \end{bmatrix}. \tag{1}$$

It defines keypoints as those points for which the Hessian determinant is greater than a certain threshold t. In order to obtain responses that are invariant to scaling, we include an additional scale normalization factor σ^4. (The reason for this particular factor can be derived from scale space theory. Briefly stated, different scales of the image are handled by smoothing it with a Gaussian kernel. Each Gaussian derivative operation needs to be compensated by a normalization with the same scale factor σ in order to have responses that can be compared to each other (or to the same threshold). Since we have a product of two second derivatives here, this leads to the normalization factor σ^4).

$$\sigma^4 \det(\mathbf{H}) = \sigma^4 \left(D_{xx} D_{yy} - D_{xy}^2\right) \overset{!}{>} t \tag{2}$$

(a) Write a function `hessian` which computes the Hessian determinant for each pixel of a given image, performs non-maximum suppression on the determinant image and returns the coordinates of all points that pass the threshold. (Note: for obtaining the coordinates, you can use the following command: `[py, px] = find(imgPts > thresh)`).

```
function [px, py] = hessian(img, sigma, thresh)
```

(b) Use the function `drawpoints` to display the detected points overlaid on the original image. Load the images `graf.png` and `gantrycrane.png` and compute Hessian interest points for them. Experiment with different parameter settings. What do you observe?

Question 4: Harris Detector ..... $(\mathbf{\Sigma = 0})$

In this exercise, we will implement a Harris detector. This detector searches for corner-like structures by looking for points $p = (x, y)$ where the autocorrelation matrix $\mathbf{C}$ around p has two large eigenvalues. The matrix $\mathbf{C}$ can be computed from the first derivatives in a window around p, weighted by a Gaussian $G(x, y; \tilde{\sigma})$:

$$\mathbf{C}(x, y; \sigma, \tilde{\sigma}) = G(x, y; \tilde{\sigma}) \star \sigma^2 \begin{bmatrix} D_x^2(x, y; \sigma) & D_x D_y(x, y; \sigma) \\ D_x D_y(x, y; \sigma) & D_y^2(x, y; \sigma) \end{bmatrix}, \tag{3}$$

where "$\star$" denotes the convolution operator (*hint*: this notation means the convolution is applied to each of the result images $D_x^2, D_x D_y, D_y^2$). Instead of explicitly computing the eigenvalues λ_1 and λ_2 of $\mathbf{C}$, the following equivalences are used

$$\det(\mathbf{C}) = \lambda_1 \lambda_2 \tag{4}$$

$$\text{trace}(\mathbf{C}) = \lambda_1 + \lambda_2 \tag{5}$$

to check if their ratio $r = \frac{\lambda_1}{\lambda_2}$ is below a certain threshold. With

$$\frac{\text{trace}^2(\mathbf{C})}{\det(\mathbf{C})} = \frac{(\lambda_1 + \lambda_2)^2}{\lambda_1 \lambda_2} = \frac{(r\lambda_2 + \lambda_2)^2}{r\lambda_2^2} = \frac{(r+1)^2}{r} \tag{6}$$

we can express this by the following condition

$$\det(\mathbf{C}) - \alpha \cdot \text{trace}^2(\mathbf{C}) > t. \tag{7}$$

In practice, the parameters are usually set to the following values: $\tilde{\sigma} = 1.6 \cdot \sigma$, $\alpha = 0.06$.

(a) Write a function `harris` which computes the matrix $\mathbf{C}$ for each pixel of a given image, calculates its trace and determinant, and combines them according to equation (7). After applying non-maximum suppression on the result image, it should compare the remaining points to the given threshold and return the coordinates of all points that pass the threshold.

```
function [px, py] = harris(img, sigma, thresh, norm_sigma)
```

(b) Load the images `graf.png` and `gantrycrane.png` and compute Harris interest points for them. Compare the results to those of the Hessian detector from Question 3. Experiment with different parameter settings. You can start with the settings `sigma=1.6, thresh=1000`. What do you observe?

(c) The factors σ^2 in (3) and σ in $\tilde{\sigma} = 1.6 \cdot \sigma$ are also derived from scale space theory. Scale the image, i.e. convolve it with a Gaussian filter with $\sigma = 2$ and 4 and compare the results for the different scales. What happens if you do not use the scale space normalization factors and why? (*Hint:* You may add parameters to the specified interface, if you provide a default value, see **`nargin`**.)