**CS4186 Computer Vision Assignment 1 Report**

**Introduction**

The Development environment used is: Windows10 and Python3.

The methods used in this programming assignment are Convolutional Neural Networks and Colour-Histogram.

To implement CNN, I have used Tensorflow Keras' ResNet50 pre-trained model, not including the fully connected layer at the top.

To implement colour-histogram, I have not used any libraries. However, it should be noted that the Euclidian distance between two numpy arrays will be quicker by a significant amount of time if the numpy.linealg.norm() function is used.

Colour Histogram is the most widely used method for extracting colour features from images. However, as a method for extraction and finding similar images, it is not precise since it is very sensitive to the orientation, the scale, the lighting ,any slight linear transformation and other circumstantially present properties of images. It is simple to understand and implement.

Convolutional Neural Network is able to extract features without any human supervision. It skips the steps of manual calculation. It is very good at extracting complex features and spatial information of images which general feature extractors cannot. CNN is a black-box and not easy to comprehend and debug.

**The Methods and their Implementation**

1. **The Convolutional Neural Network Method**

Step 1:

As stated above, this method uses the Tensorflow Keras ResNet50 model. The VGG16() pre-trained model was tried out as well, but ResNet50 was generally giving better results so it was chosen.

To instantiate the model architecture:

The parameter "include\_top" is set to False as the fully-connected layer at the top is to not be included in the model instance.

Note: In order to speed up the whole process (but use quite a bit of memory), a dictionary called "allFeatures" has been created. This dictionary will hold all the feature numpy arrays once the model.predict() is run.

Step 2:

For each of the 5000 images, the features are extracted. This is done by first calling the extractInstance() function and passing the imgUrl parameter and isCNN=True to it. The extractInstance() function will check if there is a .txt file with the same name in the ./Images folder. If that .txt file is present, it will read the .txt file and obtain the dimensions of the instance. It will then use those dimensions to crop the image, resize it to (244, 244) (this is only done if isCNN is True; the default size of ResNet50 input images, used later-on) and return the instance-image. If the .txt file is absent, it will resize it to (244, 244) and return the image.

Once the instance is extracted, some pre-processing needs to be done for the model's input. The instance is converted into a numpy array and its shape is expanded with axis=0 using image.img\_to\_array()and np.expand\_dims() respectively and stored in img\_data.

The ResNet50 feature is then obtained and squeezed (converted into a 1D array) by running:

The feature is then stored in the dictionary allFeatures with the key equal to the iteration count (which is the same as the image name without the leading zeroes and the extension .jpg)

This is done for all 5000 images, and the 5000 feature numpy arrays are stored in the dictionary allFeatures.

Step 3:

For each of the 20 query images, the steps in Step 2 above are repeated, that is, the instance is extracted, the corresponding numpy array is obtained using img\_to\_array() using axis=0 and the array shape is expanded using expand\_dims(). After that model.predict() is run using the qImg\_data value and squeezed.

Then the obtained query instance feature (called q\_resnet50\_feature), is compared with the 5000 stored images' feature features. Since they are both numpy arrays and the function numpy.linealg.norm()is relatively quick to return the normalized Euclidian distance. This distance is then stored in the "distances" dictionary.

After the above is done for 5000 images, the distances dictionary is sorted based on the distances (values) and the image numbers (keys) are then appended to the allItems list and appended to the file based on the required format.

1. **The Colour Histogram Method**

This method is implemented without using any libraries.

This method uses one helper function, called create\_hist\_bgr()which takes the image or instance as input after extractInstance() is called with an imgURL. Once the numpy array of zeroes corresponding to 16 bins is created, the height, width and channel (unused) is extracted from the image.shape. The bin-size is set to 16 after some trial-and-error and trade-offs between computation-time and correctness. The image is then iterated through row-and-column-wise and the histogram bins are updated.

Step 1:

The above histogram creation is done for the 5000 images (after extracted instance is obtained) first and the returned histogram (numpy array) is stored in the dictionary allHistograms, to be used later for comparison.

Step 2:

For each of the query instances, extract the query instance and create the histogram. Once this is done, compare it with each of the 5000 images by using a distance function (here, user-defined Euclidian distance is used). Store this distance in the dictionary histogram\_distances.

Sort histogram\_distances in ascending order and iterate through it and store the keys in order into histogram\_allItems. Write the content of histogram\_allItems into the rankList file based on the required format.

1. **The Combination Method (SIFT + BRISK)**

In order to accomplish a combination method, the SIFT and BRISK methods were used. Note that the descriptor size of SIFT is 128 bytes and that of BRISK is 64 bytes.

Step 1:

For each of the 5000 images, first the descriptors from SIFT was extracted using its detectAndCompute() function. Then the descriptors from the BRISK method were extracted using its detectAndCompute() function. After that the descriptor arrays were flattened (due to the different shapes of the return values from the two functions) and appended to each other to make one descriptor array. These descriptors are stored in a dictionary with the image number as the key.

Step 2:

For each of the query instances, the appended descriptor is obtained by using the same method as above. After that, the distances are computed and stored in a dictionary.

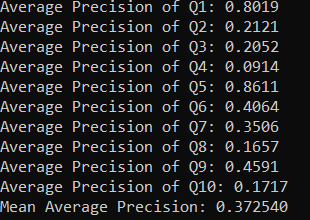
The dictionary is then sorted in the ascending order of the values (distances) and the keys are appended into another list that is used to append it to rankList.

**Results**

Consider the results from the method that gives the highest precision, the CNN method.

For all the methods, if a .txt dimensions file was present, the image was cropped according to it.

Results from metric\_map.py, after running the algorithm on the 10 example queries and comparing it with the ground\_truth given:



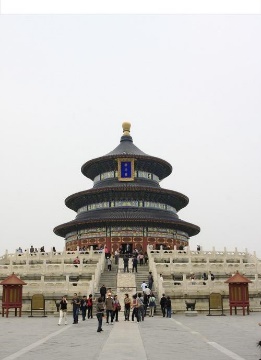
The Mean Average Precision is approximately 37%, which is fairly good when compared with other methods. The advantages mentioned above can be seen in action. It is able to detect spatial and complex features that other methods such as colour histogram and SIFT cannot.

Q1

Ex Query Image 1 and Top 4 matches from rankList (jpg files 1997, 1075, 930 and 253):



Q5

Ex Query Image 5 and Top 4 matches from rankList (jpg files 664, 523, 1348 and 1938):



Q9

Ex Query Image 9 and Top 4 matches from rankList (jpg files 62, 786, 614 and 759):

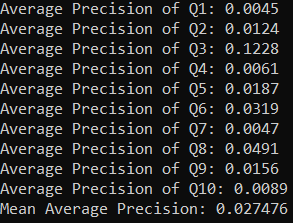


Q6

Ex Query Image 6 and Top 4 matches from rankList (jpg files 1954, 1983, 1442 and 295):



Now consider the other method, the self-implemented Colour-Histogram method:



The Mean Average Precision is approximately 2.7%. This is understandable as the colour histogram method only uses the colours in the image to find similar ones. It is extremely sensitive to circumstantial issues like orientation and lighting.

Q3

Ex Query Image 3 and Top 4 matches from rankList (jpg files 1700, 342, 563 and 1383):

It can be seen that when the lighting and orientation is almost the same, this method is able to get the correct results. However, sometimes the orientation is different and some colours do stand-out and match (here red and blue), tricking the method into giving the wrong results.

