SCC5830/0251 — Prof. Moacir Ponti

Teaching Assistant: Leo Sampaio Ferraz Ribeiro

Teaching Assistant: Flavio Salles

Assignment 5: image descriptors

Code the assignment by yourself. Ask if you need help. Plagiarism is not tolerated.

1 Introduction

1.1 Task

In this assignment you have to implement a method to find a small object in a bigger image (like 'Where's Wally?'). We will use Image Descriptors for this, specifically, you'll have to implement **extractors for colour, texture and gradient descriptors** and then use them to compare the object's image to patches of the larger image. Read the instructions for each step. Use `python 3` and the libraries `numpy`, `imageio` and `scipy` to complete the task.

Follow the instructions carefully:

1. **Read the parameters**
2. **Load** the object image g
3. **Transform the image** to black&white using the Luminance technique
4. **Quantise** the image to use only b bits
5. **Compute each of the three image descriptors**
   a) Normalised Colour Histogram
   b) Haralick Texture Descriptors
   c) Histogram of Oriented Gradients
6. **Concatenate the descriptor** to obtain the final descriptor for g, d_g
7. **Load** the large image f
8. **Break it in patches** and repeat steps 3-6 for each patch to obtain descriptors $d_{f,i} \in \{d_{f,1}, d_{f,2}, ..., d_{f,n}\}$
9. **Find the object** by comparing the object against each image descriptor
10. **Print** the location of the object

1.2 Input Parameters

The following parameters will be input to your program in the following order through `stdin`, as usual for run.codes:

1. filename for the image with the object f,
2. filename for the larger image g,
3. quantisation parameter b, determining the number of bits after quantisation

2 Preprocessing and Quantisation

Before computing the image descriptors we need to (1) transform the images to black&white and (2) quantise the image. For (1) we are going to use the common *Luminance* method, for each RGB pixel:

$$f(x, y) = \lfloor 0.299R + 0.587G + 0.114B \rfloor,$$

where $\lfloor . \rfloor$ is the floor operation and letters R, G, B represent each colour channel.

Then we need to quantise our image to only use b bits, per the input parameter. This procedure will help make the colour descriptor more space-efficient later on. You should use the bit shift operator to the right to compress the range to $[0, 2^b - 1]$.[1].

3 Image Descriptors

As we've seen in class, the colour intensity information on images is seldom helpful in representing any semantic information. With this in mind, it is desirable to design image descriptors that, by looking at the intensities, are able to better capture semantic information. For this assignment we are going to implement three image descriptors, each of which will yield a vector that will compose our final descriptor. This will allow us to compare images in a space where distances have semantic significance.

The descriptors we are going to compute are (i) the Normalised Histogram, (ii) Haralick Texture Descriptors and (iii) Histogram of Oriented Gradients.

3.1 Normalised Histogram

The normalised histogram is a good compact size-independent representation of the intensities in an image. After normalisation the histogram can be read as a probability distribution determining how likely each colour is in an image. Normalise using:

$$p(k) = \frac{h(k)}{\sum_{k=0}^{2^B-1} h(k)},$$

[1] Not that this is slightly different than what we did back in Assignment 01. Here we want to compress the range and leave it compressed, whilst in A01 we wanted to drop the bits but keep the $[0, 255]$ range

where $k \in [0, 2^B - 1]$ are the intensities (after quantisation), $h(k)$ is how many times that intensity is present in the image (i.e. the histogram). It is also possible to simply divide $h(k)$ by the total number of pixels in the image.

This descriptor will be a vector with B^2 values, which we will call d_c. Normalise d_c by dividing it by its magnitude (you can use `np.linalg.norm`).

3.2 Haralick Texture Descriptors

For our texture descriptors we are going to use the Haralick Texture Descriptors. The main idea behind this is to gather metrics from an *intensity level co-ocurrence matrix*.

3.2.1 Intensity-level Co-ocurrence Matrix

Let's unpack what a *intensity level co-ocurrence matrix* is. We'll call our matrix C, each value $c_{i,j}$ will be a count of the co-ocurrences of the pair of intensities i and j in the image. For example, $c_{2,245} = 2$ means that the intensities 2 and 245 co-ocurred two times on the image.

How do we define a co-ocurrence? There are of course many ways (at least 4) to do so taking a neighbourhood into account, but for this assignment *we are going to count a co-ocurrence when i and j are diagonally connected (one step to the right, one down).* Here is an example with a full image, consider that intensities are in the range $[0, 2]$:

$$g = \begin{bmatrix} 1 & 1 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 \\ 1 & 1 & 0 & 2 & 2 \\ 0 & 0 & 2 & 0 & 2 \\ 1 & 2 & 2 & 2 & 2 \end{bmatrix}$$

our intensity level co-ocurrence matrix would be:

$$C = \begin{bmatrix} 1 & 0 & 3 \\ 2 & \mathbf{3} & 2 \\ 0 & 0 & 5 \end{bmatrix}$$

where we highlighted the co-ocurrences where $i = 1$ and $j = 1$, which totals three and result in $C[1, 1] = 3$.

Note that it is not possible to compute co-ocurrences on the right and bottom borders, this is ok, you can skip those. To finish our matrix then, we divide every value in C by the total sum of values:

$$C_n = \begin{bmatrix} 1/16 & 0 & 3/16 \\ 1/8 & 3/16 & 1/8 \\ 0 & 0 & 5/16 \end{bmatrix}$$

3.2.2 Computing Descriptors

Now taking matrix C_n into consideration, we are going to compute 5 metrics and take those as our texture descriptors d_t:

1. Energy:
$$\sum_{i=0}^{N}\sum_{j=0}^{N} C_n(i,j)^2$$

2. Entropy:
$$-\sum_{i=0}^{N}\sum_{j=0}^{N} C_n(i,j)\log[C_n(i,j)+\varepsilon],$$
where $\varepsilon = 0.001$ is a tiny value to avoid $\log(0)$.

3. Contrast:
$$\frac{1}{(N)^2}\sum_{i=0}^{N}\sum_{j=0}^{N}(i-j)^2 C_n(i,j)$$

4. Correlation:
$$\frac{\sum_{i=0}^{N}\sum_{j=0}^{N} ijC_n(i,j)-\mu_i\mu_j}{\sigma_i\sigma_j},$$
where $\sigma_i, \sigma_j > 0$, otherwise consider correlation to be null (0). Plus, $\sigma_i, \sigma_j, \mu_i$ and μ_j are the directional means and standard deviations, computed as:
$$\mu_i = \sum_{i=0}^{N} i\sum_{j=0}^{N} C_n(i,j) \quad \mu_j = \sum_{j=0}^{N} j\sum_{i=0}^{N} C_n(i,j)$$
$$\sigma_i = \sum_{i=0}^{N}(i-\mu_i)^2\sum_{j=0}^{N} C_n(i,j) \quad \sigma_j = \sum_{j=0}^{N}(j-\mu_j)^2\sum_{i=0}^{N} C_n(i,j)$$

5. homogeneity:
$$\sum_{i=0}^{N}\sum_{j=0}^{N}\frac{C_n(i,j)}{1+|i-j|}$$

After computing all metrics, normalise d_t by dividing it by its magnitude (you can use `np.linalg.norm`).

3.3 Histogram of Oriented Gradients

This descriptor is a good way of capturing how the textures in an image are "arranged" by looking at the angle of the gradients (the *rate-of-change* from one pixel to the next at a certain angle). To compute this we must first obtain the gradient of the image in both x and y directions. This can be done by performing convolution (use

`scipy.ndimage.convolve` because otherwise run.codes will timout) of an image with the sobel operator in each direction:

$$w_{Sx} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \; ; \; w_{Sy} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

After computing cross-correlation between the image and each filter, we will have the image gradients g_x and g_y. Using those matrices, we can separate the gradient's magnitude and angle into two other matrices. The magnitude can be computed:

$$M(x,y) = \frac{\sqrt{g_x(x,y)^2 + g_y(x,y)^2}}{\sum_x \sum_y \sqrt{g_x(x,y)^2 + g_y(x,y)^2}}$$

and the angles:

$$\phi(x,y) = \tan^{-1} \frac{g_y(x,y)}{g_x(x,y)}$$

Now, we need to use those angles to accumulate magnitudes in a binned fashion. First, **(i)** sum $\pi/2$ to all values in ϕ to make the angle range shift from $[-\pi/2, \pi/2]$ to $[0, \pi]$, then, **(ii)** convert the angles from radians to degrees (see `np.degree`); **(iii)** digitise the angles into 9 bins, this means slicing the angle range in 20 degree intervals $[0, 19], [20, 39], ...$ and then for each $\phi(x, y)$, determine in which of the 9 bins the value falls into.

We want to use the bins determined for each x, y position to accumulate the magnitudes. Let's consider an example with post-shift and conversion matrix:

$$\phi = \begin{bmatrix} 10 & 23 & 34 \\ 43 & 12 & 91 \\ 0 & 92 & 1 \end{bmatrix}$$

And the magnitude matrix:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 1 & 2 & 3 \end{bmatrix}$$

Computing the bins for each position we would have:

$$\phi_d = \begin{bmatrix} 0 & 1 & 1 \\ 2 & 0 & 4 \\ 0 & 4 & 0 \end{bmatrix}$$

where, for example, $\phi_d(1, 2) = 4$ because $\phi(1, 2) = 91$ falls into the range $[80, 99]$, which is bin 4. Now, using this setup to accumulate magnitudes using the bins, we'd have a resulting descriptor:

$$d_g = \begin{bmatrix} 7 & 5 & 3 & 0 & 3 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where, for example, $d_g(1) = 5$ because positions $\phi_d(0, 1)$ and $\phi_d(0, 2)$ are in bin 1 and

the magnitude for those positions is $M(0,1) = 2$, $M(0,2) = 3$.

This descriptor is then a vector with 9 dimensions, d_g. Normalise d_g by dividing it by its magnitude (you can use `np.linalg.norm`). All of those normalisations help to make the distance take each descriptor into account, if we'd normalised them all together, d_t has less dimensions and would "matter less" in the computation.

NOTE: In this example there will be some divisions by zero that will result in infinity. This is ok as `np.arctan` works with infinity. Your code will however spit some warnings that would make run.codes unhappy. Fix it by suppressing the warnings with `np.seterr(divide='ignore', invalid='ignore')`.

4 Finding Our Object

Now, as described in the steps in the Introduction, we want to do everything we did for the object image g on patches of the bigger image f. We want to compute the descriptors for patches of f sized 32×32 using a sliding window with stride 16; this means that each subimage will be 32×32 and will overlap 50% with the previous and next one since we will take 16-sized steps for each window.

These sliding windows will yield $W \times W$ windows, where $W = \lfloor C/32 \rfloor$, C being the size of the large image f. For each window, we want to follow the same steps as we did for g to obtain descriptors d_c, d_t, d_g. Concatenate the three vectors to build d for the object image g and d_i for each window i of the larger image f. Then we want to compare the descriptors for g against all descriptors for the windows:

$$D(d, d_i) = \sqrt{\sum_x (d(x) - d_i(x))^2}$$

Determine the closest window using the computed distances and print the window's coordinates separated by a space. For example, if the closest window was the second window in the third row, print `2 1`. You did it! You have found the object, congratulations!

5 Visualise Your Results

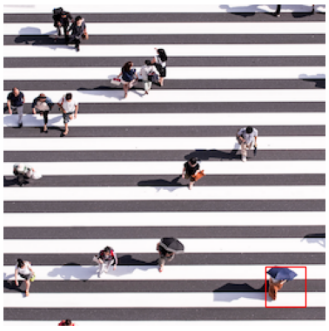
If you want to see your results (instead of just numbers), try plotting a bounding box around the area you've found. The results for most cases should be pretty good, but there are failure cases (no method is perfect). Use the following code to plot:

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
fig, ax = plt.subplots()
ax.imshow(search_img)
rect = patches.Rectangle((i * 16, j * 16), 32, 32,
                         linewidth=1, edgecolor='r', facecolor='none')
```

```
ax.add_patch(rect)
plt.show()
```

where `i` and `j` are the indices of the window you've selected. Some correct results are shown in Figure 1.

Found Wally!



Found Blue Umbrella!

Failed weird Pokeball

Found Pumpkin

Figure 1: Examples of

6 Submission

Submit your source code to run.codes (only the `.py` file).

1. **Comment your code**. Use a header with name, USP number, course code, year/semestre and the title of the assignment. If you are in a group, include all the info for both and submit the assignment under both of your accounts. A penalty on the evaluation will be applied if your code is missing the header and comments.

2. **Organize your code in programming functions**.