Project 1

CIS 666 Artificial Intelligence

Spring 2020

By

Bhuvana Chandra Atche

Steps towards object detection and computer vision.

Introduction

Using Harris corner detection technique to find the key points of the enclosed image and then retrieving the most important points on the image.Applying K-Means clustering to identify different objects in the image then bounding them with boxes first steps towards computer vision.

Uses libraries

import numpy as np

import cv2

import matplotlib.image as mpimg

from matplotlib import pyplot as plt

from copy import deepcopy

Implementation

Harris corner detection

A corner is a point whose local neighborhood stands in two dominant and different edge directions. In other words, a corner can be interpreted as the junction of two edges, where an edge is a sudden change in image brightness.Corners are the important features in the image, and they are generally termed as interest points which are invariant to translation, rotation and illumination. Although corners are only a small percentage of the image, they contain the most important features in restoring image information, and they can be used to minimize the amount of processed data for motion tracking, image stitching, building 2D mosaics, stereo vision, image representation and other related computer vision areas.

In order to capture the corners from the image, researchers have proposed many different corner detectors including the Kanade-Lucas-Tomasi (KLT) operator and the Harris operator which are most simple, efficient and reliable for use in corner detection. These two popular methodologies are both closely associated with and based on the local structure matrix. Compared to the Kanade-Lucas-Tomasi corner detector, the Harris corner detector provides good repeatability under changing illumination and rotation, and therefore, it is more often used in stereo matching and image database retrieval. Although there still exists drawbacks and limitations, the Harris corner detector is still an important and fundamental technique for many computer vision applications.

The figure above is the original image on which I am going to apply these.

Code for finding the corners of the image

```
# path to input image specified and
# image is loaded with imread command
image = cv2.imread('im-1.jpg')
# convert the input image into
# grayscale color space
operatedImage = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# modify the data type
# setting to 32-bit floating point
operatedImage = np.float32(operatedImage)

# apply the cv2.cornerHarris method
# to detect the corners with appropriate
# values as input parameters
dest = cv2.cornerHarris(operatedImage, 2, 5, 0.07)

# Results are marked through the dilated corners
dest = cv2.dilate(dest, None)

# Reverting back to the original image,
# with optimal threshold value
image[dest > 0.01 * dest.max()]=[0, 0, 255]
# the window showing output image with corners
cv2.imshow('Image with Borders', image)
```

Image need to be converted into gray scale as it is a 2-D array structure the inbuilt function of the harris corner operate on this 2-D data where as RGB values are three dimensional.

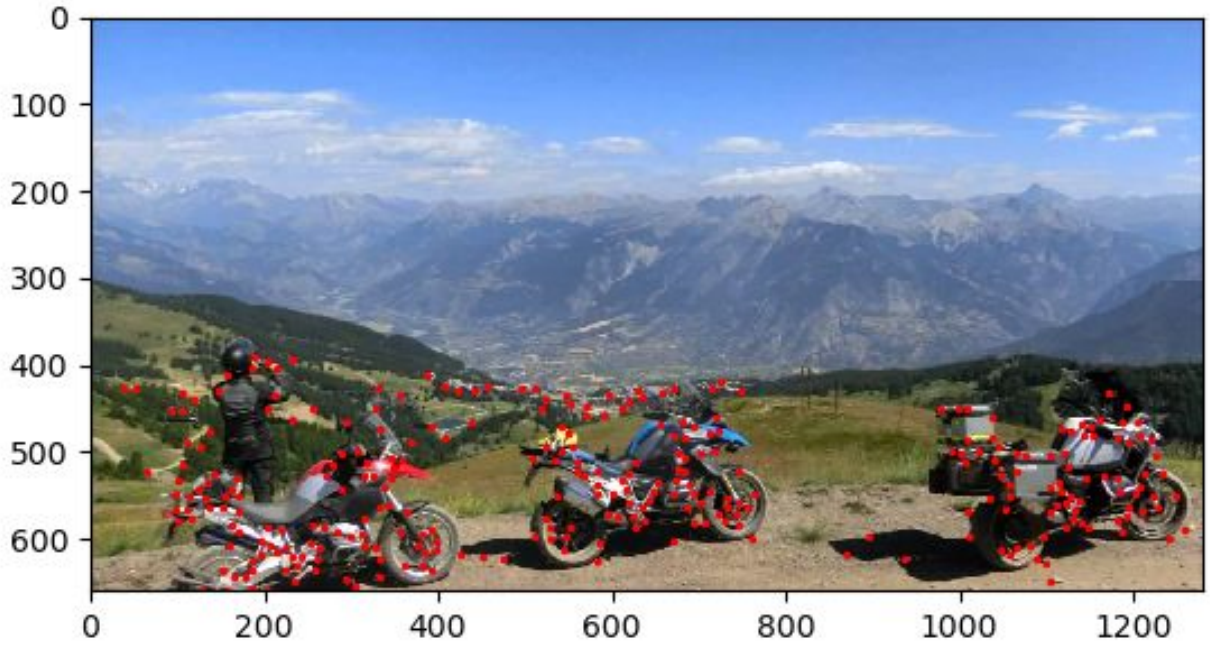
The above code snippet finds out all the possible corners on the image we will be able to see that some part of the image is far away and the color complexion is negligible because of the similar gray scale values one of the draw back of harris corner lies here.

Strongest 300 Points

```
#finding top 300 keypoint among the borders selected
corners = cv2.goodFeaturesToTrack(operatedImage,300,0.01,10)
corners = np.int0(corners)
x=list()
y=list()
for i in corners:
        a,b = i.ravel()
        x.append(a)
        y.append(b)

fig, ax = plt.subplots()
ax.imshow(im)
ax.plot(x,y, '.r', markersize =3)
ax.axis((0,1280,660,0))
plt.show()
```

Finding the strongest 300 points and then using the list of points to mark on the matplot lib



The above image gives the top 300 points on the image.

K-Means Clustering

K Means algorithm is an unsupervised learning algorithm, ie. it needs no training data, it performs the computation on the actual dataset. This should be apparent from the fact that in K Means, we are just trying to group similar data points into clusters, there is no prediction involved.

One can apply the 1-nearest neighbor classifier on the cluster centers obtained by k-means to classify new data into the existing clusters.

How k-Means Clustering works:
The K Means algorithm is iterative based, it repeatedly calculates the cluster centroids, refining the values until they do not change much. The k-means algorithm takes a dataset of ‘n ’ points as input, together with an integer parameter ‘k ’ specifying how many clusters to create(supplied by the programmer). The output is a set of ‘k ’ cluster centroids and a labeling of the dataset that maps each of the data points to a unique cluster.
The math:

$$\sum_{k=1}^{K} \sum_{\mathrm{x}_n \in C_k} ||\mathrm{x}_n - \mu_k||^2$$

In the beginning, the algorithm chooses k centroids in the dataset. Then it calculates the distance of each point to each centroid. Each centroid represents a cluster and the points closest to the centroid are assigned to the cluster. At the end of the first iteration, the centroid values are recalculated, usually taking the arithmetic mean of all points in the cluster.
After the new values of centroid are found, the algorithm performs the same set of steps over and over again until the differences between old centroids and the new centroids are negligible. Implementing k-Means:
The implementation can be divided into the following:
1. Handle Data: Clean the file, normalize the parameters, given numeric values to non-numeric attributes. Read data from the file and split the data for cross validation.
2. Find Initial Centroids: Choose k centroids in random.
3. Distance Calculation: Finding the distance between each of the datapoints with each of the centroids. This distance metric is used to find the which cluster the points belong to.
4. Re-calculating the centroids: Find the new values for centroid.
5. Stop the iteration: Stop the algorithm when the difference between the old and the new centroids is negligible.

```
# Euclidean Distance Calculator
def dist(a, b, ax=1):
        return np.linalg.norm(a - b, axis=ax)
```

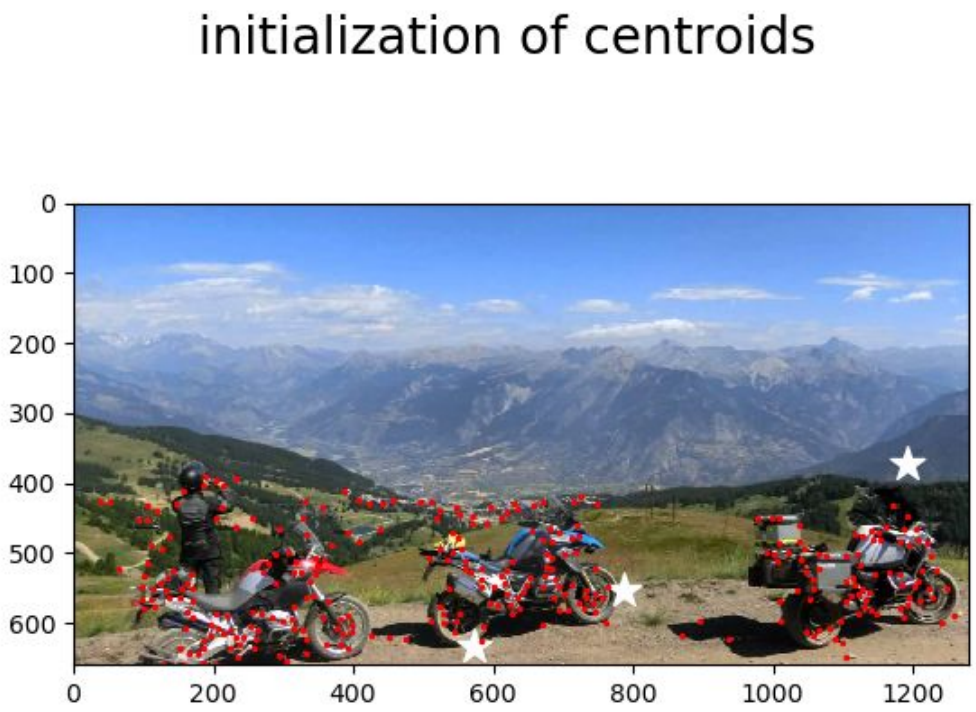
Euclidean distance calculator for the points.

I used cluster = 3 to show the output and the comparison is explained below pages.

```
# Number of clusters
k = 3
# X coordinates of random centroids
C_x = np.random.randint(100, 1980, size=k)
# Y coordinates of random centroids
C_y = np.random.randint(300, 660, size=k)
C = np.array(list(zip(C_x, C_y)), dtype=np.float32)
F = np.array(list(zip(x,y)))

# Plotting along with the Centroids
fig , ax = plt.subplots()
```

```
ax.imshow(im)
ax.axis((0,1280,660,0))
ax.imshow(im)
fig.suptitle('initialization of centroids',fontsize = 20)
ax.plot(x,y, '.r', markersize =3)
plt.scatter(C_x, C_y, marker='*', s=200, c='white')
plt.show()
```



The centroids are randomly initialised and marked with white stars for convenience.

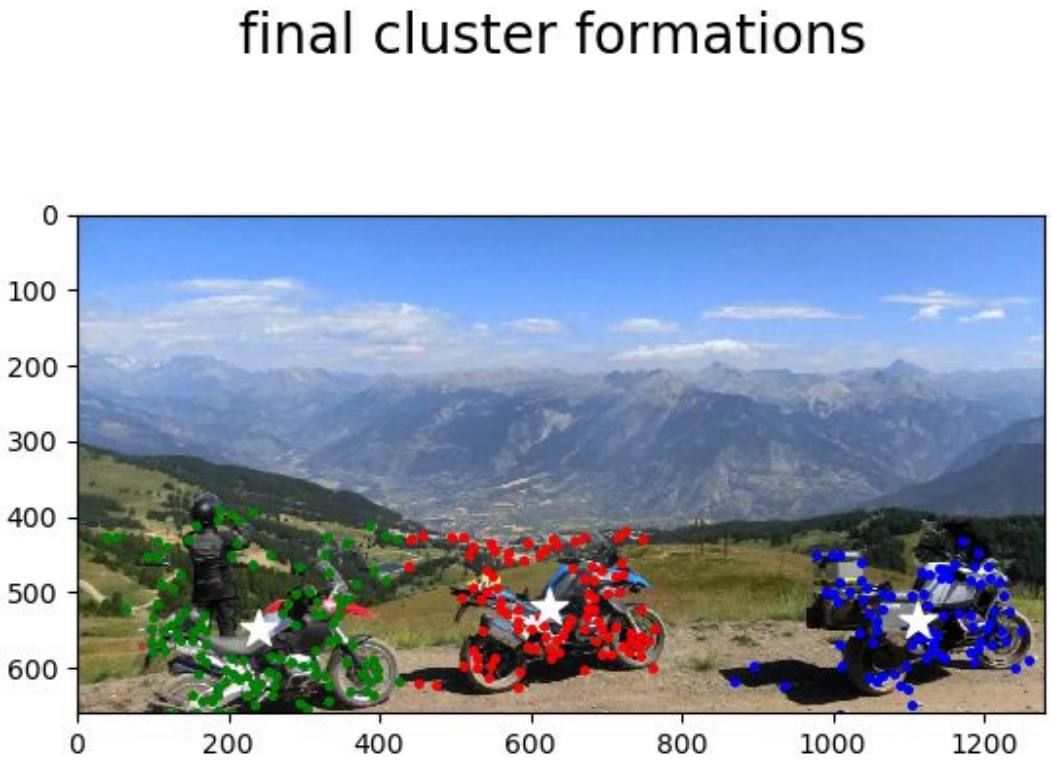
Now comes the integral part of this project which is implementation of the K-Means.

K-Means

```
# To store the value of centroids when it updates
C_old = np.zeros(C.shape)
# Cluster Lables(0, 1, 2)
clusters = np.zeros(len(F))
# Error func. - Distance between new centroids and old centroids
error = dist(C, C_old, None)
# Loop will run till the error becomes zero
while error != 0:
        # Assigning each value to its closest cluster
        for i in range(len(F)):
        distances = dist(F[i], C)
        cluster = np.argmin(distances)
```

```
        clusters[i] = cluster
        # Storing the old centroid values
        C_old = deepcopy(C)
        # Finding the new centroids by taking the average value
        for i in range(k):
        points = [F[j] for j in range(len(F)) if clusters[j] == i]
        C[i] = np.mean(points, axis=0)
        error = dist(C, C_old, None)
colors = ['r', 'g', 'b', 'y', 'c', 'm']
fig , ax = plt.subplots()
ax.axis((0,1280,660,0))
ax.imshow(im)
for i in range(k):
        points = np.array([F[j] for j in range(len(F)) if clusters[j] == i])
        ax.scatter(points[:, 0], points[:, 1], s=7, c=colors[i])
ax.scatter(C[:, 0], C[:, 1], marker='*', s=200, c='white')
fig.suptitle(' final cluster formations',fontsize = 20)
plt.show()
```

Using the algorithms above the implementation is done.



No. of cluster completely varies and there is no number which is best for all sizes.
When used small K value the identification is not done properly and includes farthest points which is completely opposing the property :

Interior distances of the cluster must be minimised.

And when large K value is used it decreases the error but within a specified place to many clusters are formed which leaves out the property :

Cluster to cluster distance should be maximised.

So, K depends on the data and goal to be reached.

Bounding Boxes

```
def bounding_box(points):
        """returns a list containing the bottom left and the top right
        points in the sequence
        Here, we use min and max four times over the collection of points
        """
        bot_left_x = min(point[0] for point in points)
        bot_left_y = min(point[1] for point in points)
        top_right_x = max(point[0] for point in points)
        top_right_y = max(point[1] for point in points)
        return [(bot_left_x, bot_left_y), (top_right_x, top_right_y)]

fig , ax = plt.subplots()
ax.axis((0,1280,660,0))
colors = [(240,248,255), (118,238,198), (0,255,255),(0,255,0),(255,0,0)]
p = list()
for i in range(k):
        points = np.array([F[j] for j in range(len(F)) if clusters[j] == i])
        p = bounding_box(points)
        c=colors[i]
        im = cv2.rectangle(im,p[0],p[1],c,2)
ax.imshow(im)
ax.scatter(C[:, 0], C[:, 1], marker='*', s=200, c='white')
fig.suptitle('Bounding box',fontsize = 20)
plt.show()
```

From each cluster the bottom left point and the top right points are found out which are used to bound the clusters with rectangles.

Bounding box