Project 2 Document

Weicheng Dai

1. The dependencies

The important dependencies are listed below:

1.1. Jupyter notebook version 6.4.4

1.2. Python version 3.9.7.

1.3. NumPy version 1.21.2

1.4. matplotlib version 3.4.3

1.5. PIL version 8.3.2

1.6. sklearn version 1.0.1

2. Instructions on using the script

There are 9 cells in the codes, which are marked by indexes, as can be seen in Figure 1. When user wants to use this script, simply click ‘run all’ and it will automatically execute each function in order.

1. The useful packages

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
# import cv2
import os
from sklearn.preprocessing import normalize
```

2. This is the function that converts RGB to gray level

In [2]:
```
def toGray(img):
  """ convert RGB img to gray level img

  Parameters
  ----------
  img: {ndarray} shape = [row * col]
        the input image which is RGB

  Returns
  -------
```

Figure 1: Example of indexes

2.1 Cells from # 1 to # 8

The cells from # 1 to # 8 each defines a function which will be helpful in this process. Their functions are listed below:

cell#1: import packages

cell#2: the function that converts RGB images to gray level ones

cell#3: the function that computes gradients of both horizontal and vertical direction

cell#4: the function that computes gradient magnitude and gradient angle

cell#5: the function that computes the orientation gradient for each pixel

cell#6: the function that cumulates orientation gradient of pixels to form blocks

cell#7: the function that computes the distance between two feature maps, which is intersection.

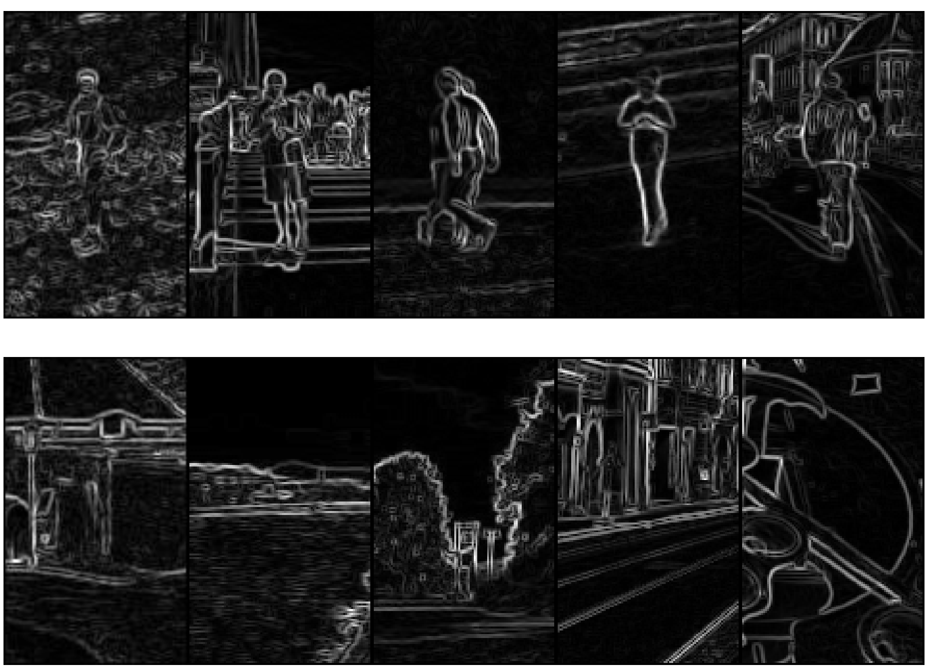
cell#8: the function that processes the training data

2.2 Cell # 9

The cell that processes the testing data and print the results.

3. Results

The gradient magnitude images for the 10 test images:



The table:

Test image	Correct Classification	File name of 1st NN, distance & classification	File name of 2nd NN, distance & classification	File name of 3rd NN, distance & classification	Classification from 3-NN
crop001034b	Human	crop001275b 0.6533 Human	crop001028a 0.592 Human	00000093a_cut 0.5537 No-human	Human
crop001070a	Human	crop001275b 0.5405 Human	00000093a_cut 0.4954 No-human	crop001028a 0.481 Human	Human
crop001278a	Human	crop001275b 0.4878 Human	crop001028a 0.4124 Human	no_person__no_bike_247_cut 0.4065 No-human	Human
crop001500b	Human	no_person__no_bike_247_cut 0.342 No-human	crop001275b 0.3027 Human	crop001028a 0.2256 Human	Human
person_and_bike_151a	Human	crop001275b 0.527 Human	no_person__no_bike_247_cut 0.515 No-human	crop001028a 0.4836 Human	Human
00000003a_cut	No-human	no_person__no_bike_247_cut 0.5156 No-human	crop001275b 0.51 Human	00000093a_cut 0.451 No-human	No-human
00000090a_cut	No-human	crop001275b 0.342 Human	00000053a_cut 0.3257 No-human	no_person__no_bike_247_cut 0.2832 No-human	No-human
00000118a_cut	No-human	no_person__no_bike_219_cut 0.4783 No-human	crop001275b 0.4722 Human	00000093a_cut 0.455 No-human	No-human
no_person_no_bike_258_ cut	No-human	crop001275b 0.4377 Human	no_person__no_bike_247_cut 0.3804 No-human	crop001028a 0.365 Human	Human
no_person_no_bike_264_ cut	No-human	no_person__no_bike_247_cut 0.4043 No-human	crop001275b 0.3838 Human	00000093a_cut 0.3677 No-human	No-human

4. Codes

```
import numpy as np

import matplotlib.pyplot as plt
```

```
from PIL import Image

# import cv2

import os

from sklearn.preprocessing import normalize


def toGray(img):

  """ convert RGB img to gray level img


  Parameters

  ----------

  img: {ndarray} shape = [row * col]

        the input image which is RGB


  Returns

  -------

  ans: {ndarray} shape = [row * col]

        the output image which is gray level


  """

  ans = np.zeros([img.shape[0],img.shape[1]], dtype = np.float16)


  # The sequence is R, G, B

  for i in range(img.shape[0]):

    for j in range(img.shape[1]):
```

```
            ans[i,j] += round(0.299 * img[i, j, 0] + 0.587 * img[i, j, 1] +  0.114 * img[i, j, 2])

    return ans

def gradient_operation(img):
    """ compute the horizontal and vertical gradient

    Parameters
    ----------
    img: {ndarray} shape = [row * col]
        the input image that we want to compute the gradient

    Returns
    -------
    grad_hori: {ndarray} shape = [row * col]
            the horizontal gradient, same shape as the input

    grad_vert: {ndarray} shape = [row * col]
            the vertical gradient, same shape as the input
    """

    # The Prewitt operator with vertical and horizontal orientation
    Prewitt_X = np.array([[-1, 0, 1],
                          [-1, 0, 1],
```

```
                      [-1, 0, 1]], dtype=np.float16)


Prewitt_Y = np.array([[1, 1, 1],

                      [0, 0, 0],

                      [-1, -1, -1]], dtype=np.float16)


# The answers initialized with all 0s, same shape as the input image

grad_hori = np.zeros([img.shape[0],img.shape[1]], dtype = np.float16)


grad_vert = np.zeros([img.shape[0],img.shape[1]], dtype = np.float16)


# The procedure of doing the convolution

# Since the two operators are of the same shape, we can do it with one iteration

for i in range(img.shape[0]):

  for j in range(img.shape[1]):

    for m in range(Prewitt_X.shape[0]):

      for n in range(Prewitt_X.shape[1]):

        if(i - Prewitt_X.shape[0] // 2 < 0 or i + Prewitt_X.shape[0] // 2 >= img.shape[0] or

           j - Prewitt_X.shape[1] // 2 < 0 or j + Prewitt_X.shape[1] // 2 >= img.shape[1]):

          continue

        else:

          grad_hori[i, j] += Prewitt_X[m, n] * img[i - 1 + m, j - 1 + n]

          grad_vert[i, j] += Prewitt_Y[m, n] * img[i - 1 + m, j - 1 + n]
```

```
    return grad_hori, grad_vert


def generate_magnitude_direction(grad_hori, grad_vert):
    """ generates gradient magnitude and gradient angle
    Parameters
    ----------
    grad_hori: {ndarray} shape = [row * col]
            the horizontal gradient of an image

    grad_vert: {ndarray} shape = [row * col]
            the vertical gradient of an image

    Returns
    -------
    gradient: {ndarray} shape = [row * col]
            the gradient magnitude of an image at each pixel

    direction: {ndarray} shape = [row * col]
            the gradient angle of an image at each pixel
    """

    # np.hypot does (x^2 + y^2)^(0.5) at each pixel
    gradient = np.hypot(grad_hori, grad_vert)
```

```
  # np.arctan2 generates the answer within the range [-pi, pi], and we convert it into [0, 180]
  direction = (np.arctan2(grad_vert, grad_hori) * 180 / np.pi) % 180

  return gradient, direction

def OG(gradient, direction):
    """ finds the corresponding orientation gradient for each pixel
    Parameters
    ----------
    gradient: {ndarray} shape = [row * col]
            the gradient magnitude of an image at each pixel

    direction: {ndarray} shape = [row * col]
            the gradient angle of an image at each pixel

    Returns
    -------
    orientation_gradient: {ndarray} shape = [row * col * 9]
                    the orientation gradient of an image
    """

    orientation_gradient = np.zeros([gradient.shape[0], gradient.shape[1], 9], dtype = np.float16)

    for i in range(gradient.shape[0]):
```

```
        for j in range(gradient.shape[1]):
            cur_class = int(direction[i, j] // 20) # where the current class is, should be 0~8
            if(cur_class == 9):
                cur_class-=1
            pivot = direction[i, j] % 20 # use pivot to find another class
            if(pivot<10):
                # use mod to prevent edge situation
                # cur_weight is computed by finding the distance with current pivot
                # but the true current weight is actually another_weight, because we have to take the inverse
value
                # another_weight + cur_weight == 20
                another_class = (cur_class - 1) % 9
                cur_weight = 10 - pivot
                another_weight = 10 + pivot
            else:
                another_class = (cur_class + 1) % 9
                cur_weight = pivot - 10
                another_weight = 30 - pivot

            orientation_gradient[i, j, cur_class] += gradient[i, j] /20 * another_weight
            orientation_gradient[i, j, another_class] += gradient[i, j] /20 * cur_weight

    return orientation_gradient
```

```
def feature(orientation_gradient):
    """ finds the cumulated orientation gradient for block

    Parameters

    ----------

    orientation_gradient: {ndarray} shape = [row * col * 9]

                    the orientation gradient of an image


    Returns

    -------

    feature_map : {ndarray} shape = [36 * n], n is the number of overlapping blocks

            the feature map of a given img

    """


    cell_size = 8

    block_size = 16

#     print(orientation_gradient.shape[0]) # 160

#     print(orientation_gradient.shape[1]) # 96


    # first we compute the feature map per cell

    # num_rows and num_cols is the size of feature per cell

    num_rows = int(orientation_gradient.shape[0] / cell_size) # 20

    num_cols = int(orientation_gradient.shape[1] / cell_size) # 12


    # this is the num of cols in the whole feature map
```

```
    num_blks = (num_rows - 1) * (num_cols - 1)

    feature_cell = np.zeros([num_rows, num_cols, 9], dtype = np.float16)
#     print(feature_cell.shape[0]) # 20
#     print(feature_cell.shape[1]) # 12

    # accumulate the orientation gradient of each cell
    for i in range(0, orientation_gradient.shape[0]-cell_size + 1, cell_size):
        for j in range(0, orientation_gradient.shape[1]-cell_size + 1, cell_size):
            for k in range(cell_size):
                for m in range(cell_size):
                    for d in range(9):
                        feature_cell[int(i/cell_size), int(j/cell_size), d] += orientation_gradient[(i+k), (j+m), d]

    # use the orientation gradient of each cell to form the blks'
    feature_map = np.zeros([36, num_blks], dtype = np.float16)
    for i in range(0, num_rows-1, 1):
        for j in range(0, num_cols-1, 1):
            for k in range(9):
                feature_map[k, i*(num_cols-1)+j] = feature_cell[i, j, k]
                feature_map[k+9, i*(num_cols-1)+j] = feature_cell[i+1, j, k]
                feature_map[k+18, i*(num_cols-1)+j] = feature_cell[i, j+1, k]
                feature_map[k+27, i*(num_cols-1)+j] = feature_cell[i+1, j+1, k]
```

```
    # use l2 norm
    feature_map = normalize(feature_map, axis=0, norm='l2')

    return feature_map

def distance(map1, map2):
    """computes the distance between two feature maps

    Parameters
    ----------
    map1: {ndarray}, shape = [36 * n], n is the number of overlapping blocks
        the test img

    map2: {ndarray}, shape = [36 * n], n is the number of overlapping blocks
        the train img

    returns
    -------
    ans: float, the IOU of two maps
    """

    numerator = np.sum(np.minimum(map1, map2))
    denominator = map2.sum()
```

```
        return numerator/denominator


    def process_data():
        """Deals with the training images, save files and export features


        Returns
        -------
        training_Pos: list, shape = [m1 * 36 * n], n is the number of overlapping blocks, m1 is the number of positive training img


        training_Neg: list, shape = [m2 * 36 * n], n is the number of overlapping blocks, m2 is the number of negative training img
        """
        training_Pos = []
        # for each file in Positive training file, execute the functions above in order.
        for filename in os.listdir("./Training images (Pos)"):
            img = plt.imread("./Training images (Pos)" + "/" + filename)
            img = toGray(img)
            grad_hori, grad_vert = gradient_operation(img)
            gradient, direction = generate_magnitude_direction(grad_hori, grad_vert)
            orientation_gradient = OG(gradient, direction)
            feature_map = feature(orientation_gradient)
            training_Pos.append(feature_map)
            # for those whose HOG should be saved, execute this separately.
            if(filename[:-4] == 'crop001028a' or filename[:-4] == 'crop001030c'):
```

```
        fo = open('pos_{}_lines.txt'.format(filename[:-4]), "w")
        for i in range(feature_map.shape[0]):
            for j in range(feature_map.shape[1]):
                fo.write(str(feature_map[i, j])+"\n")
        fo.close()


# for each file in Negative training file, execute the functions above in order.
training_Neg = []
for filename in os.listdir("./Training images (Neg)"):
    img = plt.imread("./Training images (Neg)" + "/" + filename)
    img = toGray(img)
    grad_hori, grad_vert = gradient_operation(img)
    gradient, direction = generate_magnitude_direction(grad_hori, grad_vert)
    orientation_gradient = OG(gradient, direction)
    feature_map = feature(orientation_gradient)
    training_Neg.append(feature_map)
    # for those whose HOG should be saved, execute this separately.
    if(filename[:-4] == '00000091a_cut'):
        fo = open('neg_{}_lines.txt'.format(filename[:-4]), "w")
        for i in range(feature_map.shape[0]):
            for j in range(feature_map.shape[1]):
                fo.write(str(feature_map[i, j])+"\n")
        fo.close()
```

```
    return training_Pos, training_Neg


# first retrieve the training dataset with the function above
training_Pos, training_Neg = process_data()
training_Pos = np.array(training_Pos) # shape = [m1 * 36 * n]
training_Neg = np.array(training_Neg) # shape = [m2 * 36 * n]


# then concatenate them, in order to sort more conveniently.
# remember the index between 0 to 9 is positive, index between 10 to 19 is negative
training = np.concatenate((training_Pos, training_Neg), axis=0)


# class value has the shape of [10*20], 10 means 10 test imgs while 20 means 20 training imgs
class_value = []


# same order as above
# except for computing the distance (IOU) between test imgs and training imgs
for filename in os.listdir("./Test images (Pos)"):
    single_test = []
    img = plt.imread("./Test images (Pos)" + "/" + filename)
    img = toGray(img)
    grad_hori, grad_vert = gradient_operation(img)
    gradient, direction = generate_magnitude_direction(grad_hori, grad_vert)
    plt.imsave("test_gradient_{}.png".format(filename[:-4]),
               (gradient.astype(np.int16))/np.max(gradient.astype(np.int16)) *255, cmap = 'gray')
```

```
    orientation_gradient = OG(gradient, direction)
    feature_map = feature(orientation_gradient)
    for i in range(training.shape[0]):
        single_test.append(distance(feature_map, training[i]))
    class_value.append(single_test)
    if(filename[:-4] == 'crop001278a' or filename[:-4] == 'crop001500b'):
        fo = open('test_{}_lines.txt'.format(filename[:-4]), "w")
        for i in range(feature_map.shape[0]):
            for j in range(feature_map.shape[1]):
                fo.write(str(feature_map[i, j])+"\n")
        fo.close()


for filename in os.listdir("./Test images (Neg)"):
    single_test = []
    img = plt.imread("./Test images (Neg)" + "/" + filename)
    img = toGray(img)
    grad_hori, grad_vert = gradient_operation(img)
    gradient, direction = generate_magnitude_direction(grad_hori, grad_vert)
    plt.imsave("test_gradient_{}.png".format(filename[:-4]),
            (gradient.astype(np.int16))/np.max(gradient.astype(np.int16)) *255, cmap = 'gray')
    orientation_gradient = OG(gradient, direction)
    feature_map = feature(orientation_gradient)
    for i in range(training.shape[0]):
        single_test.append(distance(feature_map, training[i]))
```

```
    class_value.append(single_test)
    if(filename[:-4] == '00000090a_cut'):
        fo = open('test_{}_lines.txt'.format(filename[:-4]), "w")
        for i in range(feature_map.shape[0]):
            for j in range(feature_map.shape[1]):
                fo.write(str(feature_map[i, j])+"\n")
        fo.close()


# convert to ndarray for sorting
# 3-NN so find the largest 3 results, then print them
# remember the first 5 are positive test imgs, second 5 are negative test imgs
# and the value from 0 to 9 means positive sample, from 10 to 19 means negative sample
for i in range(len(class_value)):
    print(class_value[i])


class_value = np.array(class_value)
class_result = []
for i in range(class_value.shape[0]):
    idx = np.argsort(class_value[i])[-3:]
    class_result.append(idx)


print(class_result)
```