01_mean_shift

May 12, 2019

1 Mean-Shift Clustering and Segmentation

In the first part of this task you will implement the *mean-shift* clustering algorithm in a general way (not specifically for anything to do with images, just simply for n-dimensional data points).

Then you will apply mean-shift for image segmentation, by clustering data points that represent pixels (e.g. the colors).

```
In [ ]: %%html
        <!-- Add heading numbers -->
        <style>
        body {counter-reset: section;}
        h2:before {counter-increment: section;
                   content: counter(section) " ";}
        </style>
```

```
In [ ]: # Some imports
        %matplotlib notebook
        import numpy as np
        import matplotlib.pyplot as plt
        import time
        import imageio
        from mpl_toolkits.mplot3d import Axes3D
        import cv2
        import os
```

1.1 Recap from the lecture

The *mean-shift* algorithm clusters an n-dimensional data set (i.e., each data point is described by a feature vector of n values) by associating each point with a peak in the estimated probability density of the dataset's distribution. Points associated with the "same" peak (up to some small threshold) become members of the same cluster.

For each point, mean-shift finds the associated peak by first defining a spherical window of radius r centered on that point, and computing the **mean** of the points that lie within the window. The algorithm then **shifts** the window to be centered on that mean and repeats these steps until convergence, i.e., until the shift is smaller than a specified threshold (set this as 0.1). At each iteration the window shifts to a more densely populated portion of the data set until a peak is reached, where the data is approximately equally distributed in the window.

1.2 Finding a peak from a query point

Implement the peak searching process as the function `find_peak(data, query, radius)` where

- `data` is a $p \times n$ matrix containing p data points; each point is defined by an n-dimensional row vector of feature values
- `query` is the n-dimensional starting point from which we wish to compute a density peak
- `radius` is the search window radius.

Return the peak as an n-dimensional vector.

Hints: You can use `np.linalg.norm` to compute the Euclidean norm of a vector. You can also index NumPy arrays with Boolean arrays, e.g. to select only rows that fulfil some criterion.

```
In [ ]: def find_peak(data, query, radius):
            # YOUR CODE HERE
            raise NotImplementedError()
```

You can use the synthetic dataset `gaussian_mixture_samples_3d.csv` to test your implementation. The data points in this file are 2000 samples from two 3D Gaussian distributions. The following plots only show the projection on the XY plane.

```
In [ ]: data = np.genfromtxt(f'gaussian_mixture_samples_3d.csv', delimiter=',')
        point_ids = [0, 5, 1500]
        radius = 2

        fig, axes = plt.subplots(1, len(point_ids), figsize=(12,4))
        for query, ax in zip(data, axes):
            peak = find_peak(data, query, radius)
            print('Found peak', peak)
            ax.scatter(data[:, 0], data[:, 1], marker='.', color='gray')
            ax.scatter(query[0], query[1], s=150, linewidth=5,
                       color='blue', marker='x', label='starting point')
            ax.scatter(peak[0], peak[1], color='orange', marker='x',
                       s=150, linewidth=5, label='found peak')
            ax.legend()
        fig.tight_layout()
```

1.3 Clustering all points

Implement `mean_shift(data, radius)`, which calls `find_peak` for each point and then assigns a label to each point according to its peak. `mean_shift` should return two arrays: `peaks` and `labels`.

- `peaks` is a matrix with k rows, storing the found density peaks, where k is the data-dependent number of clusters found.
- `labels` is a p-sized vector that has an entry for each data point, storing its associated cluster label (an integer)

Similar peaks within a distance of `radius`/2 should be considered the same and should be merged after each call to `find_peak`. More specifically, if the peak computed for a data point already exists in `peaks` (according to the distance threshold), then discard the newly computed peak and give the label of the already existing peak to the considered data point.

```
In [ ]: def mean_shift(data, radius):
            labels = np.full(len(data), fill_value=-1, dtype=int)

            # YOUR CODE HERE
            raise NotImplementedError()
            return peaks, labels
```

Now check the result of your implementation. The found peaks (cluster centers) are shown as black X marks. You can rotate the interactive 3D plots with the mouse (but it may be somewhat slow).

```
In [ ]: def plot_3d_clusters(ax, data, labels, peaks,
                              peak_colors=None, colors=None, axis_names='xyz'):
            """Plots a set of point clusters in 3D, each with different color."""

            def luv2rgb(color):
                expanded = color[np.newaxis, np.newaxis]
                rgb = cv2.cvtColor(expanded.astype(np.uint8), cv2.COLOR_LUV2RGB)
                return rgb[0,0]/255

            if peak_colors is None:
                peak_colors = peaks

            for label in range(len(peaks)):
                if colors=='rgb':
                    cluster_color = color = peak_colors[label]/255
                elif colors=='luv':
                    cluster_color = luv2rgb(peak_colors[label])
                else:
                    cluster_color=None

                cluster = data[labels==label]
                ax.scatter(cluster[:, 0], cluster[:, 1], cluster[:, 2],
                           alpha=0.15, color=cluster_color)
                ax.scatter(peaks[label, 0], peaks[label, 1], peaks[label, 2],
                           color='black', marker='x', s=150, linewidth=3)

            ax.set_xlabel(axis_names[0])
            ax.set_ylabel(axis_names[1])
            ax.set_zlabel(axis_names[2])

        data = np.genfromtxt(f'gaussian_mixture_samples_3d.csv', delimiter=',')
        radii = [1, 1.25, 2, 8]
        fig, axes = plt.subplots(
            1, len(radii), figsize=(15,4), subplot_kw={'projection': '3d'})

        for radius, ax in zip(radii, axes):
            start_time = time.time()
```

```
            peaks, labels = mean_shift(data, radius)
            plot_3d_clusters(ax, data, labels, peaks)
            duration = time.time()-start_time
            ax.set_title(
                f'Found {len(peaks)} peaks using radius={radius:.2f}\n'
                f'Computation took {duration:.4f} s\n')

        fig.tight_layout()
```

1.4 Speedups

As described so far, the mean-shift algorithm is too slow to be used for image segmentation where each data point corresponds to a pixel. Therefore, you should incorporate the following two speedups from the lecture into your implementation.

First speedup: upon finding a new peak, associate each data point within `radius` distance from that peak with the cluster defined by that peak. This speedup is known as the *"basin of attraction"* and is based on the intuition that points within one window size distance from the peak will, with high probability, converge to that peak.

Call the new function `mean_shift_opt`.

```
In [ ]: def mean_shift_opt(data, radius):
            labels = np.full(len(data), fill_value=-1, dtype=int)

            # YOUR CODE HERE
            raise NotImplementedError()

            return peaks, labels
```

Now run the code to check the result.

```
In [ ]: data = np.genfromtxt(f'gaussian_mixture_samples_3d.csv', delimiter=',')
        radii = [1, 1.25, 2, 8]
        fig, axes = plt.subplots(
            1, len(radii), figsize=(15,4), subplot_kw={'projection': '3d'})

        for radius, ax in zip(radii, axes):
            start_time = time.time()
            peaks, labels = mean_shift_opt(data, radius)
            plot_3d_clusters(ax, data, labels, peaks)
            duration = time.time()-start_time
            ax.set_title(
                f'Found {len(peaks)} peaks using radius={radius:.2f}\n'
                f'Computation took {duration:.4f} s\n')

        fig.tight_layout()
```

The **second speedup** is based on a similar principle: Associate points within a distance `radius`$/c$ of the search path with the converged peak (c is some constant value). Use $c = 3$ for this assignment.

To realize this speedup, you will need to modify `find_peak` into `find_peak_opt`, which returns two values: `peak` and `is_near_search_path`. The latter should be a Boolean output vector of length p (number of data points) containing `True` for each point that we encountered within a distance `radius`$/c$ on our search path, and `False` for the others. Then use this boolean vector in a new version of `mean_shift_opt`, called `mean_shift_opt2` to do the label assignments accordingly.

```
In [ ]: def find_peak_opt(data, query, radius, c=3):
            is_near_search_path = np.zeros(len(data), dtype=bool)

            # YOUR CODE HERE
            raise NotImplementedError()

        def mean_shift_opt2(data, radius):
            labels = np.full(len(data), fill_value=-1, dtype=int)

            # YOUR CODE HERE
            raise NotImplementedError()

            return peaks, labels

        data = np.genfromtxt(f'gaussian_mixture_samples_3d.csv', delimiter=',')
        radii = [1, 1.25, 2, 8]
        fig, axes = plt.subplots(
            1, len(radii), figsize=(15,4), subplot_kw={'projection': '3d'})

        for radius, ax in zip(radii, axes):
            start_time = time.time()
            peaks, labels = mean_shift_opt2(data, radius)
            plot_3d_clusters(ax, data, labels, peaks)
            duration = time.time()-start_time
            ax.set_title(f'Found {len(peaks)} peaks using radius={radius:.2f}\n'
                         f'Computation took {duration:.4f} s\n')

        fig.tight_layout()
```

1.5 Questions

1. Which radius gives good results and how can one find it?
2. How much faster is the optimized version? Does it change the result? If yes, is the result worse?

YOUR ANSWER HERE

1.6 Image Segmentation by Mean-Shift

Now use your mean-shift implementation from above to segment images. Note that although very efficient mean-shift implementations exist, our version here may take several minutes per image. Debug using small images.

Implement the funtion `mean_shift_segment(im, radius)` where `im` is an input RGB image of shape `height` $\times$ `width` $\times$ 3 and `radius` is the parameter associated with the mean-shift algorithm. The output should be `data`, `peaks`, `labels`, `segmented_image`:

- `data` is an array of the data points that you input to the mean-shift algorithm, with p rows and 3 columns.
- `peaks` and `labels` are simply the results returned by the `mean_shift` call (without changing them).
- The `segmented_image` is constructed by assigning to each pixel the color value of the corresponding cluster's peak.

You will need to reshape (`np.reshape`) the image array before feeding it to your mean-shift clustering function. Also, do not forget to convert the pixel values to floating point, using `somearray.astype(float)` and then convert the result back to 8-bit unsigned integers using `somearray.astype(np.uint8)`.

Segment the image `terrain_small.png` using `radius` 15.

```
In [ ]: def mean_shift_segment(im, radius):
            # YOUR CODE HERE
            raise NotImplementedError()
            return data, peaks, labels, segmented_im

In [ ]: def make_label_colormap():
            """Create a color map for visualizing the labels themselves,
            such that the segment boundaries become more visible, unlike
            in the visualization using the cluster peak colors.
            """
            import matplotlib.colors
            rng = np.random.RandomState(2)
            values = np.linspace(0, 1, 20)
            colors = plt.cm.get_cmap('hsv')(values)
            rng.shuffle(colors)
            return matplotlib.colors.ListedColormap(colors)

        label_cmap = make_label_colormap()

        im = imageio.imread('terrain_small.png')
        start_time = time.time()
        data, peaks, labels, segmented_im = mean_shift_segment(im, radius=15)
        duration= time.time()-start_time
        print(f'Took {duration:.2f} s')

        fig = plt.figure(figsize=(10,8))
        ax = fig.add_subplot(2, 2, 1)
        ax.set_title('Original Image')
        ax.imshow(im)

        ax = fig.add_subplot(2, 2, 2)
        ax.set_title('Segmented')
```

```
        ax.imshow(segmented_im)

        ax = fig.add_subplot(2, 2, 3)
        ax.set_title('Labels')
        ax.imshow(labels.reshape(im.shape[:2]), cmap=label_cmap)

        ax = fig.add_subplot(2, 2, 4, projection='3d')
        ax.set_title(f'RGB space')
        plot_3d_clusters(ax, data, labels, peaks, colors='rgb', axis_names='RGB')
        fig.tight_layout()
```

1.7 Segmentation in LUV color space

Note that Mean-Shift uses the Euclidean distance metric. Unfortunately, the Euclidean distance in RGB color space does not correlate well to perceived difference in color by people. For example, in the green portion of RGB space, large distances are perceived as the same color, whereas in the blue part a small RGB-distance may represent a large change in perceived color. For this reason you should use the non-linear LUV color space. In this space Euclidean distance better models the perceived difference in color.

In the function `mean_shift_segment_luv(...)` cluster the image data in LUV color space by first converting the RGB color vectors to LUV using OpenCV: `cv2.cvtColor(rgb_image, cv2.COLOR_RGB2LUV)`. Then convert the resulting cluster centers back to RGB using `cv2.cvtColor(luv_image, cv2.COLOR_LUV2RGB)`.

If we want to include spatial *position information* in the feature vectors as well, we can make the feature vectors 5 dimensional by specifying the LUV values as well as the x,y coordinates of each pixel. Write a function `mean_shift_segment_luv_pos(im, radius)` that does this. **Hint:** useful functions are `np.arange`, `np.meshgrid`, `np.concatenate`, `np.expand_dims`).

```
In [ ]: def mean_shift_segment_luv(im, radius):
            # YOUR CODE HERE
            raise NotImplementedError()
            return data, peaks, labels, segmented_im

        def mean_shift_segment_luv_pos(im, radius, pos_factor=1):
            # YOUR CODE HERE
            raise NotImplementedError()
            return data, peaks, labels, segmented_im
```

```
In [ ]: im = imageio.imread('terrain_small.png')
        data, peaks, labels, segmented_im = mean_shift_segment_luv(im, radius=10)
        fig = plt.figure(figsize=(12,8))

        ax = fig.add_subplot(2, 3, 1)
        ax.set_title('Segmented (LUV)')
        ax.imshow(segmented_im)

        ax = fig.add_subplot(2, 3, 2)
        ax.set_title('Labels (LUV)')
```

```
ax.imshow(labels.reshape(im.shape[:2]), cmap=label_cmap)

ax = fig.add_subplot(2, 3, 3, projection='3d')
ax.set_title(f'LUV space')
plot_3d_clusters(ax, data, labels, peaks, colors='luv', axis_names='LUV')

data, peaks, labels, segmented_im = mean_shift_segment_luv_pos(im, radius=20)
ax = fig.add_subplot(2, 3, 4)
ax.set_title('Segmented (LUV+pos)')
ax.imshow(segmented_im)

ax = fig.add_subplot(2, 3, 5)
ax.set_title('Labels (LUV+pos)')
ax.imshow(labels.reshape(im.shape[:2]), cmap=label_cmap)

ax = fig.add_subplot(2, 3, 6, projection='3d')
ax.set_title(f'VXY space')
plot_3d_clusters(
    ax, data[:, 2:], labels, peaks[:, 2:],
    peak_colors=peaks[:, :3], colors='luv', axis_names='VXY')
ax.invert_zaxis()
ax.view_init(azim=20, elev=15)

fig.tight_layout()
```

1.8 Experiment with the parameters

How does the `radius` and the *feature vector* affect the resulting segmentations? What effect does adding position information have? What are the advantages and disadvantages of using each type of feature vector? Can you suggest any extensions that might avoid many of the situations where single regions are over-segmented into multiple regions?

YOUR ANSWER HERE

```
In [ ]: radii = [5, 10, 20]
        fig, axes = plt.subplots(len(radii), 3, figsize=(15, 15))
        for radius, ax in zip(radii, axes):
            segmented_im = mean_shift_segment(im, radius)[-1]
            ax[0].imshow(segmented_im)
            ax[0].set_title(f'Radius {radius} RGB')

            segmented_im = mean_shift_segment_luv(im, radius)[-1]
            ax[1].imshow(segmented_im)
            ax[1].set_title(f'Radius {radius} LUV')

            segmented_im = mean_shift_segment_luv_pos(im, radius)[-1]
            ax[2].imshow(segmented_im)
            ax[2].set_title(f'Radius {radius} LUV+pos')
        fig.tight_layout()
```

1.9 [BONUS] Test it on a larger image

Run your algorithm on at least one larger (approx. 300x200) test image using your own choice of `radius` and feature vector definition. One source for possible test images is the dataset of images available at http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/. You can also try the example images included in the scikit-image library, e.g. `import skimage.data; im = skimage.data.astronaut()`. Or any image from anywhere.

Processing can take several minutes for larger images.

```
In [ ]: import skimage.data
        im = skimage.data.astronaut()
        im = cv2.resize(im, (256,256))

        # YOUR CODE HERE
        raise NotImplementedError()

        fig, axes = plt.subplots(1, 3, figsize=(15, 5))
        axes[0].set_title('Original image')
        axes[0].imshow(im)
        axes[1].set_title('Segmented image')
        axes[1].imshow(segmented_im)
        axes[2].set_title('Labels')
        axes[2].imshow(labels.reshape(im.shape[:2]), cmap=label_cmap)
        fig.tight_layout()
```

1.10 [BONUS++] Efficient Implementation

Mean-shift is highly parallelizable and GPU-based implementations can be many times faster for such workloads. If you already know TensorFlow or CUDA, you can try implementing mean-shift for the GPU.

```
In [ ]: # YOUR CODE HERE
        raise NotImplementedError()
```

02_graph_cuts

May 12, 2019

1 Graph Cut Segmentation

In this part you will implement foreground-background segmentation using *Markov random fields* (MRF).

1.1 Recap from the lecture

A Markov random field is a graphical model that expresses the structure of (input and output) variables. In our case that means we do not only use a color model for foreground and background but also take into account the neighborhood relations of the pixels. This encodes the intuition that neighboring pixels usually belong to the same region.

The color (or more generally, appearance) models and the neighborhood relations are combined in a so-called *energy function* (or cost function), which is then minimized to obtain an optimal label-assignment.

Given a structured input (here: image pixel colors) $\mathcal{Y} = \{Y_j | j \in I\}$ we want to find the output (here: labeling) $\mathcal{X} = \{X_j | j \in I\}$ such that

$$\hat{\mathcal{X}} = \arg\min_{\mathcal{X}} E(\mathcal{X}, \mathcal{Y})$$

$$E(\mathcal{X}, \mathcal{Y}) = \sum_{j \in I} \psi_u(X_j, Y_j) + \sum_{i,j \in I} \psi_p(X_j, X_j, Y_i, Y_j, i, j).$$

The set I contains all possible pixel indices. In our two-label (binary) segmentation case, the label variables must be either 0 (background) or 1 (foreground) $X_j \in \{0, 1\}$.

The so-called *unary potential* $\psi_u(X_j, Y_j)$ is the cost of assigning the label X_j to a pixel with color Y_j. In probabilistic terms, the unary potential is

$$\psi_u(X_j, Y_j) = -\omega_u \cdot \log p(X_j | Y_j),$$

with an appropriate model p for the foreground and the background and a weighting factor ω_u. The unaries encourage labeling each pixel with the label (foreground/background) whose color model is a better fit for that particular pixel.

The *pairwise potential* ψ_p incorporates the dependencies between pixels. To speed up the computation, the pairwise model is usually restricted to neighboring pixels and is therefore set to zero if the ith and jth pixels are not direct neighbors in a 4-neighborhood. In our case it written as:

$$\psi_p(X_i, X_j, Y_i, Y_j, i, j) = \omega_p \cdot \begin{cases} 1, & \text{if } X_i \neq X_j \text{ and } i, j \text{ are neighbors} \\ 0, & \text{otherwise} \end{cases}$$

with some weighting factor ω_p. Such pairwise potentials encourage neighboring pixels to have the same label.

A Graph Cut method is used to find the optimal solution $\hat{\chi}$ of the energy function.

1.2 Bird's eye overview

It's easy to get lost in all the details, so here's an roadmap of what we're going to do:

1. Set up the Markov Random Field (define unaries and pairwise potentials), in more detail:
2. Model the distribution of background and foreground colors based on the colors found in approximate initial regions
3. For each pixel independently, calculate the posterior probability of being foreground, based on the models from the previous step ("probability map")
4. Calculate the unary potentials based on the foreground probability map
5. Define the pairwise potentials (using the neighborhood relations)
6. Use graph cuts to minimize the energy function of the Markov Random Field and obtain a labeling

You will not have to implement the min-cut algorithm yourself, we will use the `pygco` ("Python Graph Cut Optimizer") package for that. You can install it using `conda install pygco -c kayarre`. (This is a Python wrapper over the C++ gco library, which can be found at https://vision.cs.uwaterloo.ca/code/)

In [ ]:
```
%%html
<!-- Add heading numbers -->
<style>
body {counter-reset: section;}
h2:before {counter-increment: section;
           content: counter(section) " ";}
</style>
```

In [ ]:
```
# Some imports and helper functions
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
import imageio
import time
import cv2
import pygco  # conda install pygco -c kayarre

def draw_mask_on_image(image, mask, color=(0, 255, 255)):
    """Return a visualization of a mask overlaid on an image."""
    result = image.copy()
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
    dilated = cv2.morphologyEx(mask.astype(np.uint8), cv2.MORPH_DILATE, kernel)
    outline = dilated > mask
    result[mask == 1] = (result[mask == 1] * 0.4 +
                         np.array(color) * 0.6).astype(np.uint8)
    result[outline] = color
    return result
```

1.3 Initial masks

First, manually create initial boxes of foreground and background regions.

We will use these to build color models. That is, to model the probability of a pixel color occuring, given either that it is a foreground or a background pixel.

```
In [ ]: im = imageio.imread('lotus320.jpg')
        h,w = im.shape[:2]

        # Set up initial foreground and background
        # regions for building the color model
        init_fg_mask = np.zeros([h, w])
        init_bg_mask = np.zeros([h, w])

        # Do something like the following but with sensible
        # indices
        # init_fg_mask[10:20, 15:30] = 1
        # init_bg_mask[60:90, 50:110] = 1

        # YOUR CODE HERE
        raise NotImplementedError()

        fig, axes = plt.subplots(1, 2, figsize=(10,5))
        axes[0].set_title('Initial foreground mask')
        axes[0].imshow(draw_mask_on_image(im, init_fg_mask))
        axes[1].set_title('Initial background mask')
        axes[1].imshow(draw_mask_on_image(im, init_bg_mask))
        fig.tight_layout()
```

1.4 Color histograms

Usually *Gaussian mixture models* are used to model color distributions. However, to keep this exercise simple, we will only use color histograms (i.e. the relative frequencies of quantized colors) in the respective region of the image defined by the boxes. In other words, we model the color simply as a categorical random variable.

Implement the function `calculate_histogram`. It should take as input the image `img` with values in $[0, 255]$ and a `mask` the same size as the image. The mask is 1 at the positions of the image where the histogram should be computed, zero otherwise. The final parameter `n_bins` defines how many bins should be used in the histogram in each dimension. The function should **return a 3-dimensional array** of shape `[n_bins, n_bins, n_bins]`, containing the relative frequency for each (r,g,b) color bin within the region of the image defined by the mask, i.e. the fraction of pixels falling within each bin. The histogram should be normalized (sum to 1). Initialize all bins with a small value (10^3) to counteract relative frequencies which are zero (this method is called additive smoothing).

```
In [ ]: def calculate_histogram(img, mask, n_bins):
            histogram = np.full((n_bins, n_bins, n_bins), fill_value=0.001)

            # YOUR CODE HERE
```

```
            raise NotImplementedError()
            return histogram
```

```
In [ ]: n_bins = 10
        fg_histogram = calculate_histogram(im, init_fg_mask, n_bins)
        bg_histogram = calculate_histogram(im, init_bg_mask, n_bins)

        fig, axes = plt.subplots(
            3, 2, figsize=(5,5), sharex=True,
            sharey=True, num='Relative frequency of color bins')

        x = np.arange(n_bins)
        axes[0,0].bar(x, np.sum(fg_histogram, (1, 2)))
        axes[0,0].set_title('red (foreground)')
        axes[1,0].bar(x, np.sum(fg_histogram, (0, 2)))
        axes[1,0].set_title('green (foreground)')
        axes[2,0].bar(x, np.sum(fg_histogram, (0, 1)))
        axes[2,0].set_title('blue (foreground)')

        axes[0,1].bar(x, np.sum(bg_histogram, (1, 2)))
        axes[0,1].set_title('red (background)')
        axes[1,1].bar(x, np.sum(bg_histogram, (0, 2)))
        axes[1,1].set_title('green (background)')
        axes[2,1].bar(x, np.sum(bg_histogram, (0, 1)))
        axes[2,1].set_title('blue (background)')
        fig.tight_layout()
```

What do you see in these histograms?
YOUR ANSWER HERE

1.5 Foreground probability map

The next step in the segmentation process is to estimate a probability map: For each pixel we want to estimate the probability that it belongs to the foreground. This will be used as basis for the unary potential.

The function `foreground_pmap(img, fg_histogram, bg_histogram)` should take the image `img` and the two histograms `fg_histogram`, `bg_histogram` estimated from the foreground region and the background region respecively. It should return an array of shape `height` $\times$ `width` containing the probability of each pixel belonging to the foreground. To estimate the required probability $p(c|[r,g,b])$ from the computed histograms, a class prior $p(c)$ of 0.5 should be used, which means that both foreground and background pixels are equally likely a priori.

Recall Bayes' theorem applied to this case:

$$p(c \mid r,g,b) = \frac{p(c) \cdot p(r,g,b \mid c)}{p(r,g,b)} = \frac{p(c) \cdot p(r,g,b \mid c)}{\sum_{\tilde{c}} p(\tilde{c}) \cdot p(r,g,b \mid \tilde{c})}$$

```
In [ ]: def foreground_pmap(img, fg_histogram, bg_histogram):
            # YOUR CODE HERE
            raise NotImplementedError()
```

```
In [ ]: foreground_prob = foreground_pmap(im, fg_histogram, bg_histogram)
        fig, axes = plt.subplots(1, 2, figsize=(10,5), sharey=True)
        axes[0].imshow(im)
        axes[0].set_title('Input image')
        im_plot = axes[1].imshow(foreground_prob, cmap='viridis')
        axes[1].set_title('Foreground posterior probability')
        fig.tight_layout()
        fig.colorbar(im_plot, ax=axes)
        foreground_map = (foreground_prob > 0.5)
```

Explain what you see in the probability map.
YOUR ANSWER HERE

1.6 Unary potentials

Use the previously computed probability map `foreground_map` to compute the unary potential for both foreground and background.

This function `unary_potentials(probability_map, unary_weight)` shall use the `probability_map` and a scalar weighting factor to compute the unary potentials. It should return a matrix of the same size as the probability matrix.

```
In [ ]: def unary_potentials(probability_map, unary_weight):
            # YOUR CODE HERE
            raise NotImplementedError()

        unary_weight = 1
        unary_fg = unary_potentials(foreground_prob, unary_weight)
        unary_bg = unary_potentials(1 - foreground_prob, unary_weight)
        fig, axes = plt.subplots(1, 2, figsize=(10,5), sharey=True)
        axes[0].imshow(unary_fg)
        axes[0].set_title('Unary potentials (foreground)')
        im_plot = axes[1].imshow(unary_bg)
        axes[1].set_title('Unary potentials (background)')
        fig.tight_layout()
        fig.colorbar(im_plot, ax=axes)
```

Why are the unary potentials for the foreground so small in the middle of the flower?
YOUR ANSWER HERE

1.7 Pairwise potentials

Create a function to compute the prefactor w_p of the pairwise potential for two specific pixels. Implement the funtion below, where `img` is the image, `(x1, y1)`, `(x2, y2)` are the pixel coordinates in the image and the last parameter is the weight ω_p for the pairwise potential. (Do not confuse `(x1, y1)`, `(x2, y2)` with the X_j, Y_j from the top of the page)

Keep in mind that this prefactor does not depend on the labels and is therefore independent of $\mathcal{X}$.

Also, the function signature is more general (see the contrast-sensitive Potts Model question later on), not all parameters are needed here.

Hint: the function is extremely simple.

```
In [ ]: def pairwise_potential_prefactor(img, x1, y1, x2, y2, pairwise_weight):
            # YOUR CODE HERE
            raise NotImplementedError()
```

Using the functions from the previous task, implement a function to compute all the pairwise potentials for the image using 4-neighborhoods. That means only the top, bottom, left and right neighboring pixels should be connected to a given pixel.

The function `pairwise_potentials` should return the `edges` (represented as index pairs) and an array `costs` containing the corresponding edge costs (i.e. the value of the pairwise potential prefactor). Note that you have to use a linearized index instead of (x,y)-coordinates. A conversion function is supplied (`coords_to_index(x, y, width)`).

Again, `edges` should be an integer array of shape $k \times 2$, while `costs` should have length k, where k is the number of neighborhood-edges in the image grid.

```
In [ ]: def coords_to_index(x, y, width):
            return y * width + x

        def pairwise_potentials(im, pairwise_weight):
            # YOUR CODE HERE
            raise NotImplementedError()
            return edges, costs

        pairwise_edges, pairwise_costs = pairwise_potentials(im, pairwise_weight=3)
```

Now you can execute the optimization and plot the resulting labeling.

```
In [ ]: def graph_cut(unary_fg, unary_bg, pairwise_edges, pairwise_costs):
            unaries = np.stack([unary_bg.flat, unary_fg.flat], axis=-1)
            labels = pygco.cut_general_graph(
                pairwise_edges, pairwise_costs, unaries,
                1-np.eye(2), n_iter=-1, algorithm='swap')
            return labels.reshape(unary_fg.shape)

        graph_cut_result = graph_cut(unary_fg, unary_bg, pairwise_edges, pairwise_costs)
        fig, axes = plt.subplots(1, 2, figsize=(10,5), sharey=True)
        axes[0].set_title('Simple thresholding of foreground probability')
        axes[0].imshow(draw_mask_on_image(im, foreground_prob>0.5))
        axes[1].set_title('Graph cut result')
        axes[1].imshow(draw_mask_on_image(im, graph_cut_result))
        fig.tight_layout()
```

Why is the segmentation the way it is?
YOUR ANSWER HERE

1.8 [BONUS] Try another image

First, create a single function that runs the whole segmentation pipeline starting from the image.

`segment_image(...)` should return the final binary segmentation mask with 1 at the foreground pixels and 0 at the background.

```
In [ ]: def segment_image(im, init_fg_mask, init_bg_mask,
                         unary_weight, pairwise_weight, n_bins):
            # YOUR CODE HERE
            raise NotImplementedError()
```

```
In [ ]: import skimage.data

        def run_on_another_image():
            im = skimage.data.immunohistochemistry()
            #im = imageio.imread('flowers.jpg')
            #im = skimage.data.stereo_motorcycle()[0]
            h, w = im.shape[:2]
            fg_mask = np.zeros([h, w])
            bg_mask = np.zeros([h, w])

            # Set some appropriate parts of fg_mask and bg_mask to 1 for initialization.
            # YOUR CODE HERE
            raise NotImplementedError()

            graph_cut_result = segment_image(
                im, fg_mask, bg_mask,
                unary_weight=1, pairwise_weight=1, n_bins=8)

            fig, axes = plt.subplots(1, 3, figsize=(14,5), sharey=True)
            axes[0].set_title('Initial foreground mask')
            axes[0].imshow(draw_mask_on_image(im, fg_mask))
            axes[1].set_title('Initial background mask')
            axes[1].imshow(draw_mask_on_image(im, bg_mask))
            axes[2].set_title('Graph cut result')
            axes[2].imshow(draw_mask_on_image(im, graph_cut_result))
            fig.tight_layout()

        run_on_another_image()
```

Does it look good? Which parameter would you need to change to reduce the number of holes in the segmentation? Try it.

Now try segmenting `im = skimage.data.stereo_motorcycle()[0]` using this technique. Can you segment out the motorbike by fiddling with the parameters? Why or why not?

1.9 [BONUS] Contrast-Sensitive Potts Model

Go back to the `pairwise_potential_prefactor` function and modify it to incorporate a new term, resulting in the so-called *contrast sensitive Potts model*. The new pairwise potential should be:

$$\psi_p(X_i, X_j, Y_i, Y_j, i, j) = \omega_p \cdot \exp\left(-\omega_d \|Y_i - Y_j\|^2\right) \cdot \begin{cases} 1, & \text{if } X_i \neq X_j \text{ and } i, j \text{ are neighbors} \\ 0, & \text{otherwise} \end{cases}$$

This means the prefactor is now $\omega_p \cdot \exp\left(-\omega_d \|Y_i - Y_j\|^2\right)$. For simplicity, you can hardcode the parameter ω_d.

What changes when using the contrast sensitive Potts model? What is the intuition behind adding this new term?

YOUR ANSWER HERE

1.10 [BONUS] Iterative Segmentation

We can make the result better if we iterate the labeling process several times. Implement `iterative_opt`, a method to execute the optimization process iteratively.

Use the previously computed labeling as initial segmentation (instead of the rectangular masks we used above) and estimate new models (histograms and unaries) for foreground and background based on these. Solve the graph cut problem and use the resulting segmentation in the next iteration.

```
In [ ]: def iterative_opt(img, fg_mask, n_bins, unary_weight,
                           pairwise_edges, pairwise_costs, n_iter):
            # YOUR CODE HERE
            raise NotImplementedError()

        labels_5 = iterative_opt(
            im, graph_cut_result, n_bins, unary_weight, pairwise_edges, pairwise_costs, n_iter=
        labels_10 = iterative_opt(
            im, labels_5, n_bins, unary_weight, pairwise_edges, pairwise_costs, n_iter=5)

        fig, axes = plt.subplots(1, 3, figsize=(12,4), sharex=True, sharey=True)
        axes[0].set_title('Initial')
        axes[0].imshow(draw_mask_on_image(im, graph_cut_result))
        axes[1].set_title(f'After 5 iterations')
        axes[1].imshow(draw_mask_on_image(im, labels_5))
        axes[2].set_title(f'After 10 iterations')
        axes[2].imshow(draw_mask_on_image(im, labels_10))
        fig.tight_layout()
```

How did the labeling change? Do you have an explanation why?

YOUR ANSWER HERE

1.11 [BONUS++] Interactive Segmentation

We can get even better results by incorporating user input into the iterative segmentation process you implemented above.

Extend the given framework to allow the user to add or remove rectangular regions from the graph cut result. Then recalculate the foreground and background model according to new mask. Iterate this process until the user is satisfied with the result.

For this, look up how to create interactive graphical interfaces using Matplotlib, see for example `matplotlib.widgets.RectangleSelector` and `matplotlib.widgets.Button`.

```
In [ ]: # YOUR CODE HERE
        raise NotImplementedError()
```

03_sliding_window_detection

May 12, 2019

1 Sliding-Window Object Detection

In this exercise we will implement a simple car detector. To accomplish this, we will first implement a feature descriptor similar to the Histogram of Orientated Gradients (HOG). Then using the features computed for (small) image patches with fixed size, we will train a support vector machine (SVM) classifier, to classify whether the input patch corresponds to a car.

In the end, given a test image with arbitary shape, we will run our classifier over the image in a sliding window (patch) fashion. We will generate detections at the places where the classifier is very confident that the patch contains a car.

You may refer to the original HOG paper, or the following two tutorials, in case you want to freshen up your knowledge on HOG a little bit:

- N. Dalal and B. Triggs: Histograms of oriented gradients for human detection. CVPR 2005. http://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf
- https://www.learnopencv.com/histogram-of-oriented-gradients/
- http://mccormickml.com/2013/05/09/hog-person-detector-tutorial/

In [ ]:
```
%%html
<!-- Add heading numbers -->
<style>
body {counter-reset: section;}
h2:before {counter-increment: section;
           content: counter(section) " ";}
</style>
```

In [ ]:
```
%matplotlib notebook
import os
import glob
import cv2
import re
import time
import numpy as np
import matplotlib.pyplot as plt
import imageio
import sklearn.svm
import scipy.ndimage
```

```
def plot_multiple(images, titles=None, colormap='gray',
                  max_columns=np.inf, imwidth=4, imheight=4, share_axes=False):
    """Plot multiple images as subplots on a grid."""
    if titles is None:
        titles = [''] *len(images)
    assert len(images) == len(titles)
    n_images = len(images)
    n_cols = min(max_columns, n_images)
    n_rows = int(np.ceil(n_images / n_cols))
    fig, axes = plt.subplots(
        n_rows, n_cols, figsize=(n_cols * imwidth, n_rows * imheight),
        squeeze=False, sharex=share_axes, sharey=share_axes)

    axes = axes.flat
    # Hide subplots without content
    for ax in axes[n_images:]:
        ax.axis('off')

    if not isinstance(colormap, (list,tuple)):
        colormaps = [colormap]*n_images
    else:
        colormaps = colormap

    for ax, image, title, cmap in zip(axes, images, titles, colormaps):
        ax.imshow(image, cmap=cmap)
        ax.set_title(title)
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

    fig.tight_layout()
```

1.1 Dataset

To train our classifier, we will use the *UIUC dataset* (http://cogcomp.org/Data/Car/). Download and extract it, then use the `load_dataset` function to pre-load images (modify `dataset_dir` to the path where you extracted the dataset). The function will return three lists, containing images for positive training sample, negative training sample, and test set.

```
In [ ]: def load_dataset(dataset_dir):
            def natural_sort_key(s):
                return [float(t) if t.isdigit() else t for t in re.split('([0-9]+)', s)]

            def load_images(*path_parts):
                paths = glob.glob(os.path.join(dataset_dir, *path_parts))
                return [imageio.imread(p) for p in sorted(paths, key=natural_sort_key)]

            train_images_pos = load_images('TrainImages', 'pos-*.pgm')
            train_images_neg = load_images('TrainImages', 'neg-*.pgm')
```

```
    test_images = load_images('TestImages', 'test-*.pgm')
    assert (len(train_images_pos) == 550 and
            len(train_images_neg) == 500 and
            len(test_images) == 170)
    return train_images_pos, train_images_neg, test_images

dataset_dir = '/home/sarandi/main/teaching/cv_ss19/exercise-cv/CarData'
train_images_pos, train_images_neg, test_images = load_dataset(dataset_dir)
```

1.2 HOG-like Descriptor

First we want to implement a simple HOG-like descriptor `hoglike_descriptor()` which takes an image and computes the corresponding HOG-like representation. The function should take in following arguments:

- `image`: the grayscale image,
- `cell_size`: the size of each HOG-like cellin both dimensions,
- `n_bins` the number of bins for the gradient orientation,

The output should be a three dimensional array. The first two dimensions are the spatial indices of the HOG cell. The third dimension describes the orientation bins of the HOG descriptor. Each cell has to be independently L_2 normalized to 1. Note that the original HOG descriptor uses a more elaborate two-stage normalization scheme, that is why our version here is only called a "HOG-like" descriptor.

When the dimensions of the images are not a multiple of the `cell_size`, discard the remaining pixels to the right and to the bottom of the image.

```
In [ ]: def image_gradients_polar(image):
            filter_kernel = np.array([[-1,0,1]], dtype=np.float32)
            dx = scipy.ndimage.convolve(image, filter_kernel, mode='reflect')
            dy = scipy.ndimage.convolve(image, filter_kernel.T, mode='reflect')
            magnitude = np.hypot(dx, dy)
            direction = np.arctan2(dy, dx) # between -pi and +pi
            return magnitude, direction

        def hoglike_descriptor(image, cell_size=8, n_bins=16):
            image = image.astype(np.float32)/255
            grad_mag, grad_dir = image_gradients_polar(np.sqrt(image+1e-4))

            # YOUR CODE HERE
            raise NotImplementedError()

            # Normalization
            bin_norm = np.linalg.norm(hog, axis=-1, keepdims=True)
            return hog / (bin_norm+1e-4)
```

A simple way to visualize HOG features is to plot the 90ř rotated gradient vector for each bin, with length propotional to the value of the bin. The function `plot_hog` implements this. The 90ř rotation makes the image easier to interpret intuitively.

```
In [ ]: def plot_hog_cell(image_roi, hog_cell):
            """Visualize a single HOG cell."""
            output_size = image_roi.shape[0]
            half_bin_size = np.pi / len(hog_cell) / 2
            tangent_angles = np.linspace(0, np.pi, len(hog_cell), endpoint=False)+np.pi/2
            center = output_size / 2

            for cell_value, tangent_angle in zip(hog_cell, tangent_angles):
                cos_sin = np.array([np.cos(tangent_angle), np.sin(tangent_angle)])
                offset = cell_value * output_size * cos_sin *0.5
                pt1 = tuple(np.round(center - offset).astype(int))
                pt2 = tuple(np.round(center + offset).astype(int))
                cv2.line(image_roi, pt1, pt2, color=(0.976,0.506,0.165),
                         thickness=2, lineType=cv2.LINE_AA)

        def plot_hog(image, hog, cell_size=8):
            upsample_factor = 48 / cell_size
            result = cv2.resize(image, (0, 0), fx=upsample_factor, fy=upsample_factor,
                                interpolation=cv2.INTER_NEAREST)
            result = cv2.cvtColor(result, cv2.COLOR_GRAY2RGB).astype(np.float32)/255*0.6

            for y, x in np.ndindex(*hog.shape[:2]):
                yx = np.array([y, x])
                y0_out, x0_out = (yx * cell_size * upsample_factor).astype(int)
                y1_out, x1_out = ((yx+1) * cell_size * upsample_factor).astype(int)
                result_roi = result[y0_out:y1_out, x0_out:x1_out]
                plot_hog_cell(result_roi, hog[y, x])
            return result
```

```
In [ ]: # Two simple wave images are here to help understand the visualization
        waves = [imageio.imread('sine.png'), imageio.imread('circular_sine.jpg')]
        images = waves + train_images_pos[:6] + train_images_neg[:6]
        hogs = [hoglike_descriptor(image) for image in images]
        hog_plots = [plot_hog(image, hog) for image, hog in zip(images, hogs)]
        titles = ['Wave 1', 'Wave 2'] + ['Positive']*6 + ['Negative']*6
        plot_multiple(hog_plots, titles, max_columns=2, imheight=2, share_axes=False)
```

Describe what you see. Can you spot any interesting HOG-cells in the positive and negative example?

YOUR ANSWER HERE

1.3 Support Vector Machine for Classifying Image Windows

We now want to train a classifier in our HOG-like feature space to tell cars and non-cars apart. We use a kernel SVM with Gaussian (radial basis function, RBF) kernel for this.

Given the HOG representation of an image patch, the classifier should predict if the image patch corresponds to a car. The classifier will then be used to detect objects in new test images using sliding windows.

```
In [ ]: def train_svm(positive_hog_windows, negative_hog_windows):
            svm = sklearn.svm.SVC(C=10, probability=True, kernel='rbf', gamma='scale')
            hog_windows = np.concatenate([positive_hog_windows, negative_hog_windows])
            svm_input = hog_windows.reshape([len(hog_windows),-1])
            svm_target = np.concatenate((
                    np.full(len(positive_hog_windows), 1, dtype=np.float32),
                    np.full(len(negative_hog_windows), 0, dtype=np.float32)))
            svm.fit(svm_input, svm_target)
            return svm

        def predict_svm(svm, hog_window):
            """Return the confidence of classifying as a car."""
            return svm.predict_proba(hog_window.reshape(1, -1))[:, 1]
```

```
In [ ]: start_time = time.time()
        print('Computing features...')
        positive_hog_windows = [hoglike_descriptor(im) for im in train_images_pos]
        negative_hog_windows = [hoglike_descriptor(im) for im in train_images_neg]
        duration = time.time()-start_time
        print(f'Done. Used {duration:.2f} s.')

        start_time = time.time()
        print('Training SVM...')
        svm = train_svm(positive_hog_windows, negative_hog_windows)
        duration = time.time()-start_time
        print(f'Done. Used {duration:.2f} s.')
```

1.4 Sliding Window-based Detection

Now implement sliding window classification in the function `get_score_map`. It takes as input the trained classifier object `svm`, the HOG representation of a query image and `window_shape`, the shape of the sliding window (height, width).

The function should slide a window over the HOG representation, compute the classification score for each window location, and return a score map. Notice that the score map will not have the same shape as the input HOG representation.

Use `predict_svm(svm, hog_window)` to get classification score for a HOG window.

```
In [ ]: def get_score_map(svm, hog, window_shape):
            # YOUR CODE HERE
            raise NotImplementedError()
```

The next step is to convert the score map to actual detections. Implement the function `score_map_to_detections` which returns the indices as well as the values of scores that are higher than certain `threshold`.

```
In [ ]: def score_map_to_detections(score_map, threshold):
            # YOUR CODE HERE
            raise NotImplementedError()
            return xs, ys, scores
```

Finally, we can test our car detector!

```
In [ ]: import itertools
        def draw_detections(image, xs, ys, scores, window_shape,
                            cell_size=8):
            offset_size = 0

            h, w = image.shape
            scale_out = 5
            output_image = cv2.resize(
                image, (w*scale_out, h*scale_out), interpolation=cv2.INTER_NEAREST)
            output_image = cv2.cvtColor(output_image, cv2.COLOR_GRAY2RGB)//2

            window_size_out = np.array(window_shape[::-1]) * cell_size * scale_out
            color = (197,255,0)

            for x, y, score in zip(xs, ys, scores):
                im_p0 = (np.array([x,y]) * cell_size + offset_size) * scale_out
                im_p1 = im_p0 + window_size_out
                cv2.rectangle(output_image, tuple(im_p0), tuple(im_p1),
                              color, thickness=3, lineType=cv2.LINE_AA)
                cv2.putText(output_image, f'{score:.0%}', tuple(im_p0),
                            cv2.FONT_HERSHEY_COMPLEX, 1.5, color,
                            thickness=2, lineType=cv2.LINE_AA)
            return output_image
```

```
In [ ]: images, titles = [], []
        window_shape = positive_hog_windows[0].shape[:2]

        for test_image in test_images[:8]:
            hog = hoglike_descriptor(test_image)
            score_map = get_score_map(svm, hog, window_shape)
            xs, ys, scores = score_map_to_detections(score_map, 0.8)
            detection_image = draw_detections(
                test_image, xs, ys, scores, window_shape)

            images += [plot_hog(test_image, hog), score_map, detection_image]
            titles += ['HOG', 'Score map', 'Detections']

        plot_multiple(
            images, titles, max_columns=3,
            imheight=2, imwidth=3.5, colormap='viridis')
```

How do the results look?
YOUR ANSWER HERE

1.5 Non-Maximum Suppression

Sliding window based detectors often give multiple responses for the same target. A way to compensate such effect is to use non-maximum-suppression (NMS) on the score map. NMS simply looks at every pixel of the score map and keeps it only if it is the maximum in its 8-neighborhood (set to 0 otherwise). Implement `nms` which takes a score map, and returns the non-maximum-suppressed one.

```
In [ ]: def nms(score_map):
            # YOUR CODE HERE
            raise NotImplementedError()
```

```
In [ ]: images, titles = [], []
        for test_image in test_images[:8]:
            hog = hoglike_descriptor(test_image)
            score_map = nms(get_score_map(svm, hog, window_shape))
            xs, ys, scores = score_map_to_detections(score_map, 0.8)
            detection_image = draw_detections(
                test_image, xs, ys, scores, window_shape)

            images += [plot_hog(test_image, hog), score_map, detection_image]
            titles += ['HOG', 'Score map after NMS', 'Detections after NMS']

        plot_multiple(
            images, titles, max_columns=3,
            imheight=2, imwidth=3.5, colormap='viridis')
```

Does the result look better? Can you find examples where it fails? Can you imagine a downside of non-maximum-suppression?

YOUR ANSWER HERE

Let's evaluate the performance of our detector on the full UIUC test set. The dataset's zip contains a Java-based evaluation program, which we can call from this notebook (lines starting with ! are interpreted as shell commands).

With correct implementation the F-measure should be above 90%.

```
In [ ]: def evaluate(test_images, svm, window_shape, descriptor_func=hoglike_descriptor,
                       cell_size=8, threshold=0.8):
            # Write the detections to a file that is understood by the Java-based
            # evaluation program supplied with the dataset.
            with open('foundLocations.txt', 'w') as f:
                for i, test_image in enumerate(test_images):
                    hog = descriptor_func(test_image)
                    score_map = nms(get_score_map(svm, hog, window_shape))
                    xs, ys, scores = score_map_to_detections(score_map, threshold)

                    f.write(f'{i}: ')
                    for x,y in zip(xs, ys):
                        f.write(f'({y*cell_size}, {x*cell_size}) ')
                    f.write('\n')
```

```
        # Run the evaluation program on our generated file
        !java -classpath $dataset_dir Evaluator $dataset_dir/trueLocations.txt foundLocati

    evaluate(test_images, svm, window_shape, threshold=0.8)
```

1.6 [BONUS] Soft Assignment to Multiple Bins and Cells

In our `hoglike_descriptor`, we have used a simple assignment scheme of gradient vector to HOG bins. Each pixel voted for a single gradient orientation bin of a single spatial cell.

Now imagine if a gradient orientation falls on the end of an orientation bin. A small rotation would make it change to its neighboring bin, thus suddenly altering the HOG feature.

Similarly, imagine a pixel near the border of between HOG-cells (spatially). A small translation of the object by a few pixels would make this gradient vote in the neighboring cell, again largely changing the features.

To make our descriptor more robust to small rotations and translations, let's replace this simple assignment scheme with a smooth voting. This will distribute the gradient magnitude over neighboring bins and cells.

In particular, we will use trilinear interpolation weights for weighting votes to neighboring bins. This is analogous to bilinear interpolation for three dimensional arrays. Remember that our descriptor is a three-dimensional array, and is indexed by two spatial cell indices and an orientation bin index. (If you do not understand what is bilinear interpolation, read the first tutorial provided at the beginning of this exercise.)

Implement a `hoglike_descriptor_with_interp` function, which has same functionality and signature with `hoglike_descriptor` implemented earlier, but with simple assignment replaced with soft assignment according to trilinear interpolation weights.

```
In [ ]: def hoglike_descriptor_with_interp(image, cell_size=8, n_bins=16):
            # YOUR CODE HERE
            raise NotImplementedError()

            # Normalization
            bin_norm = np.linalg.norm(hog, axis=-1, keepdims=True)
            return hog / (bin_norm + 1e-4)
```

```
In [ ]: start_time = time.time()
        print('Computing features...')
        descriptor_func = hoglike_descriptor_with_interp
        positive_hog_windows = [descriptor_func(im) for im in train_images_pos]
        negative_hog_windows = [descriptor_func(im) for im in train_images_neg]
        duration = time.time()-start_time
        print(f'Done. Used {duration:.2f} s.')

        start_time = time.time()
        print('Training SVM...')
        svm2 = train_svm(positive_hog_windows, negative_hog_windows)
        duration = time.time()-start_time
        print(f'Done. Used {duration:.2f} s.')
```

Does it obtain better results?

```
In [ ]: evaluate(test_images, svm2, window_shape,
                hoglike_descriptor_with_interp, threshold=0.8)
```

YOUR ANSWER HERE

Congratulations, you have now implemented a car detector! To conclude, visualize it on different test images.

```
In [ ]: extra_im = imageio.imread('cars.jpg')
        images = []

        for test_image in [extra_im, *test_images[:53]]:
            hog = hoglike_descriptor_with_interp(test_image)
            score_map = nms(get_score_map(svm2, hog, window_shape))
            xs, ys, scores = score_map_to_detections(score_map, 0.8)
            detection_image = draw_detections(
                test_image, xs, ys, scores, window_shape)
            images.append(detection_image)

        plot_multiple(images, max_columns=3, imheight=2, imwidth=3.5)
```