CSC3014 Computer Vision

Lab 6: Pattern Recognition

A. Template Matching

It could be the easiest way for us to search for certain pattern or object from an image. In fact, the idea of this is quite simple. The only difficulty is in preparing a set of templates that consists of the patterns or objects that we are trying to search. For example, if we are trying to search for all handwritten digits in an image, then we must prepare a set of templates that consists of different handwritten digits from 0 to 9. To improve the accuracy, we might need to prepare a large number of templates.

Once we have the templates, then we can proceed to search for the regions that match with the templates. Of course, we will have to do it template-by-template. Because we need to search through the entire image, the process can be slow for large image (in terms of spatial resolution), and if we have large number of templates. Moreover, it does not work well if the regions we try to match have sizes different from the templates we prepared. This can be resolved by matching the templates with the image under different spatial resolutions.

As shown below in the sample code to perform template matching by using cv.matchTemplate() function from OpenCV. At the beginning, we have to first read the image as well as the template. Although it is not necessary to know the size of the template, it would allow us to draw the rectangular box to show the location of the pattern or object that we are looking for. When using the cv.matchTemplate() function, we have to provide the image, the template, and the way to calculate the differences. There are several options for you to choose. But in this example, we will use the normalized sum of square differences.

Once we have the normalized sum of square differences for every possible matching point, we will use the cv.minMaxLoc() function to find the point with smallest differences. This point here serves as the top left corner of the region that matches with the template. Based on this top left corner, we can use the size of the template that we have determined earlier to find the bottom right corner. With these two points, the cv.rectangle() function is used to draw a rectangular box on the image for visualization purpose.

```
1  import cv2 as cv
2  import numpy as np
3  from matplotlib import pyplot as pt
4
5  #Read the image and the template, then convert both of them to grayscale directly.
6  img = cv.imread('mario.png',0)
7  template = cv.imread('marioTemplate.png',0)
8
9  #To get the height (row) and the width (column) of the template.
10 [height,width] = template.shape
11
12 #Perform template matching using normalized sum of square differences at every possible matching point.
13 difResult = cv.matchTemplate(img, template, cv.TM_SQDIFF_NORMED)
14
15 #Find the location with the smallest differences.
16 [minVal, maxVal, minLoc, maxLoc] = cv.minMaxLoc(difResult)
17
```

```
18 #The location with the smallest difference is the top left corner of the region that we are looking for.
19 #The coordinate is arranged in (col, row) format (this matches the arrangement used by cv.rectangle()).
20 topLeftCorner = minLoc
21
22 #Calculate the bottom right corner.
23 bottomRightCorner = (topLeftCorner[0]+width, topLeftCorner[1]+height)
24
25 #Draw a rectangular box based on the top left corner and bottom right corner.
26 cv.rectangle(img,topLeftCorner, bottomRightCorner, 255, 2)
27
28 #Display the results.
29 pt.subplot(121)
30 pt.imshow(difResult, cmap='gray')
31 pt.title('Difference Result')
32 pt.subplot(122)
33 pt.imshow(img, cmap='gray')
34 pt.title('Matching Result')
35 pt.show()
```

B. k Nearest Neighbours (kNN)

The basic idea of kNN is to classify a pattern or object based on the k nearest neighbours. In this case, we will try to recognize handwritten digits 0 to 9 by using kNN. The dataset that we will be using comes from OpenCV. You can download it from eLearn of search for it in your OpenCV folder. It is an image that consists of 5000 handwritten digits with the size of 20x20. In other words, there are 500 samples for each digit, starting from 0 to 9. There are 50 rows, each row contains 100 digits. Before we can use the image, we have to first split the digits. As shown in Figure 1 is how the digits are organized in the image, and how are we going to split the digits.

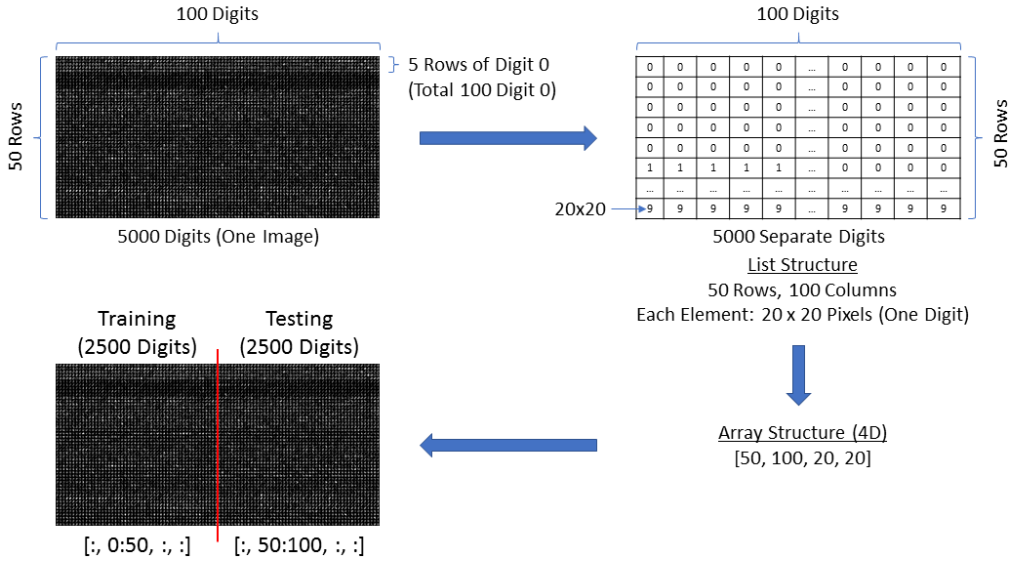

Figure 1: Splitting of 5000 digits into training set and testing set.

First, we will split the image into 50 separate rows, each with 100 elements. In each of the element is a digit with the size of 20x20. We will then convert it to a 4D array (we will not be able to visualize it in

Spyder), and then split it into two. The first half will be used as the training set, and the second half will be used as the testing set. Each set contains 2500 digits.

Note
This is very normal when using machine learning algorithm for pattern or object recognition. You must determine how many samples you should use for training and testing respectively. It is very important to take note that you are not supposed to use the same samples for training and testing.

After we have separated the digits into training set and testing set, we will need to reshape the digits and produce the labels required for training. As shown in Figure 2, each digit is reshaped from 20x20 to 1x400. In this case, we are using all 400 intensity values of a digit as the features for recognition. Take the training set as an example. Because there are 2500 digits in total, this will produce an array with 2500 rows and 400 columns. Each row contains the features of a digit. Next, we will proceed to produce the labels. It is necessary to "teach" the classifier what are the features trying to "describe". If the first row contains the intensity values of digit 0, then we need to assign a label of digit 0 to this row. In this case, we will save the labels into a separate array, and create a copy for the testing set.

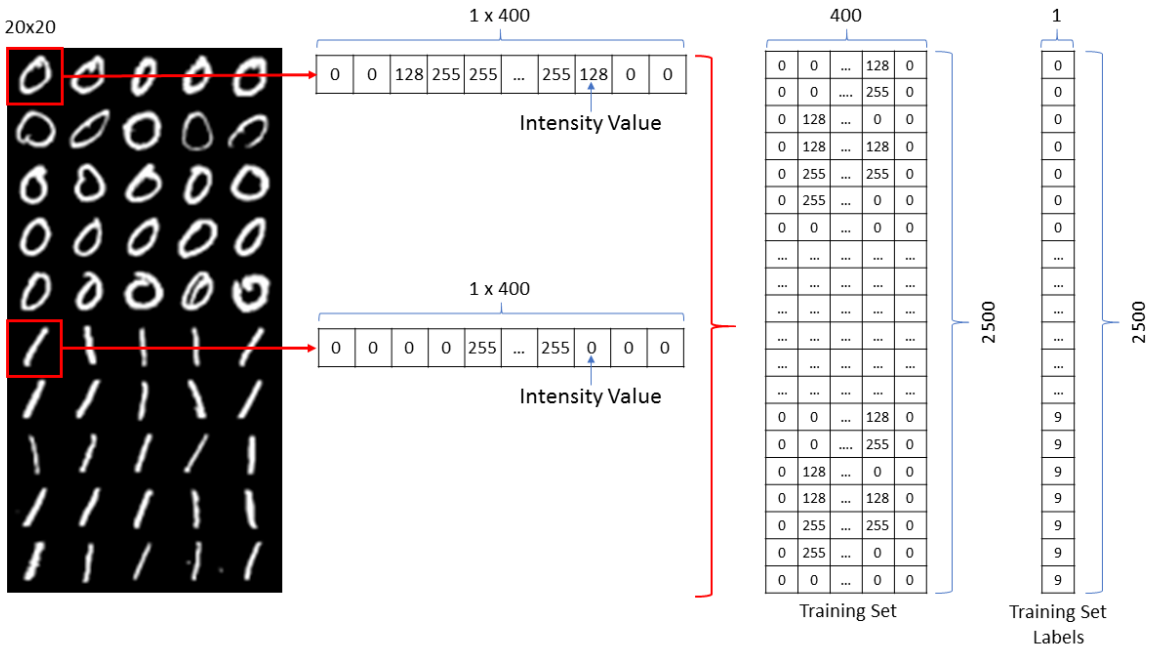

Figure 2: Reshape the digits and generate the labels for each of them.

Note
In fact, the steps of preparing the samples for training and testing are about the same. If you are changing to another type of machine learning algorithm, most probably you will be performing the same steps (reshape and generate labels) to prepare the samples.

As shown below is the sample code that performs what we have gone through above. Please try your best to understand how you should prepare the samples for training.

```
1  import numpy as np
2  import cv2 as cv
3
4  #Read the image and covert it to grayscale.
5  img = cv.imread('digits.png')
6  imgGray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
7
8  # Now we split the image to 5000 digits, each with the size of 20x20.
9  digits = []
10 separatedRows = np.vsplit(imgGray,50)
11 for row in range(0,len(separatedRows)):
12    digits.append(np.hsplit(separatedRows[row],100))
13
14 # Make it into a numpy array. It will be in the size of [50,100,20,20]
15 digits = np.array(digits)
16
17 #Separate the digits into two sets, one for training, one of testing.
18 trainingSet = digits[:,0:50]
19 testingSet = digits[:,50:100]
20
21 #Reshape each digit into the form of 1x400 (flatten).
22 trainingSet = trainingSet.reshape(2500,400).astype(np.float32)
23 testingSet = testingSet.reshape(2500,400).astype(np.float32)
24
25 #To generate the labels needed for the training.
26 trainingLabels = np.zeros((2500,1),dtype=np.int32)
27 rowPointer = 0
28 for digitLabel in range(0,10):
29    for repeatTime in range(0,250):
30       trainingLabels[rowPointer,0] = digitLabel
31       rowPointer = rowPointer + 1
32
33 #Create a copy of the labels for testing.
34 testingLabels = trainingLabels.copy()
35
36 #Create a kNN module and train it with the training data.
37 kNN = cv.ml.KNearest_create()
38 kNN.train(trainingSet, cv.ml.ROW_SAMPLE, trainingLabels)
39
40 #Apply the trained kNN to recognize those digits in the testing set.
41 ret ,result, neighbours, dist = kNN.findNearest(testingSet, k=5)
42
43 #Compare the result with the labels to calculate the accuracy of classification.
44 matches = result==testingLabels
45 correct = np.count_nonzero(matches)
46 accuracy = correct*100.0/result.size
47 print(accuracy)
```

C. Histogram of Oriented Gradients (HOG)

Before we move on to Support Vector Machine (SVM), we still first take a look at the HOG features that we will be using to replace the intensity values. In the previous section, all 400 intensity values of a digit are used as the features for recognition. Although the accuracy of recognition is not bad, it is not always necessary to use so many features for recognition. As shown in Figure 3 is how we will be extracting the HOG features of all the digits, and rearrange the features into an array to train the SVM. Assuming that we have already extracted a digit from the image, we will use the Sobel kernels to calculate the gradient magnitudes and orientations of every single pixel. Because the size of a digit is 20x20, we will have 400 gradient magnitudes and 400 orientations at the end.

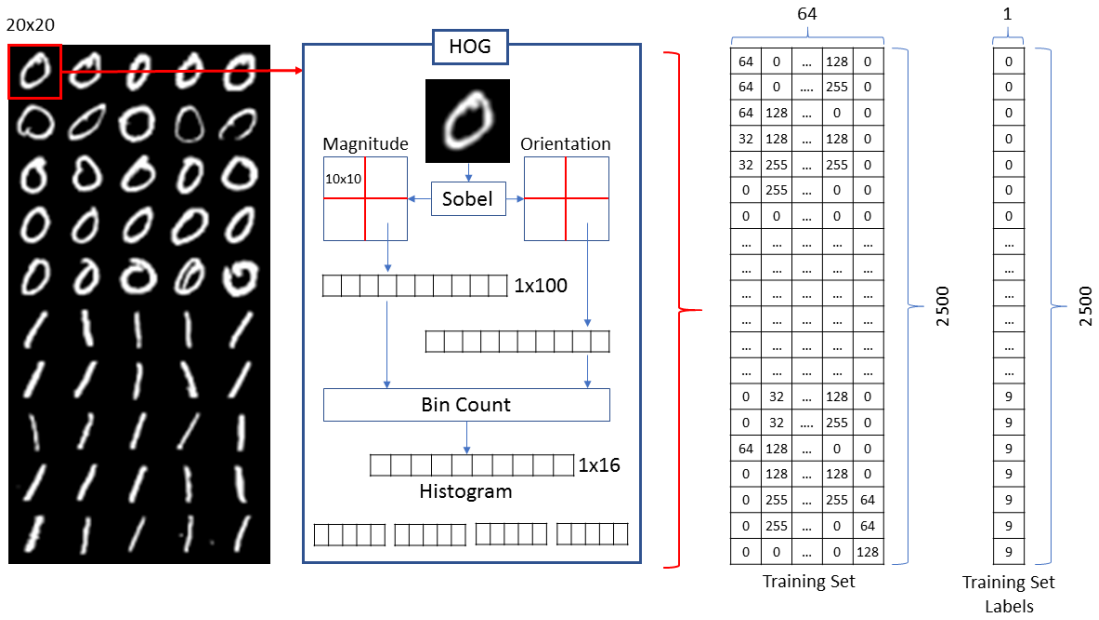

Figure 3: Applying HOG to a digit.

Before we proceed further, we have to first normalize the orientations, so that it will be easier for us to determine the bin where a pixel should be added to. In this case, we divide the orientations into 16 bins as shown in Figure 4, and label them with the value of 0 to 15. Hence, we need to "convert" the orientation (whether in degree or radian) into the value of 0 to 15. This can be achieved by dividing the orientations by 360° or 2π (depends on whether you are working on degree or radian) and then multiply them by 16 (because there are 16 bins).

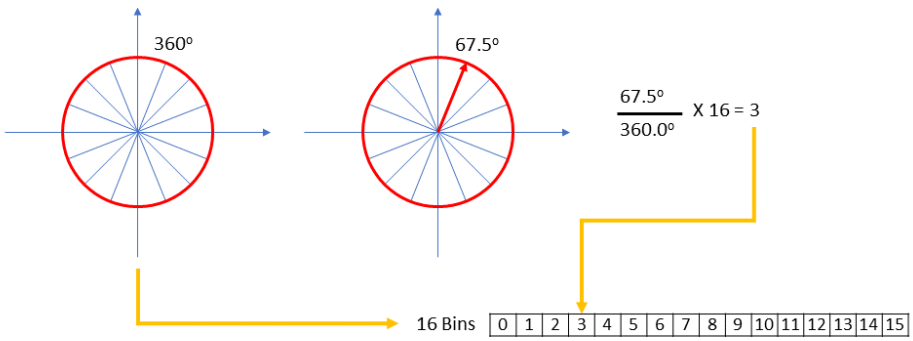

Figure 4: Normalize the gradient orientation.

As you can see from Figure 3, each digit is then separated into four separate quadrants, each with 100 pixels. In other words, we will use the 100 gradient magnitudes and 100 orientations from a quadrant to produce a histogram. But before we can generate the histogram by using the np.bincount() function, we have to first flatten the quadrant into a 1D array. The histogram that we will obtain is in the size of 1x16 because we have 16 bins. There are four quadrants in total, that means we will obtain four histograms at the end, one for each quadrant. The four histograms are then concatenated horizontally to form a 1D array with the size of 1x64. The 64 values inside the concatenated histogram are serving as the features that will be used for training and recognition.

As shown below is the sample code of a function that can be used to generate the HOG features for an image with the size of 20x20. Please save this into a separate Python file and name the file as "hog.py". If you can another file name if you want, but please remember to make the necessary changes to the sample code in the next section.

```
1  def hog(img, binNumber):
2      #Calculate the gradient magnitudes and orientations using Sobel kernels
3      gx = cv.Sobel(img, cv.CV_32F, 1, 0)
4      gy = cv.Sobel(img, cv.CV_32F, 0, 1)
5      mag, ang = cv.cartToPolar(gx, gy)
6
7      #Normalized the orientations based on the number of bins.
8      bins = np.int32((ang/(2*np.pi))*binNumber)
9
10     #Separate a digit into four cells (each cell will have one histogram).
11     allBinCells = bins[:10,:10], bins[10:,:10], bins[:10,10:], bins[10:,10:]
12     allMagCells = mag[:10,:10], mag[10:,:10], mag[:10,10:], mag[10:,10:]
13
14     #Create a list to store all the histograms
15     hists = []
16
17     #Process each cell separately.
18     for cell in range(0,4):
19
20         #Take one cell from orientation and magnitude respectively.
21         oneBinCell = allBinCells[cell]
22         oneMagCell = allMagCells[cell]
23
24         #Flatten the cell from 10x10 to 1x100.
25         oneBinCell = oneBinCell.reshape(100)
26         oneMagCell = oneMagCell.reshape(100)
27
28         #Use the bincount function to generate a histogram based on the predefined number of bins.
29         hists.append(np.bincount(oneBinCell, oneMagCell, binNumber))
30
31     #Concatenate all four histograms horizontally to produce an array of 1x64.
32     hist = np.hstack(hists)
33     return hist
```

D. Support Vector Machine (SVM)

In this case, we will use SVM to again recognize handwritten digits from 0 to 9. We will use back the same image from OpenCV that consists of 5000 handwritten digits from 0 to 9. The steps to train and use SVM for recognizing the handwritten digits are about the same. First, we will follow the same steps illustrated in Figure 1 to split the digits into training set and testing set. Similarly, we will use 2500 samples for training and the remaining 2500 samples for testing.

Next, we will follow the steps illustrated in Figure 3 to produce the HOG features for each digit. As mentioned earlier, each digit will produce a concatenated histogram in the size of 1x64, and the 64 values inside this concatenated histogram are used as the features for training and recognition. Take the training set as an example. Because there are 2500 digits in total, this will produce an array with 2500 rows and 64 columns. Each row contains the features of a digit. But before we proceed to train the SVM, please remember to generate the labels required for training and testing. As shown below it the sample code to recognize handwritten digits using SVM.

```
1  import cv2 as cv
2  import numpy as np
3  from hog import hog
4
5  #Set the number of bins for the histogram.
6  binNumber = 16
7
8  #Read the image and covert it to grayscale.
9  img = cv.imread('digits.png')
10 imgGray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
11
12 # Now we split the image to 5000 digits, each with the size of 20x20.
13 digits = []
14 separatedRows = np.vsplit(imgGray,50)
15 for row in range(0,len(separatedRows)):
16    digits.append(np.hsplit(separatedRows[row],100))
17
18 # Make it into a numpy array. It will be in the size of [50,100,20,20]
19 digits = np.array(digits)
20
21 #Separate the digits into two sets, one for training, one of testing.
22 trainingSet = digits[:,0:50]
23 testingSet = digits[:,50:100]
24
25 #Create a list to store the HOG features of every single digit in the training set.
26 trainingSetHog = []
27
28 #Process every single digit in the training set.
29 for row in range(0,50):
30    for col in range(0,50):
31       #Take one digit from the training set and send it to the HOG function.
32       digitData = trainingSet[row,col]
33       hist = hog(digitData, binNumber)
34       #Append the HOG features of a digit to the list that we have created.
35       trainingSetHog.append(hist)
```

```
36
37  #Make it into a numpy array with the format of float32.
38  trainingSetHog = np.array(trainingSetHog).astype(np.float32)
39
40  #Apply the same steps to the testing set.
41  testingSetHog = []
42  for row in range(0,50):
43      for col in range(0,50):
44          digitData = testingSet[row,col]
45          hist = hog(digitData, binNumber)
46          testingSetHog.append(hist)
47
48  testingSetHog = np.array(testingSetHog).astype(np.float32)
49
50  #To generate the labels needed for the training.
51  trainingLabels = np.zeros((2500,1),dtype=np.int32)
52  rowPointer = 0
53  for digitLabel in range(0,10):
54      for repeatTime in range(0,250):
55          trainingLabels[rowPointer,0] = digitLabel
56          rowPointer = rowPointer + 1
57
58  #Create a copy of the labels for testing.
59  testingLabels = trainingLabels.copy()
60
61  #Create an SVM module.
62  svm = cv.ml.SVM_create()
63  svm.setType(cv.ml.SVM_C_SVC)
64  svm.setKernel(cv.ml.SVM_LINEAR)
65
66  #Train it with the training data.
67  svm.train(trainingSetHog, cv.ml.ROW_SAMPLE, trainingLabels)
68
69  #Apply the trained SVM to recognize those digits in the testing set.
70  result = svm.predict(testingSetHog)[1]
71
72  #Compare the result with the labels to calculate the accuracy of classification.
73  matches = result==testingLabels
74  correct = np.count_nonzero(matches)
75  accuracy = correct*100.0/result.size
76  print(accuracy)
```

Exercise

Change the number of bins in the SVM sample code to see if you can obtain better results. Save those results (number of bins vs. accuracy) into a text file and submit this text file to eLearn.

Exercise

Modify the kNN sample code to use HOG features instead of using intensity values as the features for recognition. Save the accuracy into a text file and submit this text file to eLearn.

References

[1] OpenCV, Template Matching, Link.
[2] OpenCV, OCR of Handwritten Data using kNN, Link.
[3] OpenCV, OCR of Handwritten Data using SVM, Link.