Assignment #6

HARNEET SINGH - 400110275

COMPENG 4TN4: Image Processing

March 26, 2022

Theory

1. Hough Transform

For the given image, origin is assumed at the highlighted cell as shown below:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 255 & 255 & 255 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 255 & 0 & 255 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 255 & 255 & 255 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \boxed{255} & 0 & 0 & 255 & 0 \\ 0 & 0 & 0 & 0 & 0 & 255 & 0 & 0 & 255 & 0 \\ 0 & 0 & 0 & 0 & 0 & 255 & 0 & 0 & 255 & 0 \\ 0 & 0 & 0 & 0 & 0 & 255 & 255 & 255 & 255 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The total length of ρ is calculated by using the Pythagoras theorem i.e. diagonal $= \sqrt{rows^2 + columns^2}$. Based on the Hough transform algorithm, 180 (arbitrary number) θ and ρ values are stored in a vector.

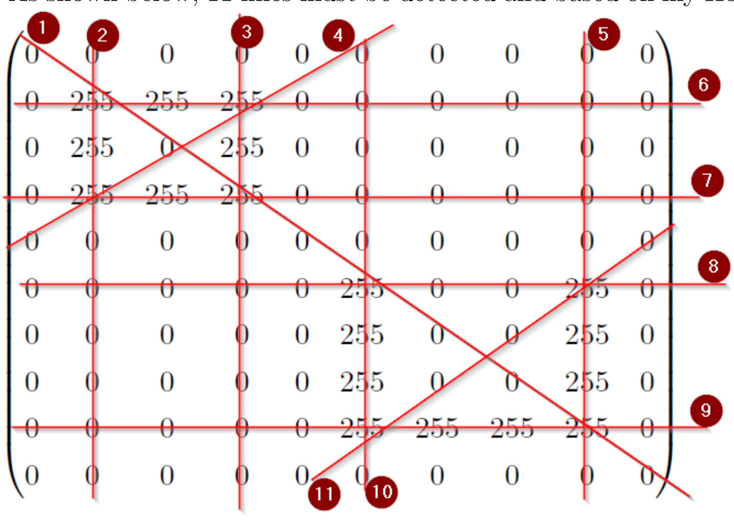
Initially, accumulator is set to 0 and for each non-zero pixel values i.e. 18 values of 255, the ρ value is calculated as:

$$\rho = x.cos\theta - y.sin\theta$$

for each θ in the accumulator. Then, calculated ρ value is rounded down to the available ρ value in the accumulator matrix and the, accumulator cell is incremented for that specific ρ and θ value.

In the end, the accuracy of the Hough Transform is dependent on the resolution in accumulator matrix. In my accumulator matrix, the max accumulator value is 4.

As shown below, 11 lines must be detected and based on my Hough transform algorithm, I receive 11 ρ and θ pairs.



Following Python code uses the same technique to compute Hough transform on all the non-zero pixels in the given image:
NOTE: Zero-padding is not applied to the given image.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# read given images
img = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 255, 255, 255, 0, 0, 0, 0, 0, 0],
                [0, 255, 0, 255, 0, 0, 0, 0, 0, 0],
                [0, 255, 255, 255, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 255, 0, 0, 255, 0],
                [0, 0, 0, 0, 0, 255, 0, 0, 255, 0],
                [0, 0, 0, 0, 0, 255, 0, 0, 255, 0],
                [0, 0, 0, 0, 0, 255, 255, 255, 255, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
img_height, img_width = img.shape[:2]

# diagonal distance between the image (max rho value)
diagonal = np.sqrt(np.square(img_height) + np.square(img_width))

# design accumulator based on number of theta (180) and number of rhos (180) we need
tick_rho = 2 * diagonal/180
tick_theta = 1

rhos = np.arange(-diagonal, diagonal, tick_rho)
thetas = np.arange(0, 180, tick_theta)
# print(rhos)
# print(thetas)

cosTheta = np.cos(np.deg2rad(thetas))
sinTheta = np.sin(np.deg2rad(thetas))

# defining an accumulator of size 180x180
accumulator = np.zeros((len(rhos), len(thetas)))
# print(accumulator.shape)

for i in range(img_height):
    for j in range(img_width):
        if img[i, j] != 0:
            # image origin is in the middle
            # print(f'image index is {i - img_height / 2}, {j - img_width / 2}')

            # need lists to store rho and theta values
            rhoList, thetaList = [], []

            for theta in thetas:
                rho = ((j - img_width / 2) * (cosTheta[theta])) + ((i - img_height / 2) * (sinTheta[theta]))
                # print(rho, theta)
                rhoList.append(rho)
                thetaList.append(theta)

                # now we need to find the index of row where we need to increment the accumulator

                rhoIndex = np.argmin(np.abs(rhos - rho))
                # print(f'{rhoIndex}')

                accumulator[rhoIndex, theta] += 1
```

```python

accumulatorMax = np.amax(accumulator)
print(f'max value in accumulator is: {accumulatorMax}')

# now we need to iterate through the accumulator matrix to find the max value
numMaxValues = 0
for k in range(len(rhos)):
    for l in range(len(thetas)):
        if accumulator[k, l] == accumulatorMax:
            rho = rhos[k]
            theta = thetas[l]
            print(f'{k}, {l}, rhos is {rho}, and theta is {theta}')
            numMaxValues += 1

print(f'number of max values in the accumulator are {numMaxValues}')
```

<u>Result:</u>

Line Number	ρ	θ
1	-3	179
2	0	0
3	0	1
4	0	135
5	0	179
6	1	135
7	3	0
8	3	1
9	3	89
10	3	90
11	3	91

2. Optical Flow

Zero-padding is applied to the given image to get accurate results at the edges.
Applying kernel of length 2 to get the gradient values on the given image. For I_x considering right-pixel to be the anchor pixel.

$$I_x = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 63 & 63 & 0 & -63 & -63 \\ 0 & 127 & 63 & 0 & -63 & -127 \\ 0 & 63 & 0 & 0 & 0 & -63 \end{bmatrix}$$

For I_y considering bottom-pixel to be the anchor pixel.

$$I_y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 63 & 191 & 255 & 191 & 63 \\ 0 & 0 & -63 & -127 & -63 & 0 \\ 0 & -63 & -127 & -127 & -127 & -63 \end{bmatrix}$$

For I_t considering 'next frame' to be the anchor pixel.

$$I_t = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -63 & -63 & 0 & 63 & 63 \\ 0 & -127 & -63 & 0 & 63 & 127 \\ 0 & -63 & 0 & 0 & 0 & 63 \end{bmatrix}$$

Applying the flow equation and constraint equation, we can find $\vec{u} = [u\,v]'$ because we get a over-constrained linear equation, 1 equation for each pixel value:
A.d = b, where A is 9x1, d is 2x1 and b is 9x1 because the spatial coherency is 3x3 window i.e. 9 pixels in the window. Using the given equation: $A'A = \sum \begin{bmatrix} I_x^2 & I_x.I_y \\ I_x.I_y & I_y^2 \end{bmatrix}$ for each pixel using the values.

$$A'T = -\sum \begin{bmatrix} I_x.I_t \\ I_y.I_t \end{bmatrix}$$

Result: Vertical is rows and horizontal are columns. First value in the cell shows the horizontal velocity and second value is the vertical value. Zero-padding is applied on the given frames:

-	1	2	3	4	5	6
1	0.5, 0.5	1, 0	1, 0	1, 0	1, 0	1, 0
2	1, 0	1, 0	1, 0	1, 0	1, 0	1, 0
3	1, 0	1, 0	1, 0	1, 0	1, 0	1, 0
4	1, 0	1, 0	1, 0	1, 0	1, 0	1, 0

NOTE: values have been rounded down, please look at the supplied code for exact values.

As we can see, pixel at location (1, 1) has components in both direction but all the other ones are showing flow in the horizontal direction. This result is consistent with the zero-padding around the edges.
Following Python code uses the same technique to compute optical flow for all the pixels in the given image using 3x3 window:
NOTE: Zero-padding is applied to the given image.

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

# read given images
img0 = np.array([[0, 0, 0, 0, 0, 0],
                 [0, 255, 255, 255, 0, 0],
                 [0, 255, 0, 255, 0, 0],
                 [0, 0, 0, 0, 0, 0]])

img1 = np.array([[0, 0, 0, 0, 0, 0],
                 [0, 0, 255, 255, 255, 0],
                 [0, 0, 255, 0, 255, 0],
                 [0, 0, 0, 0, 0, 0]])
imgHeight, imgWidth = img0.shape[:2]
```

```python

# zero padding
frame0 = np.zeros((imgHeight+2, imgWidth+2), dtype='uint8')
frame1 = np.zeros((imgHeight+2, imgWidth+2), dtype='uint8')
frame0[1:5, 1:7] = img0
frame1[1:5, 1:7] = img1

Grad_Ix = np.zeros_like(frame0, dtype='int')
Grad_Iy = np.zeros_like(frame0, dtype='int')
Grad_It = np.zeros_like(frame0, dtype='int')

# find partial derivative in x-direction
for h in range(1, imgHeight+1):
    for k in range(1, imgWidth+1):
        # I_x = (1/4) * (frame1[h, k] + frame1[h+1, k] + frame0[h, k] + frame0[h+1, k])
        # I_xPlus1 = (1/4) * (frame1[frame1[h, k+1] + frame1[h+1, k+1] + frame0[h, k+1] + frame0[h+1, k+1]])
        x_frame0 = np.sum(frame0[h:h+2, k:k+1])
        x_frame1 = np.sum(frame1[h:h+2, k:k+1])

        xPlus1_frame0 = np.sum(frame0[h:h+2, k+1:k+2])
        xPlus1_frame1 = np.sum(frame1[h:h+2, k+1:k+2])

        xPlus1 = (xPlus1_frame0 + xPlus1_frame1) * (1/4)
        xPlus0 = (x_frame0 + x_frame1) * (1/4)

        Grad_Ix[h+1, k+1] = xPlus1 - xPlus0


# find partial derivative in y-direction
for h in range(1, imgHeight+1):
    for k in range(1, imgWidth+1):
        y_frame0 = np.sum(frame0[h:h+1, k:k+2])
        y_frame1 = np.sum(frame1[h:h+1, k:k+2])

        yPlus1_frame0 = np.sum(frame0[h+1:h+2, k:k+2])
        yPlus1_frame1 = np.sum(frame1[h+1:h+2, k:k+2])

        yPlus1 = (yPlus1_frame0 + yPlus1_frame1) * (1/4)
        yPlus0 = (y_frame0 + y_frame1) * (1/4)

        Grad_Iy[h+1, k+1] = yPlus1 - yPlus0

# find partial derivative in t-direction
for h in range(1, imgHeight+1):
    for k in range(1, imgWidth+1):
        t_frame0 = np.sum(frame0[h:h+2, k:k+2], dtype='int')
        t_frame1 = np.sum(frame1[h:h+2, k:k+2], dtype='int')

        t = (t_frame1 - t_frame0) * (1/4)

        Grad_It[h+1, k+1] = t

Ix = Grad_Ix[1:5, 1:7]
Iy = Grad_Iy[1:5, 1:7]
It = Grad_It[1:5, 1:7]

print(f'Ix is: \n {Ix}')
print(f'Iy is: \n {Iy}')
print(f'It is: \n {It}')

# Ix_square = np.square(Ix)
# Iy_square = np.square(Iy)
```

```python
# I_xy = np.multiply(Ix, Iy)
# I_xt = np.multiply(Ix, It)
# I_yt = np.multiply(Iy, It)

# A_transpose_A = np.array((2, 2), dtype='int')
# A_transpose_b = np.array((2, 1), dtype='int')
# # applying least square method to find u-vector for each pixel
# for h in range(0, imgHeight):
#     for k in range(0, imgWidth):
#         A_transpose_A = [[Ix_square[h, k],  I_xy[h, k]],
#                          [I_xy[h, k],       Iy_square[h, k]]]
#
#         A_transpose_b = [[-I_xt],
#                          [-I_yt]]
#
#         u = np.linalg.lstsq(A_transpose_A, A_transpose_b, rcond=None)

A = np.zeros((9, 2), dtype='int')
b = np.zeros((9, 1), dtype='int')

index = 0
# applying least square method to find u-vector for each pixel
for h in range(0, imgHeight):
    for k in range(0, imgWidth):
        i_iterator = 0

        subMatrixIx = Grad_Ix[h:h+3, k:k+3]
        subMatrixIy = Grad_Iy[h:h+3, k:k+3]
        subMatrixIt = Grad_It[h:h+3, k:k+3]

        for i in range(3):
            for j in range(3):
                A[i_iterator] = [subMatrixIx[i, j], subMatrixIy[i, j]]
                b[i_iterator] = -subMatrixIt[i, j]
                i_iterator += 1

        u = np.linalg.lstsq(A, b, rcond=None)[0]

        print(f'At index: {index} ({h}, {k}): u is:\n{u}', end='\n\n')
        index += 1
```

3. Orientation Detection

We can perform Hough transform to get the distance of the titled box and the angles at which they are titled.

Input Image → Binary → Hough Transform → H.T. Peaks → Pick 2 lines and subtract angle

- First, load the image to perform pre-processing steps.
- Next, apply a pre-processing step to convert the image to the binary scale i.e. invert the image.
- After binarizing the image, apply Hough transform to get (ρ, θ) values for different lines in the image.
- Next, using openCV's *hough_line_peaks* method(), find the peak (ρ, θ) corresponding to lines in the image.
- With the peak values, we can explore the theta values for each line and compare the theta values. By randomly selecting, unique two θ values, we can retrieve the difference in their alignment i.e. it will give us the orientation of the object, because θ is the normal vector on the lines. Subtract the two θ values to get the orientation of the object based on the lines.
- If selected properly, we can get the orientation correct in the first attempt. However, if the answer turns out to be zero, we need to replace θ with the other values from the peaks.

Implementation

1. Tracking

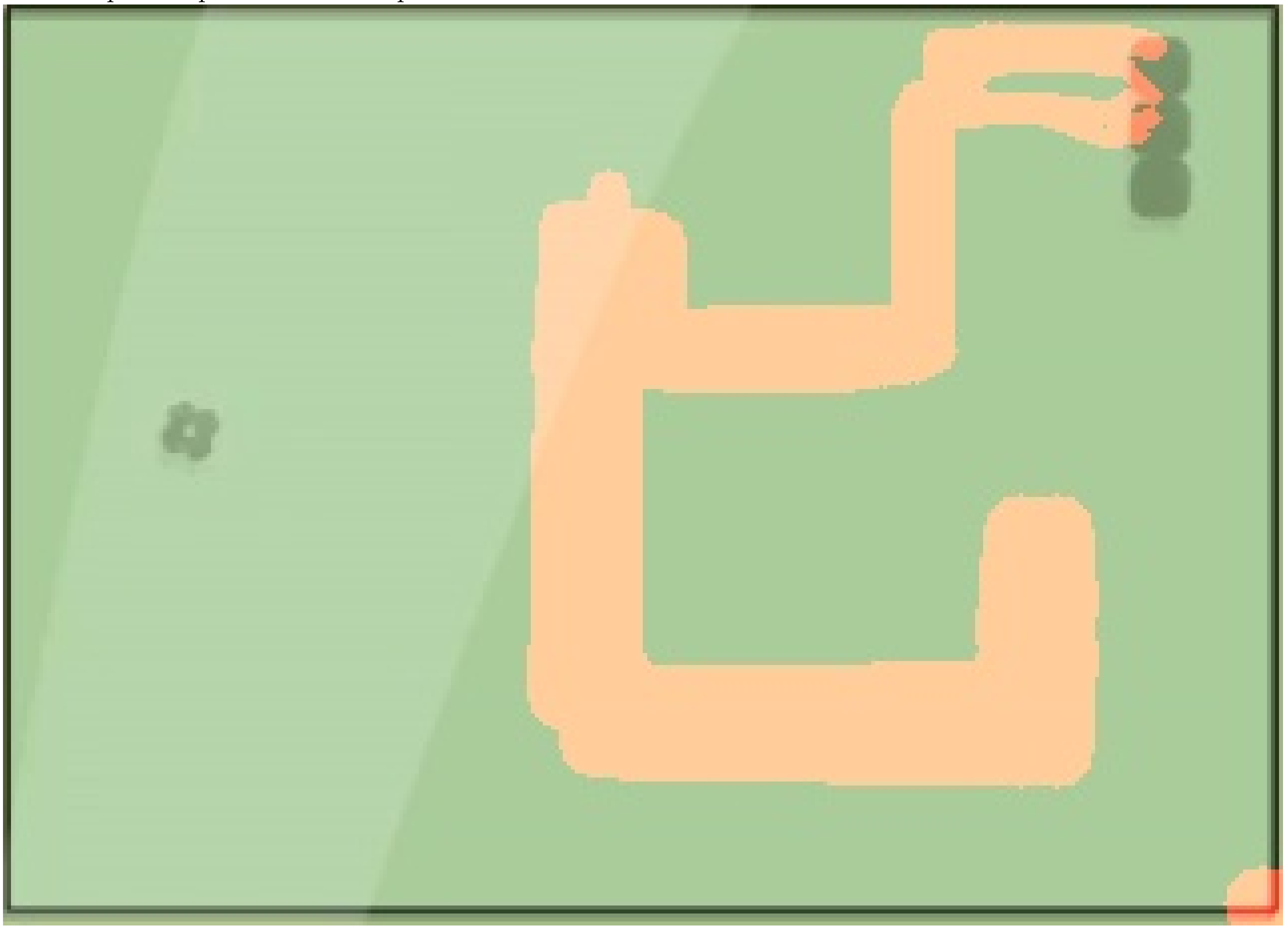
Python Script:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# read given video
cap = cv2.VideoCapture('video1.mp4')
_, frame = cap.read()
oldGrayFrame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
frameHeight, frameWidth = oldGrayFrame.shape

numberFrames = 0

# params for ShiTomasi corner detection
feature_params = dict(maxCorners=50,
                      qualityLevel=0.3,
                      minDistance=7,
                      blockSize=7)

# Parameters for lucas kanade optical flow
lk_params = dict(winSize=(25, 25),
                 maxLevel=2,
                 criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.03))

mask = np.zeros_like(oldGrayFrame)
cv2.rectangle(mask, (frameWidth // 2, frameHeight // 2), (frameWidth, frameHeight), 255, -1)
# cv2.imshow("mask", mask)
# cv2.waitKey(0)

oldPts = cv2.goodFeaturesToTrack(oldGrayFrame, mask=mask, **feature_params)
mask = np.zeros_like(frame)
# cv2.imshow("newMask", mask)
# cv2.waitKey(0)

thickness = 35

while True:
    ret, frame = cap.read()
    if not ret:
        print(f'Reached end of frames')
        break

    print(f'{numberFrames=}')

    frameGray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # optical flow
```

```python
    newPts, status, error = cv2.calcOpticalFlowPyrLK(oldGrayFrame, frameGray, oldPts, None, **lk_params)

    # Select good points
    good_new = newPts[status == 1]
    good_old = oldPts[status == 1]

    thickness -= 0.95
    # draw the tracks
    for i, (new, old) in enumerate(zip(good_new, good_old)):
        a, b = new.ravel().astype(np.int32)
        c, d = old.ravel().astype(np.int32)
        print(f'{i=}, {a=}, {b=}, {c=}, {d=}')
        mask = cv2.line(mask, (a, b), (c, d), (0, 0, 255), int(thickness*0.5))
        # mask = cv2.circle(mask, (a, b), int((thickness*0.5)), (0, 0, 255), -1)
        # frame = cv2.circle(frame, (a, b), 5, (255, 0, 0), -1)
    img = cv2.add(frame, mask)

    # cv2.line(frame, (oldPts[0][0].astype(np.int32)), (newPts[0][0].astype(np.int32)), (0, 0, 255), thickness + 1)

    cv2.imshow("frame", img)

    if numberFrames == 33:
        cv2.imwrite("Last_Frame.jpg", img)
        print("Image Created")

    key = cv2.waitKey(1)
    if key == 27:
        break

    numberFrames += 1
    oldGrayFrame = frameGray.copy()
    oldPts = good_new.reshape(-1, 1, 2)
    # print(f'{oldPts=}')

print(f'{numberFrames= }')
cap.release()
cv2.destroyAllWindows()
```

Result:

Following image shows track of the snake by each frame, thickest point represents starting point of the snake and the thinnest point represents the last point.



2. MNIST Digit Recognition

Python Script:

```
# -*- coding: utf-8 -*-
"""4TN4_Ass6_Impl2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1SbmVCdUtKqRXYHfFd-SNo8pnT8xNMDjp
"""

!pip install idx2numpy

import tensorflow as tf
import cv2
import numpy as np
from matplotlib import pyplot as plt
import idx2numpy
import random
import pandas as pd
from sklearn import svm, metrics

"""Load dataset and convert it to numpy array."""

imgsFilePath = "train-images.idx3-ubyte"
labelsFilePath = "train-labels.idx1-ubyte"

images = idx2numpy.convert_from_file(imgsFilePath)
labels = idx2numpy.convert_from_file(labelsFilePath)
# print(images)
# print(images.shape)
# print(labels)
# print(labels.shape)

# keeping only the images with labels 0, 1 and 2:
images = images[labels < 3]
labels = labels[labels < 3]
# print(images)
# print(labels)
print(images.shape)
print(labels.shape)

"""Display random 10 images"""

# print(images[0])
# print(labels[0])
# range(len(labels))

def randomTenImages(images, labels):
  for j in range(0, 10):
      i = random.randint(0, len(labels))

      plt.imshow(images[i], cmap='gray')
      plt.title('label: ' + str(labels[i]))
      plt.show()

randomTenImages(images, labels)

"""Split MNIST data into training, validation and test"""

```

```
numLabels = labels.shape[0]

trainSetSize = int(0.7 * numLabels)
ValidationSetSize = int(0.2 * numLabels)
testSetSize = int(0.1 * numLabels)
# print(trainSetSize)
# print(ValidationSetSize)
# print(testSetSize)

trainImages = images[:trainSetSize, :]
trainLabels = labels[:trainSetSize]
validationImages = images[trainSetSize:trainSetSize+ValidationSetSize, :]
validationLabels = labels[trainSetSize:trainSetSize+ValidationSetSize]
testImages = images[trainSetSize+ValidationSetSize:, :]
testLabels = labels[trainSetSize+ValidationSetSize:]
print(trainImages.shape)
print(trainLabels.shape)
# print(validationImages.shape)
# print(validationLabels.shape)
# print(testImages.shape)
# print(testLabels.shape)

# testing if data is split correctly (tried for all 3 splits)
randomTenImages(testImages, testLabels)

"""## 2.1 Preprocessing: Low pass filter to remove noise and then binarize images"""

def preProcessingImgs(Images):
  for i in range(len(Images)):
    blurImg = cv2.GaussianBlur(Images[i], (3,3), 0)

    # binarize as well
    _, Images[i] = cv2.threshold(blurImg, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

  return Images

trainImages = preProcessingImgs(trainImages)
# randomTenImages(trainImages, trainLabels)

"""## 2.2 Feature Extraction: Extract Contours"""

def featureExtraction(images, labels):
  featArray = np.empty((0, 8))

  for i in range(len(images)):
    contours, _ = cv2.findContours(images[i], cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    cnt = contours[0]

    _, _, width, height = cv2.boundingRect(cnt)

    # calculate moments for the contour
    M = cv2.moments(cnt)

    if M["m00"] != 0:
      cX = int(M["m10"] / M["m00"])
      cY = int(M["m01"] / M["m00"])
    else:
      cX, cY = 0, 0


    X_1 = cv2.countNonZero(images[i])
    X_2 = float(width / height)
```

```
    X_3 = cv2.contourArea(cnt)
    X_4 = cv2.arcLength(cnt, True)
    X_5 = cX
    X_6 = cY
    X_7 = width

    featArray = np.append(featArray, np.array([[X_1, X_2, X_3, X_4, X_5, X_6, X_7, int(labels[i])]]), axis=0)

  return featArray

featArray = featureExtraction(trainImages, trainLabels)
# my list has 8 elements because I am considering both moments i.e. along x and y direction,
# and also label is added for further processing
print(featArray.shape)
print(featArray[500])

"""Display mean and variance"""

print(len(featArray))
# print(featArray[400][-1])

featArray_digit0 = np.empty((0, 8))
featArray_digit1 = np.empty((0, 8))
featArray_digit2 = np.empty((0, 8))

# splitting the each digits feature vectors to plot mean and variance
for i in range(len(featArray)):

  if featArray[i][-1] == 0:
    featArray_digit0 = np.append(featArray_digit0, [featArray[i]], axis=0)
  elif featArray[i][-1] == 1:
    featArray_digit1 = np.append(featArray_digit1, [featArray[i]], axis=0)
  elif featArray[i][-1] == 2:
    featArray_digit2 = np.append(featArray_digit2, [featArray[i]], axis=0)
  else:
    print("Something went wrong!!")

# print(len(featArray_digit0))
# print(len(featArray_digit1))
# print(len(featArray_digit2))

def findMeanVariance(featVector):

  mean = np.mean(featVector, axis=0)
  variance = np.var(featVector, axis=0)

  # print(f'digit: {featVector[0][-1]} \n mean: {mean}, \nvariance: {variance}', end='\n\n')

  plt.subplot(121), plt.bar([1,2,3,4,5,6,7], mean[:7]),
  plt.title(f"Mean of digit_{featVector[0][-1]:.0f}")
  plt.subplot(122), plt.bar([1,2,3,4,5,6,7], variance[:7]),
  plt.title(f"Variance of digit_{featVector[0][-1]:.0f}")
  plt.show()

  print("\n")

  return mean, variance

mean_digit0, var_digit0 = findMeanVariance(featArray_digit0)
mean_digit1, var_digit1 = findMeanVariance(featArray_digit1)
mean_digit2, var_digit2 = findMeanVariance(featArray_digit2)
```

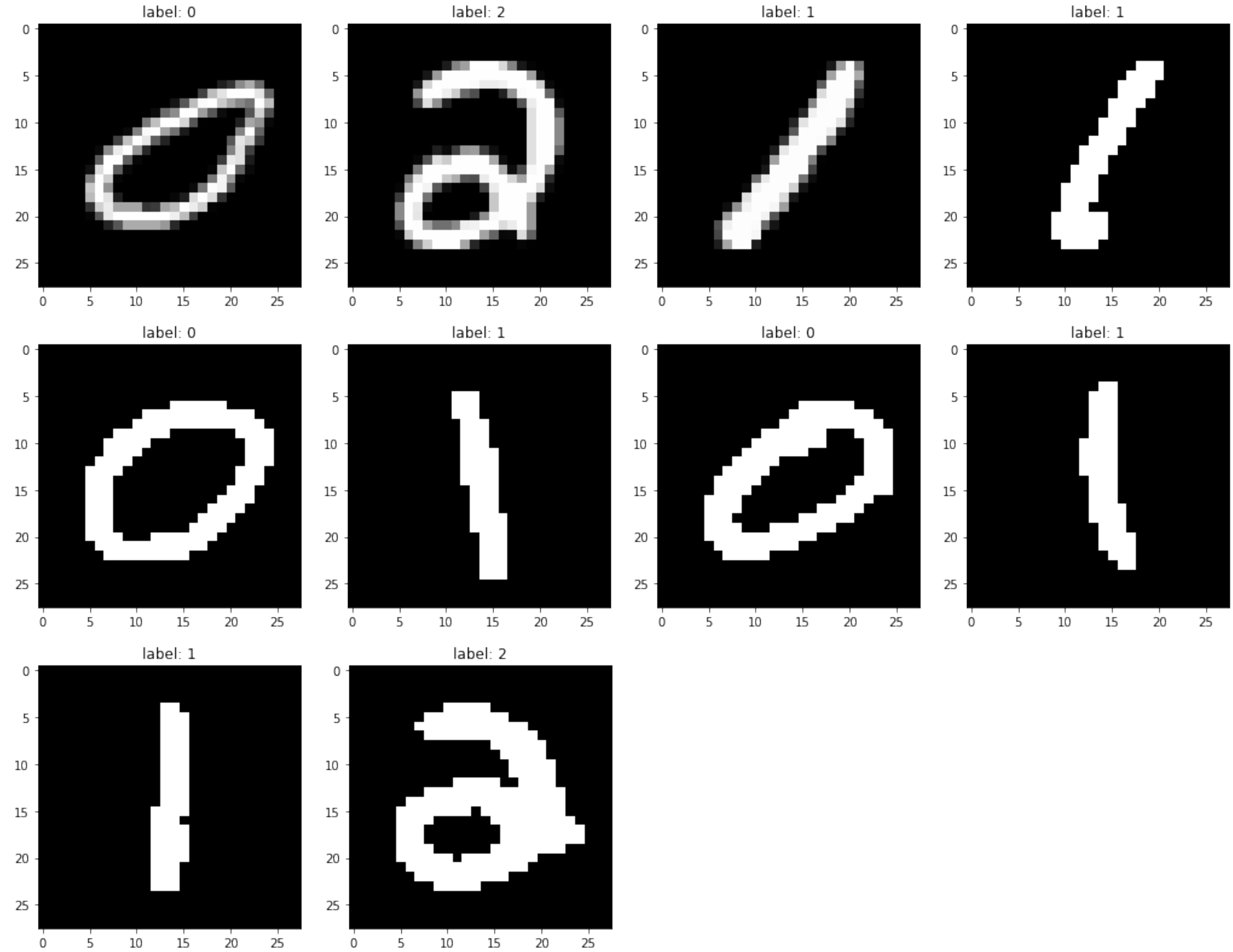
```python
mean_array = np.array([mean_digit0[:7], mean_digit1[:7], mean_digit2[:7]])
var_array = np.array([var_digit0[:7], var_digit1[:7], var_digit2[:7]])

# print(mean_array)

meanFeatDF = pd.DataFrame(mean_array,
                  columns = ['Non-zero Pxls', 'W / H', 'Area', 'Perimeter',
                                       'Moment_X', 'Moment_Y', 'W'],
                  index = ['Digit_0', 'Digit_1', 'Digit_2'])

varFeatDF = pd.DataFrame(var_array,
                  columns = ['Non-zero Pxls', 'W / H', 'Area', 'Perimeter',
                                       'Moment_X', 'Moment_Y', 'W'],
                  index = ['Digit_0', 'Digit_1', 'Digit_2'])

print("\nMean Features Table:")
print(meanFeatDF)
print("\nVariance Features Table:")
print(varFeatDF)

"""## 2.3 Dimension Reduction - Apply PCA"""

def dimensionReductionWithPlot(features, labels, numComponents):
  mean = np.empty((0))
  mean, eigenvectors, eigenvalues = cv2.PCACompute2(features[0:7], mean, maxComponents=numComponents)
  # print(f"{mean}, \n{eigenvectors}, \n{eigenvalues}")

  featArray7 = features[:]
  # print(featArray7.shape)

  numComponentTrainingSet = cv2.PCAProject(featArray7, mean, eigenvectors)

  if numComponents == 2:
    print(f"\n{numComponents} Components PCA:\n {numComponentTrainingSet}")
    print(numComponentTrainingSet.shape)

    plt.scatter((numComponentTrainingSet[:,0])[np.isin(labels, 0)], (numComponentTrainingSet[:,1])[np.isin(labels, 0)],
    color='red', marker='.')
    plt.scatter((numComponentTrainingSet[:,0])[np.isin(labels, 1)], (numComponentTrainingSet[:,1])[np.isin(labels, 1)],
    color='black', marker='^')
    plt.scatter((numComponentTrainingSet[:,0])[np.isin(labels, 2)], (numComponentTrainingSet[:,1])[np.isin(labels, 2)],
    color='yellow', marker='s')
    plt.show()

  return numComponentTrainingSet

twoPCAComponentTraining = dimensionReductionWithPlot(featArray, trainLabels, 2)

"""## 2.4 Classification: Train SVM and plot confusion matrix"""

def svmClassifier(trainingData, labels):
  #Create a svm Classifier
  clf = svm.SVC(kernel='rbf', C=0.55)

  #Train the model using the training sets
  clf.fit(trainingData, labels)

  #Predict the response for test dataset
  PredictedTwoComponent = clf.predict(trainingData)
  accuracy = metrics.accuracy_score(labels, PredictedTwoComponent)
  # print("Accuracy:", accuracy)
  # print(metrics.confusion_matrix(labels, PredictedTwoComponent))
```

```python

  return clf

twoComponentClf = svmClassifier(twoPCAComponentTraining, trainLabels)

# #Predict the response for test dataset
# PredictedTwoComponent = clf.predict(twoPCAComponentTraining)
# accuracy = metrics.accuracy_score(trainLabels, PredictedTwoComponent)
# print("Accuracy:", accuracy)

"""Confusion Matrix: with 2 PCA components"""

# metrics.confusion_matrix(trainLabels, PredictedTwoComponent)

"""# 2.5 Optimizing number of components"""

# validation accuracy
# first need to process the validation data

validationImages = preProcessingImgs(validationImages)
validationFeatures = featureExtraction(validationImages, validationLabels)

# test the models for all 7 components:

for i in range(0, 7):
  i_dimensionsFeatures = dimensionReductionWithPlot(featArray, trainLabels, i)
  i_ValidationFeatures = dimensionReductionWithPlot(validationFeatures, validationLabels, i)

  if i == 0:
    print(f"Training Accuracy of 7 components:\n")
  else:
    print(f"Training Accuracy of {i} components:\n")

  clf = svmClassifier(i_dimensionsFeatures, trainLabels)

  Predicted_i_Training_Component = clf.predict(i_dimensionsFeatures)
  TrainingAccuracy = metrics.accuracy_score(trainLabels, Predicted_i_Training_Component)
  print("Training Accuracy:", TrainingAccuracy)
  print("Training Confusion Matrix:")
  print(metrics.confusion_matrix(trainLabels, Predicted_i_Training_Component))


  print(f"\nValidation Accuracy of {i} components:\n")
  Predicted_i_Validation_Component = clf.predict(i_ValidationFeatures)
  ValidationAccuracy = metrics.accuracy_score(validationLabels, Predicted_i_Validation_Component)
  print("Validation Accuracy:", ValidationAccuracy)
  print("Validation Confusion Matrix:")
  print(metrics.confusion_matrix(validationLabels, Predicted_i_Validation_Component))
  print("############################################")
  print(end='\n\n')

"""# 2.6 Evaluation"""

# first need to process the test data

optimal_n = 2

testImages = preProcessingImgs(testImages)
testFeatures = featureExtraction(testImages, testLabels)

two_dimensionsFeatures = dimensionReductionWithPlot(featArray, trainLabels, optimal_n)
two_testFeatures = dimensionReductionWithPlot(testFeatures, testLabels, optimal_n)
```

```python

clf = svmClassifier(two_dimensionsFeatures, trainLabels)

PredictedTraining_Component = clf.predict(two_dimensionsFeatures)
TrainingAccuracy = metrics.accuracy_score(trainLabels, PredictedTraining_Component)
print("Training Accuracy:", TrainingAccuracy)
print("Training Confusion Matrix:")
print(metrics.confusion_matrix(trainLabels, PredictedTraining_Component))


print(f"\nTest Accuracy of {optimal_n} components:\n")
PredictedTest_Component = clf.predict(two_testFeatures)
TestAccuracy = metrics.accuracy_score(testLabels, PredictedTest_Component)
print("Test Accuracy:", TestAccuracy)
print("Test Confusion Matrix:")
print(metrics.confusion_matrix(testLabels, PredictedTest_Component))

"""10 Random images that are correctly classified and 10 images that are incorrectly classified"""

# print(PredictedTest_Component)
# print(testLabels)

# print((PredictedTest_Component==testLabels).all())

comparisonList = np.equal(PredictedTest_Component, testLabels)
comparisonList.shape

FalseDetected = 0
TrueDetected = 0
for i, Value in enumerate(comparisonList):
  # print(i, Value)
  if Value == False and FalseDetected < 10:
    # print(PredictedTest_Component[i], testLabels[i], Value)
    print(f'Incorrect image # {FalseDetected}, false classification at index: {i}')
    FalseDetected += 1

    plt.imshow(testImages[i], cmap='gray')
    plt.title('label: ' + str(PredictedTest_Component[i]))
    plt.show()
    print()

  elif Value == True and TrueDetected < 10:
    # print(PredictedTest_Component[i], testLabels[i], Value)
    print(f'Correct image # {TrueDetected}, true classification at index: {i}')
    TrueDetected += 1

    plt.imshow(testImages[i], cmap='gray')
    plt.title('label: ' + str(PredictedTest_Component[i]))
    plt.show()
    print()

  if FalseDetected == 10 and TrueDetected == 10:
    break
```

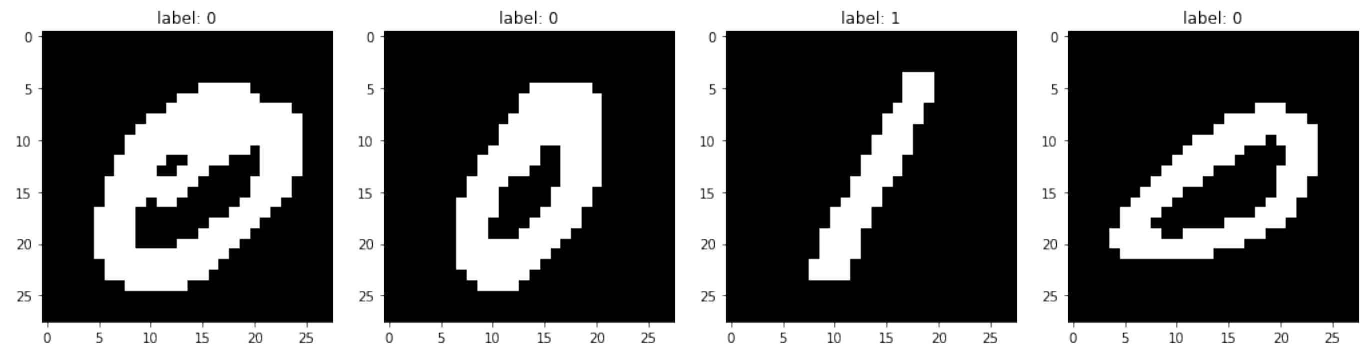
Result:

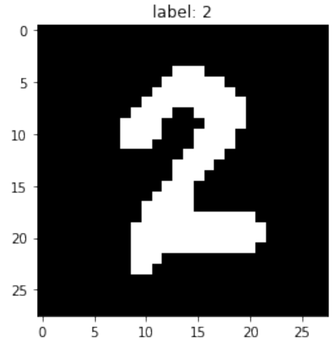
2 - 10 Random images with label:



2.1 - Pre-processing Step:

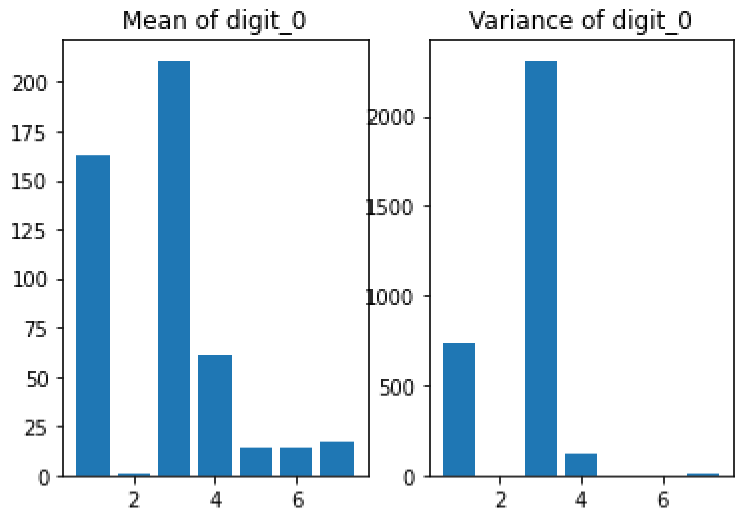
5 random images are showing after binarizing and applying gaussian blur filter:

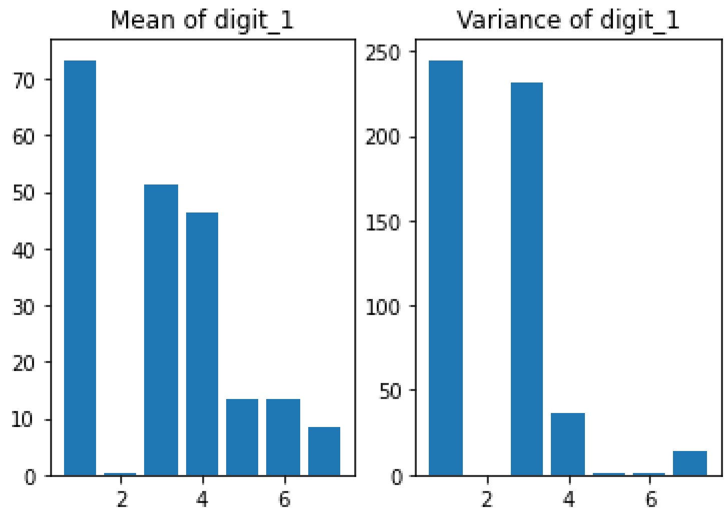


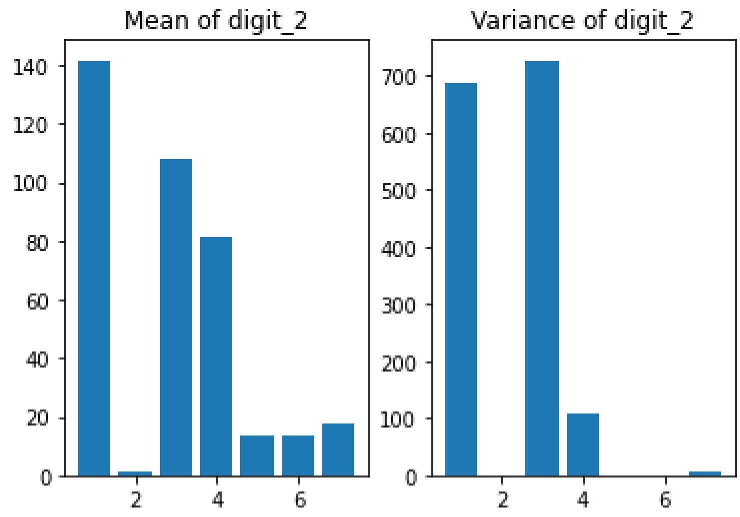


2.2 - Feature Extraction:

My feature extractor has 7 features because both x and y direction moments are used.
For instance, features of an image: [174., 0.95, 254.5, 61.69848394, 14., 15., 19., 0.,]





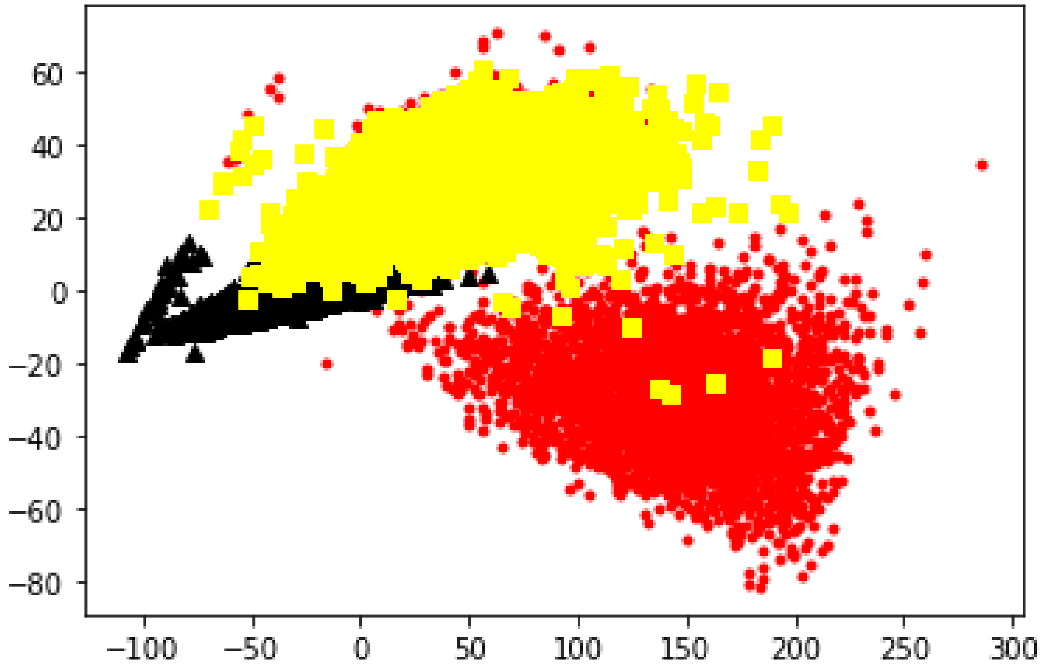



```
Mean Features Table:
          Non-zero Pxls     W / H        Area  Perimeter   Moment_X   Moment_Y  \
Digit_0      162.992012  0.918013  210.915759  61.230433  13.594529  13.627693
Digit_1       73.300524  0.429090   51.181571  46.352767  13.516021  13.418848
Digit_2      141.578692  0.944999  107.986683  81.201382  13.361743  13.737046

                 W
Digit_0  17.548293
Digit_1   8.477906
Digit_2  17.624939

Variance Features Table:
          Non-zero Pxls     W / H         Area   Perimeter  Moment_X  Moment_Y  \
Digit_0      736.358929  0.033930  2311.126890  124.000329  0.637578  0.629240
Digit_1      244.770628  0.036933   231.806509   36.645001  0.932466  0.886346
Digit_2      685.591992  0.050183   726.153576  108.739073  0.433791  0.490177

                 W
Digit_0   6.922565
Digit_1  14.183334
Digit_2   6.930274
```

Following features: number of non zero pixels, area, perimeter and moments seem to the most important.

2.3 - Dimension Reduction:

PCA with 2 components:



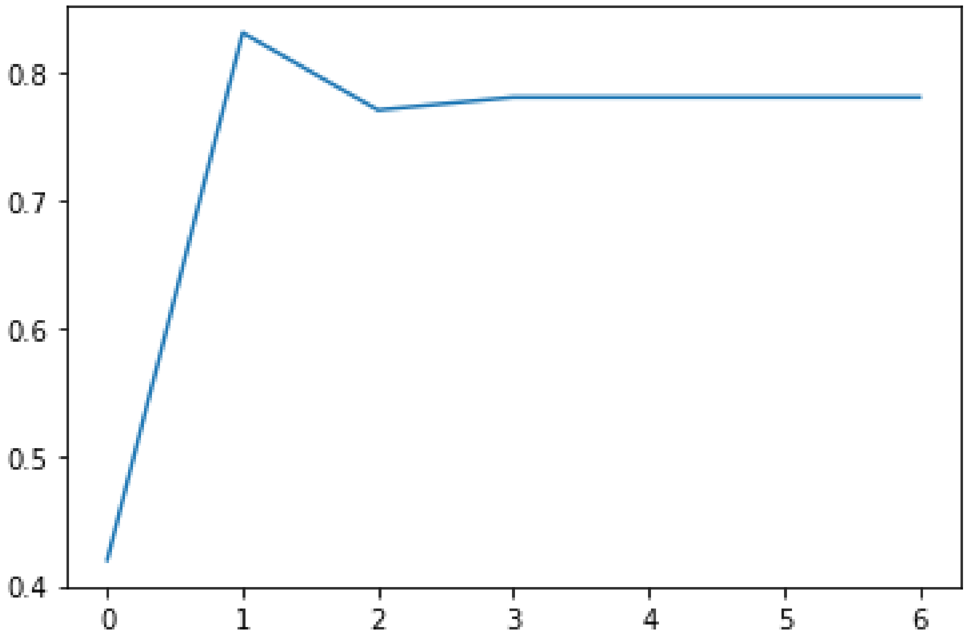
2.4 - Classification:

Confusion matrix of the training data =

```
[[3809   18  304]
 [   2 4677   96]
 [  20   75 4035]]
```

2.5 - Optimizing number of components:

Optimal number of components = 2 because the accuracy is the highest.
Plot below shows the accuracy of the validation data as the number of components are altered:



```
Training Accuracy of 1 components:

[[3353   18  760]
 [   0 4492  283]
 [ 344  275 3511]]
Training Accuracy: 0.8711261123043879
Training Confusion Matrix:
[[3353   18  760]
 [   0 4492  283]
 [ 344  275 3511]]

Validation Accuracy of 1 components:

Validation Accuracy: 0.41944146079484423
Validation Confusion Matrix:
[[ 257  120  810]
 [ 283 1041    0]
 [   3  946  264]]
###########################################
```

```
Training Accuracy of 2 components:

[[3809   18  304]
 [   2 4677   96]
 [  20   75 4035]]
Training Accuracy: 0.9604940165694998
Training Confusion Matrix:
[[3809   18  304]
 [   2 4677   96]
 [  20   75 4035]]

Validation Accuracy of 2 components:

Validation Accuracy: 0.825187969924812
Validation Confusion Matrix:
[[ 842  208  137]
 [ 155 1165    4]
 [  10  137 1066]]
###########################################
```

```
Training Accuracy of 3 components:

[[3827   13  291]
 [   2 4682   91]
 [  23   68 4039]]
Training Accuracy: 0.9625652040503222
Training Confusion Matrix:
[[3827   13  291]
 [   2 4682   91]
 [  23   68 4039]]

Validation Accuracy of 3 components:

Validation Accuracy: 0.7722878625134264
Validation Confusion Matrix:
[[ 894  131  162]
 [ 438  879    7]
 [  13   97 1103]]
###########################################
```

```
Training Accuracy of 4 components:

[[3829   12  290]
 [   2 4688   85]
 [  24   64 4042]]
Training Accuracy: 0.9634090211721387
Training Confusion Matrix:
[[3829   12  290]
 [   2 4688   85]
 [  24   64 4042]]

Validation Accuracy of 4 components:

Validation Accuracy: 0.7824919441460795
Validation Confusion Matrix:
[[ 890  125  172]
 [ 399  918    7]
 [  10   97 1106]]
###########################################
```

```
Training Accuracy of 5 components:

[[3829   12  290]
 [   2 4688   85]
 [  24   64 4042]]
Training Accuracy: 0.9634090211721387
Training Confusion Matrix:
[[3829   12  290]
 [   2 4688   85]
 [  24   64 4042]]

Validation Accuracy of 5 components:

Validation Accuracy: 0.7776584317937701
Validation Confusion Matrix:
[[ 891  125  171]
 [ 418  899    7]
 [  10   97 1106]]
###########################################
```

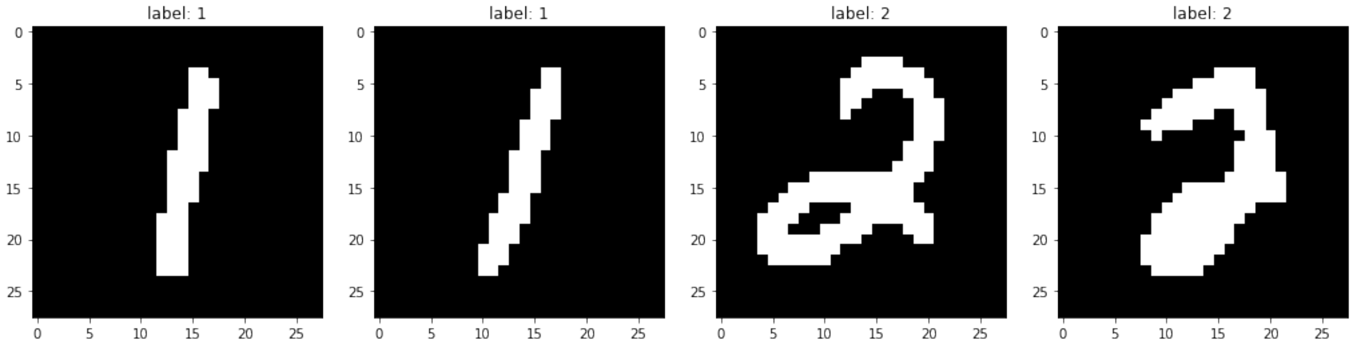
```
Training Accuracy of 6 components:

[[3829   12  290]
 [   2 4689   84]
 [  25   63 4042]]
Training Accuracy: 0.9634857318195765
Training Confusion Matrix:
[[3829   12  290]
 [   2 4689   84]
 [  25   63 4042]]

Validation Accuracy of 6 components:

Validation Accuracy: 0.7792696025778733
Validation Confusion Matrix:
[[ 889  125  173]
 [ 416  901    7]
 [   8   93 1112]]
###########################################
```

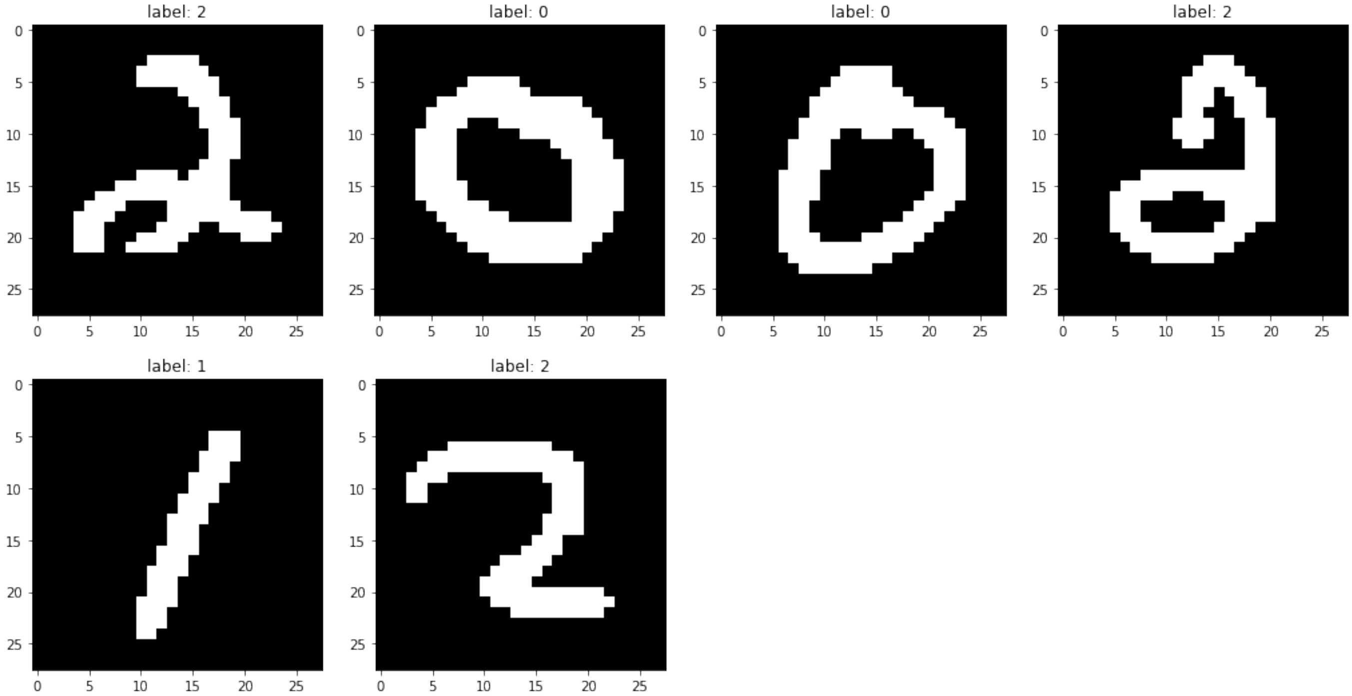
```
Training Accuracy of 7 components:

[[3829   12  290]
 [   2 4689   84]
 [  24   64 4042]]
Training Accuracy: 0.9634857318195765
Training Confusion Matrix:
[[3829   12  290]
 [   2 4689   84]
 [  24   64 4042]]

Validation Accuracy of 0 components:

Validation Accuracy: 0.7835660580021482
Validation Confusion Matrix:
[[ 886  126  175]
 [ 392  925    7]
 [   8   98 1107]]
###########################################
```

2.6 - Evaluation:

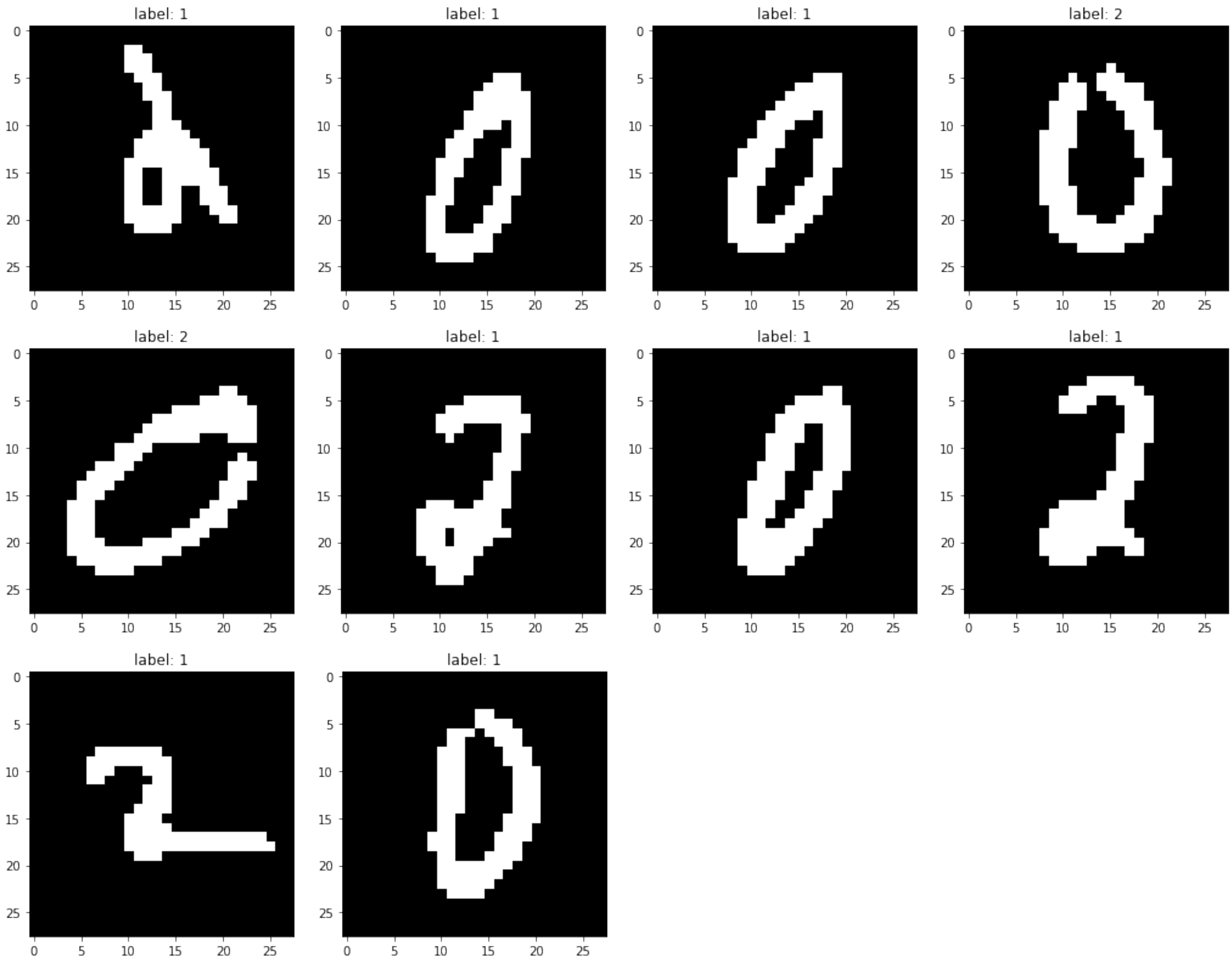
```
Training Accuracy: 0.9604940165694998
Training Confusion Matrix:
[[3809   18  304]
 [   2 4677   96]
 [  20   75 4035]]

Test Accuracy of 2 components:

Test Accuracy: 0.917337627482555
Test Confusion Matrix:
[[517  49  39]
 [  1 638   4]
 [  3  58 554]]
```

10 Correct Images:





10 Incorrect Images:



Bibliography

[1] 1.4. support vector machines. https://scikit-learn.org/stable/modules/svm.html.

[2] Introduction to motion estimation with optical flow. https://nanonets.com/blog/optical-flow/.

[3] Introduction to principal component analysis (pca). https://docs.opencv.org/3.4/d1/dee/tutorial_introduction_to_pca.html.

[4] Optical flow. https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_video/py_lucas_kanade/py_lucas_kanade.html.