UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
SCHOOL OF COMPUTER SCIENCE & APPLIED MATHEMATICS

COMS4036A & COMS7050A
Computer Vision

Nathan Michlo, Devon Jarvis, Dr Richard Klein

Lab 4: Edges, Corners and Descriptors

Overview

In this lab, we will investigate edge detectors, corner detectors and image descriptors. In particular the Canny edge detector, Harris corner detector and histogram of oriented gradients descriptor.

- From Lab 1 you should have your 3 images of puzzle pieces with their own corresponding hand-made masks. For this lab, you will need to read in one of these images and the corresponding mask to perform image processing operations on it.
- For question 1 and 2 you should downscale your image and mask for memory, computational and comparative reasons, from a size in pixels of 5120×3840 to 640×480, for question 3 you should downscale your images further to a size in pixels of 256×192.

  Hint: you should use `cv2.resize(img, (width, height))`.

  Note: Do not threshold your mask once downscaled for this lab, we want anti-aliased edges.
- Be wary of the datatype of your images being processed, silent errors can occur if you don't cast an image to float32 from uint8 before processing, or threshold values can be incorrectly scaled. `cv2.filter2D` is one notable example of where things can go wrong.

  Hint: While integer images have values in the range $[0, 255]$, float images should have values in the range $[0, 1]$. A direct cast between datatypes will not preserve this relationship, instead have a look at `skimage.img_as_float` for when the image is read in.

1 Canny Edge Detector

In this section, you will implement your own version of the Canny edge detector, as well as investigate the effects of the algorithm's different tuneable parameters.

1.1 Implement the Canny edge detection algorithm from scratch, your function should accept a greyscale image and three tuneable named parameters (defaults are given below): σ, $low\_threshold$ and $high\_threshold$, similar in behaviour to that of the skimage canny edge detector.

- Remove any unnecessary noise by applying a Gaussian filter with $\sigma = 4$ to your greyscale image. Estimate the size of your Gaussian filter with the equation $size = 2 \cdot radius + 1$, where $radius = floor(truncate \cdot \sigma + 0.5)$, and $truncate = 4.0$ is the number of standard deviations away to truncate the filter.

  Hint: Use nearest/replicate for the border padding - This is the same way skimage does it and we are going to compare later!

- Calculate the intensity gradient of the image, comprised of orientation and magnitudes, using vertical and horizontal Sobel filters.

  Hint: use np.arctan2 instead of np.arctan to avoid incorrect values or divisions by zero, and np.hypot to avoid any potential precision errors. Use reflect for the border padding mode.

- Apply non-maximum suppression to get rid of any unwanted pixels which may not form part of an edge, leaving you with the "thin edges".
- Apply double thresholding to determine potential edges. Use the weak/low threshold value of 0.1 and strong/high threshold value of 0.2.
- Track edges by Hysteresis, where weak edges are suppressed if not connected to strong edges.

1.2 Plot and label the results after each step in the Canny algorithm, include the blurred greyscale image (labelled with the estimated size of your Gaussian filter), horizontal and vertical Sobel filtered images, orientation and magnitude images, non-maximum suppressed image, double thresholded images, and the final result after Hysteresis. Make sure to label each image.

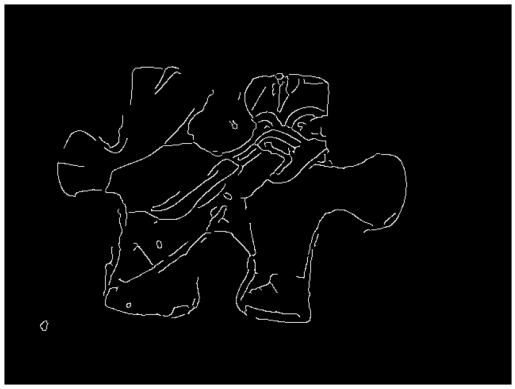

Figure 1: Example edges detected using Canny.

1.3 Compare your final result to the skimage canny function (`skimage.feature.canny`) run with default arguments.

Hint: Read the docs[1] and tune your algorithm's arguments so the outputs are as close as possible.

1.4 Give reasons for the effect on the output after increasing or decreasing each parameter, including $low\_threshold$, $high\_threshold$ and σ. Plot your different parameter traversals used to come to your conclusions. You may use the `skimage.feature.canny` function for this question.

1.5 Adjust the parameters of your implementation to try and obtain the best or cleanest result. Plot this result with your chosen parameters. Again you may use the `skimage.feature.canny` function.

[1] https://scikit-image.org/docs/dev/api/skimage.feature.html#skimage.feature.canny

2 Harris Corner Detector

2.1 Your task is to implement the Harris Corner Detector, your function signature should accept a greyscale image with three additional tuneable named parameters (defaults are given below): σ, κ and τ, where σ is the standard deviation of the Gaussian filter representing the weight matrix, κ is the sensitivity factor that separates corners from edges and τ is the normalised response threshold.

- Compute your vertical and horizontal image derivatives from your greyscale image using the appropriately oriented Sobel filters and zero padding.
- Use these derivatives to compute your image structure tensors ($\mathbf{S}_{ij}$ in your textbook), weighted according to the Gaussian with default $\sigma = 1$. Again use zero padding.

  Hint: Estimate the size of your Gaussian filter the same way you did for question 1.1. You do not need to compute and store the structure tensors for each pixel as matrices as shown in the textbook, it will be easier to decompose the matrix into its three unique components that can be convolved separately, while still being used as arguments for the next step.
- Calculate the responses (c_{ij}) from the structure tensors, with the default value for $\kappa = 0.05$.

  Note: Do not use the erroneous formula from the textbook! Instead use:
  $c_{ij} = \lambda_1\lambda_2 - \kappa(\lambda_1 + \lambda_2)^2 = \det[\mathbf{S}_{ij}] - \kappa \cdot \text{trace}[\mathbf{S}_{ij}]^2$

  Hint: Following on from the previous hint, if you have the 2×2 matrix $\begin{bmatrix} a & b \\ b & c \end{bmatrix}$ the determinant is $ac - b^2$ and the trace is $a + c$.
- Remove spurious corners from the responses by only keeping local maxima, do this by analysing the 8 neighbouring values of the centre value in a 3x3 sliding window and keeping the centre only if it is greater than its neighbours.

  Hint: Be careful not to change the image itself while you are using the sliding window.
- Use the default threshold ratio $\tau = 0.05$ to return the points where $\frac{responses}{max(responses)} > \tau$.

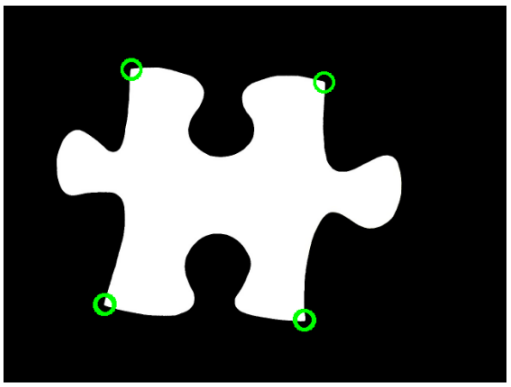

Figure 2: Example mask with corners detected.

2.2 Apply your Harris corner detector to the mask image with default arguments and plot the detected corners by drawing circles around them.

Hint: draw a circle at x, y with `cv2.circle(corner_im, (x, y), radius=12, color=(0,255,0), thickness=2)`

2.3 Test different values of $\kappa \in \{0.025, 0.05, 0.1, 0.2\}$, $\sigma \in \{1, 2, 4, 8\}$ and $\tau \in \{0.01, 0.05, 0.1, 0.2\}$ with your implementation of the Harris corner detector. You do not need to run through all permutations of parameters, rather plot a traversal for each parameter and keep the remainder as the defaults (one plot for each parameter, with a labelled image corresponding to each parameter value in the set). Perform these tests on the greyscale version of the original image, not your mask. Draw the points on your images as in the previous question.

2.4 How could you further reduce the number of points that are closely clustered together?

3 Histogram of Oriented Gradients Descriptor

For this section, you will use the RGB image of your original puzzle piece, but instead downscaled to a size of 256×192 pixels to be used for calculating the Histogram of Oriented Gradients (HoG) Descriptor. (The descriptor was originally designed to detect humans and operated on images at a size of 64×128 pixels, but we want pretty plots!)

3.1 Your final task is to implement the Histogram of Oriented Gradients Descriptor (HoG). Create a function that accepts an **RGB** image and three named arguments $orientations = 9$, $pixels\_per\_cell = 8$ and $cells\_per\_block = 2$, the cell size is the number of pixels along one axis of a square cell, and the block size is the number of cells along one axis of a square block.

- Make sure that the input image is valid to avoid any errors later. **`assert`**[2] that the width and height of the image in pixels is a multiple of $pixels\_per\_cell$, and that the width and height are at least $cells\_per\_block \cdot pixels\_per\_cell$ pixels in size.
- Compute the orientation and gradient magnitude image, this is similar to question 1.1, however use the non-square first derivative filters, $[-1, 0, 1]$ and $[-1, 0, 1]^T$, on each channel in the RGB image with reflect padding. The final magnitude of the gradient at a pixel is the maximum of the magnitude of gradients of the three channels, and the angle is the angle from the channel corresponding to the maximum gradient.

  Hint: the numpy functions `np.argmax` and `np.take_along_axis` may come in handy.
- Generate a histogram for each cell in the image by performing orientation binning using a weighted voting procedure. The weighted voting procedure splits the magnitude of the current pixel in the cell between the nearest two bins based on the ratio between the bins of the orientation of that pixel. Orientations are unsigned and are binned between $0°$ and $180°$, the number of bins is controlled by the $orientations$ parameter (for example if there are 9 bins they are centred on $0°, 20°, ..., 140°, 160°$). Remember to use wrap-around so that bin $0°$ equals bin $180°$ (For example if a pixel's orientation is $175°$ and its magnitude is 100, it contributes 25 to the total of bin $160°$ and 75 to the total of bin $0°$).

  Hint: you can use a 3D array to store these histograms, treating the first two axes as the cell y & x coordinates and the last as the histogram bins. Bonus points if you can calculate the histogram of a single cell purely with numpy operations and no python loops!
- Generate descriptor blocks by L2 normalising the concatenated histograms of the cells within a $cells\_per\_block \times cells\_per\_block$ sliding window over all the cell histograms. Per axis there should be $cells\_in\_axis - (cells\_per\_block - 1)$ blocks generated as they are overlapping. Normalisation is performed over the entire concatenated vector of cell histograms in the block ie. $normalised\_block = \frac{block}{||block||}$.

[2] **`assert`** is a python keyword useful for debugging code.

Hint: you can use slicing to extract a block of cells from your cell array and use `np.flatten` to obtain your unnormalised block vector/descriptor.

- Finally generate your image descriptor as the flattened vector of all your normalised block descriptors. This vector is meant to be quite large!

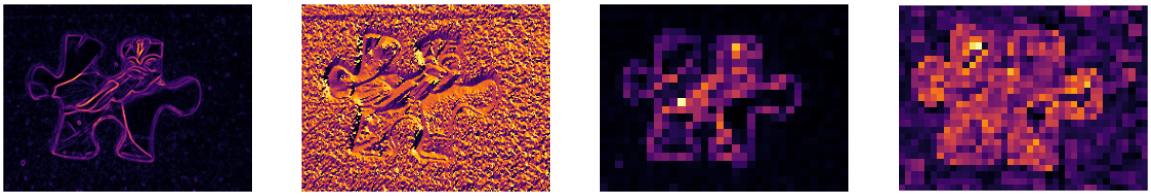

Figure 3: From left to right: max gradient magnitudes, corresponding max orientations, cell histogram maxes, block descriptor maxes. (All images are contrast stretched.)

3.2 Plot the following steps of your HoG function called with default values as in figure 3, including the maximal gradient magnitude and orientation images (these images should be the original image in size), the image formed by taking the max value of each cell's histogram, and lastly the image formed by taking the max value of each block's normalised descriptor vector. Make sure to label and give the width and height of each image.

Hint: remember to contrast stretch your plotted images and use `cmap="inferno"` for matplotlib. The reason we are taking the max over the cell histograms and the max over the block descriptors is not necessarily because that is useful in practice, but because otherwise there would just be too many images to put in your report. For your own interest have a go at plotting each bin/descriptor element separately as an image.

3.3 For the previous question, what is the shape/size of your final image descriptor?

3.4 Outline a method for using the HoG descriptor to detect puzzle pieces in images. Think about what training samples you would need, the shape and size of the images, how you would break up larger images, and how you would detect puzzle pieces at different scales.

3.5 Is the HoG descriptor a good method for detecting puzzle pieces in the previous question, give reasons for why or why not - If not, can you suggest an alternative?

4 Submission

Submit a Jupyter Notebook (and corresponding PDF version) containing the images, code and results from the tasks above. At the end of the notebook briefly[3] comment on the algorithms and discuss your results. How might these algorithms contribute to Tino's puzzle-solving system? What problems do you still envisage we will need to solve before we can complete the system?

[3] Think half a page of text.