

영상처리 PROJECT 2



지도교수	오 병 태 교수님
제출일	2021-06-13
이름	2016124103 박상준

Project2

1. (20%) (K-means/Mean shift) Implement the followings, and analyze and compare the results.

1) Implement the k-means and Mean shift algorithms.

K-means algorithm:

- K-means 알고리즘을 구현하기 위하여 image를 read한 후 shape가 64x64,3 인 array를 reshape를 통하여 RGB 값을 가지는 512*512 vector X_i 를 만든다.
- X 집합에서 무작위로 k 개의 vector를 뽑고 이를 k 개 group의 대표 means로 지정한다.

```
X=image.reshape(row*col,3) # RGB value가지는 vector로 reshape
k=10 # k 값 지정

means=np.zeros((k,3),dtype='int')
M=np.zeros(row*col)

# X로부터 대표값 (means) 무작위로 지정
for i in range(k):
    rand=np.random.randint(0, col*row)
    means[i]=X[rand]
```

- Means와 input vector X 의 거리를 계산하면서 k 번째 Means와 가까운 것을 k 번째 group으로 군집화하고 group의 vector들의 평균을 means vector로 대체하고 means vector가 수렴할 때까지 반복한다.

```
while(True):
    # 이전의 대표값과 현재 대표값이 같다면 break (수렴한 상태)
    if(means.tolist()==before.tolist()):
        break
    before=means.copy()
    # input X와 means의 거리를 구하고 군집화
    for i in range(row*col):
        for K in range(k):
            dist[K]=np.sqrt(np.sum((X[i,:]-means[K,:])**2))
        index[i]=np.argmin(dist)
    means=np.zeros((k,3),dtype='int')
    counter=np.zeros((k))
    # 군집화된 vector들의 평균을 구해 대표값으로 update
    for i in range (row*col):
        for K in range(k):
            if index[i]==K:
                counter[K]+=1
                means[K]=means[K]+X[i]
    for K in range(k):
        means[K]=means[K]/counter[K]
```

Mean shift algorithm:

- 화소의 컬러에 해당하는 3차원 (r,g,b)와 화소의 위치를 나타내는 2차원 (y,x)를 결합하여 5차원 공간으로 매핑한다.

```
# 5차원 벡터 x 만들기
x=np.zeros((row*col,5),dtype='int')
k=0
for i in range(row):
    for j in range(col):
        RGB=image[i,j]
        MN=np.array([i,j])
        S=np.concatenate((MN,RGB), axis=0)
        x[k]=S
        k=k+1

# 초기점 설정
y0=x
```

- 다른 공간의 스케일을 가지는 컬러 공간과 좌표 공간을 위한 커널 함수를 구현한다.

```
for i in range (n):
    # a : RGB 컬러 좌표의 L2 norm / b : 공간 좌표의 L2 norm
    a=np.sqrt(np.sum(((y0[i,2:5]-y0[:,2:5])/15)**2,axis=1))
    b=np.sqrt(np.sum(((y0[i,0:2]-y0[:,0:2])/15)**2,axis=1))
    # K는 커널
    K=np.zeros((a.shape))
    for j in range (n):
        if (a[j]<=1 and b[j]<=1):
            #평편한 커널
            K[j]=1
            #가우시안 커널
            #K[j]=np.exp(-a[j]**2)
```

- K는 다음 식의 함수 K의 return 값이다.

$$\tilde{c} = \frac{\sum_i K\left(\frac{x_i - c}{h}\right) x_i}{\sum_i K\left(\frac{x_i - c}{h}\right)}$$

- K를 이용하여 xi와 곱 진행을 위해 K를 reshape, transform을 한다. 위 식의 C~(y_next)를 구하고 업데이트한다.

- 위와 같은 작업을 수렴할 때까지 반복한다.

- 수렴하여 끝이 나면 각 커널 크기 안에 있는 점들을 모아 군집화하고, 그 군집의 중심을 찾는다.

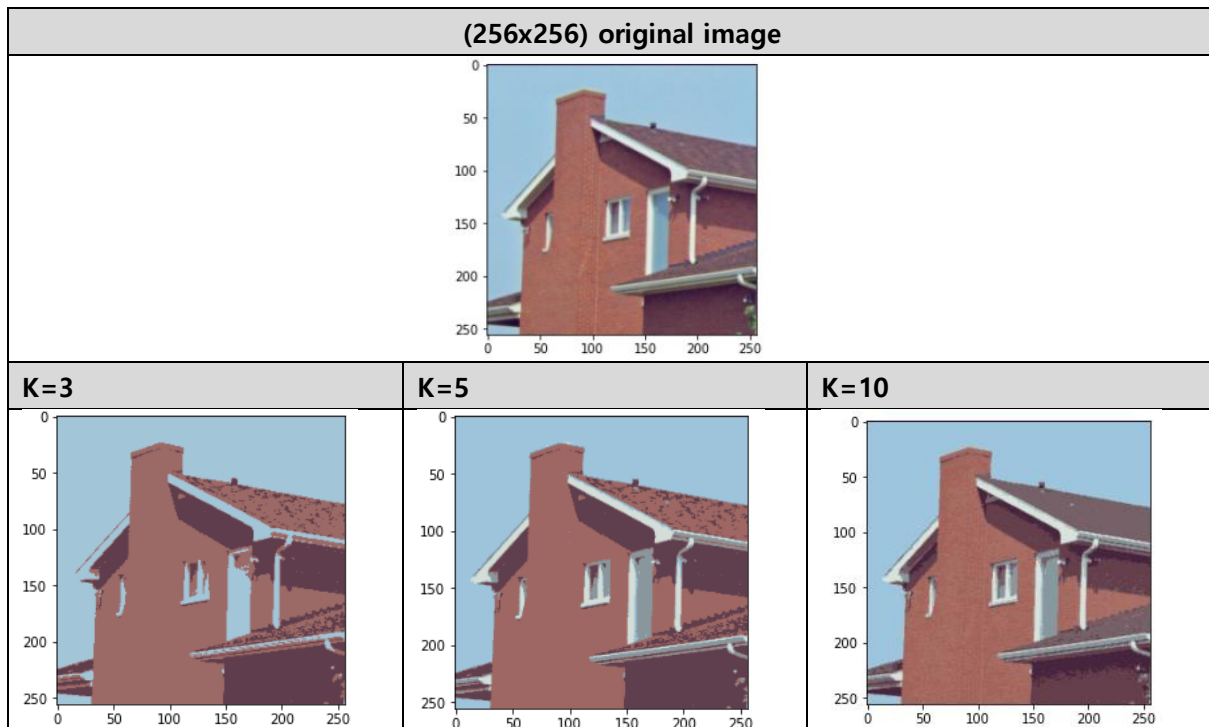
- 군집의 중심을 화소 값으로 가지는 image를 생성한다.

```
v=y0
# origin은 민시프트 결과 이미지
origin=np.zeros((row*col,channel),dtype='int')
# 군집화 후 영상 복구
for i in range (n):
    a=np.sqrt(np.sum(((v[i,2:5]-v[:,2:5])/15)**2,axis=1))
    b=np.sqrt(np.sum(((v[i,0:2]-v[:,0:2])/15)**2,axis=1))
    K=np.zeros((a.shape))
    for j in range (n):
        if (a[j]<=1 and b[j]<=1):
            K[j]=1
    P=np.vstack((K,K,K))

    W=P.T
    m1=np.sum((W*v[:,2:5]),axis=0)
    m2=np.sum(W,axis=0)
    y_next=(m1/m2)
    origin[i]=y_next
```

2) Cluster the image based on the (R, G, B) vector of the image, and visualize it as below.

K-means algorithm:



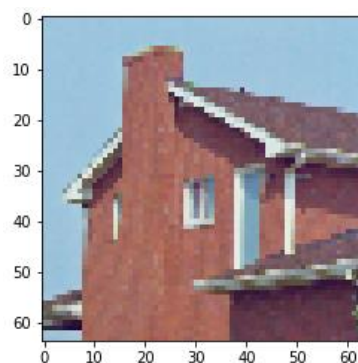
K-means algorithm, mean-shift algorithm 비교

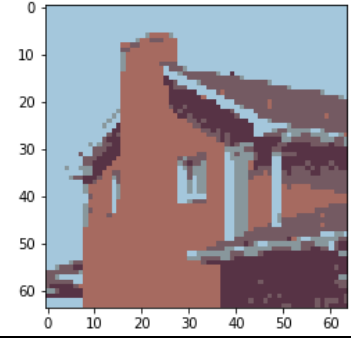
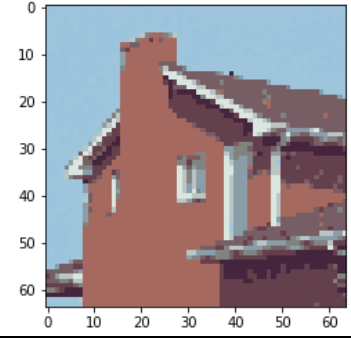
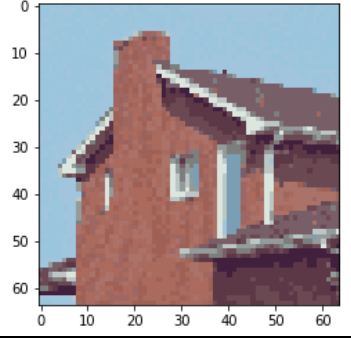
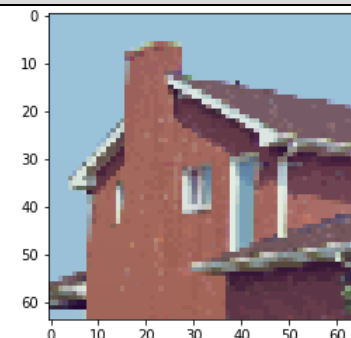
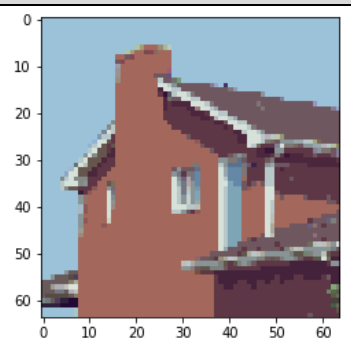
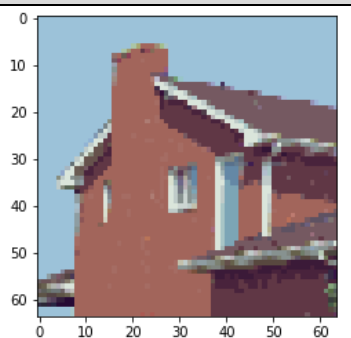
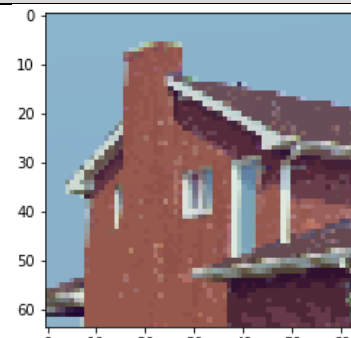
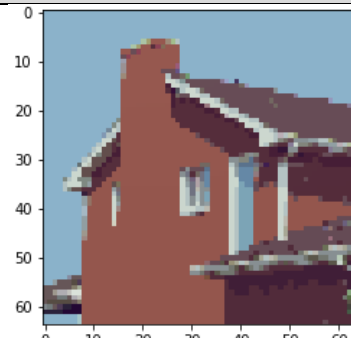
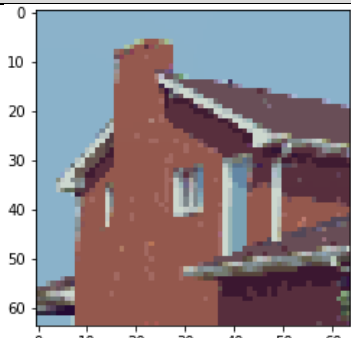
*중요 : mean shift 복잡도가 매우 높아 알고리즘 수행 시간을 단축하고자 original image (256x256)를 1/4크기로 transform한 image (64x64)를 사용함.

+ K-means algorithm과 mean shift algorithm 결과를 비교하고자 transform한 image를 똑같이 사용함.

<transformed image>

(Bilinear interpolation 적용)



K-means (K=3)	K-means (K=10)	K-means (K=15)
		
평편한 커널		
Mean shift (hs=8,hr=7)	Mean shift (hs=15,hr=15)	Mean shift (hs=20,hr=10)
		
가우시안 커널		
Mean shift (hs=8,hr=7)	Mean shift (hs=15,hr=15)	Mean shift (hs=20,hr=10)
		

Mean shift algorithm은 K-means algorithm에 비해 연산 복잡도가 매우 높아 수행시간이 오래걸린다. 특히 image 크기가 커지면 커질수록 연산량이 기하급수적으로 증가하기 때문에 image 분할에 적용할 때에는 충분한 시간이 필요하다.

Mean shift의 커널 크기를 크게 하는 것은 K-means 알고리즘에서 K 값을 작게 하는 것과 같은 효과를 준다.

2. (50%) (Local Descriptor) Based on the descriptor designed in FL class, design the followings.

1) Apply k-means (k=10) algorithm to MNIST test dataset, where input dimension is $28 \times 28 = 784$.

```
X=data
k=20
refer=np.zeros((k))
input_size=10000
means=np.zeros((k,784))
for i in range (k):
    random_index=np.random.choice(70000,1)
    means[i]=mnist_data[random_index]
    refer[i]=mnist_target[random_index]
```

위 코드와 같이 군집의 수 k를 지정하고 mnist_data에서 무작위로 k개를 뽑아 비지도 학습을 진행한다. 무작위로 k개를 뽑고 그 군집의 label에 해당하는 mnist_target 정보를 refer에 담는다.

Refer는 이후 군집화된 input data가 옳게 분류되었는지 확인하는데 사용된다. k개를 무작위로 뽑았으므로 지도학습과 다르게 군집의 label은 0~9 사이의 label의 전부가 포함되지 않았을 가능성이 있어 분류 정확도가 경우에 따라 낮을 수 있다. k개를 증가시킬수록 label의 전부가 포함될 수 있을 가능성이 증가하므로 정확도도 같이 증가한다.

k-means code

```
while(True):
    # 이전의 대표값과 현재 대표값이 같다면 break (수렴한 상태)
    if(means.tolist()==before.tolist()):
        break
    before=means.copy()
    # input X와 means의 거리를 구하고 군집화
    for i in range(input_size):
        for K in range(k):
            dist[K]=np.sum((X[i]-means[K])**2)

        index[i]=np.argmin(dist)
    print(index)
    means=np.zeros((k,784),dtype='int')
    counter=np.zeros((k))
    # 군집화된 vector들의 평균을 구해 대표값으로 update
    for i in range (input_size):
        for K in range(k):
            if index[i]==K:
                counter[K]+=1
                means[K]=means[K]+X[i]
    for K in range(k):
        means[K]=means[K]/counter[K]
```

28*28의 784 vector 1000개를 input으로 하고 다음과 같이 반복한다.

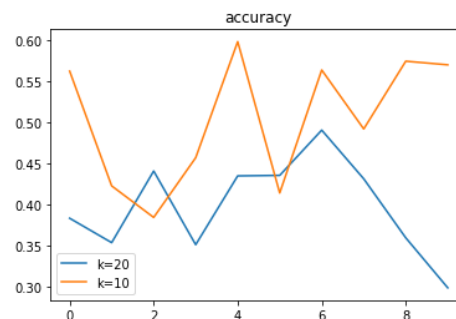
- Means와 input vector X의 거리를 계산하면서 k번째 Means와 가까운 것을 k번째 group으로 군집화하고 group의 vector들의 평균을 means vector로 대체하고 means vector가 수렴할 때까지 반복한다.

2) Measure the clustering accuracy by counting the number of incorrectly grouped images, and analyze the results.

Accuracy:

```
right=0
target=np.array(target,dtype='int')
for i in range(10000):
    if(target[i]==refer[output[i]]):
        right+=1
accuracy=right/input_size
print("K :",k, "/ accuracy :",accuracy)
```

정확도를 구할 때 군집화된 input X의 target(label)이 군집의 label과 같다면 정답으로 간주하였다.



iteration: 1 / K : 10 / accuracy : 0.3838	iteration: 1 / K : 20 / accuracy : 0.5629
iteration: 2 / K : 10 / accuracy : 0.3541	iteration: 2 / K : 20 / accuracy : 0.4232
iteration: 3 / K : 10 / accuracy : 0.4413	iteration: 3 / K : 20 / accuracy : 0.3848
iteration: 4 / K : 10 / accuracy : 0.3516	iteration: 4 / K : 20 / accuracy : 0.4575
iteration: 5 / K : 10 / accuracy : 0.4354	iteration: 5 / K : 20 / accuracy : 0.5988
iteration: 6 / K : 10 / accuracy : 0.436	iteration: 6 / K : 20 / accuracy : 0.4146
iteration: 7 / K : 10 / accuracy : 0.4911	iteration: 7 / K : 20 / accuracy : 0.5643
iteration: 8 / K : 10 / accuracy : 0.4318	iteration: 8 / K : 20 / accuracy : 0.4925
iteration: 9 / K : 10 / accuracy : 0.36	iteration: 9 / K : 20 / accuracy : 0.575
iteration: 10 / K : 10 / accuracy : 0.2991	iteration: 10 / K : 20 / accuracy : 0.5706

OPTION 1: K=10 (random 초기화)

→ 군집의 대표를 초기화할 때, random으로 initialize하기 때문에 코드를 실행할 때마다 정확도가 다르게 나타난다. Accuracy가 나쁠 때에는 0.2까지 내려오는 것을 확인할 수 있었다.

OPTION 2: K=20 (random 초기화)

→ K=10 일 때보다 정확도가 높게 나오는 것을 볼 수 있다. 하지만 K가 큰 경우에도 random으로 initialize하기 때문에 경우에 따라 정확도가 낮을 수 있다.

OPTION 3: 지도학습 K=10 (label이 0~9인 MNIST data 10개를 군집의 대표로 초기화)

→ 그룹의 대표를 initialize 할 때, random이 아닌 0~9까지의 label image를 대표로 initialize 한 경우.

- k-means algorithm 구현 이전에 MNIST test dataset을 만든다. Sklearn.datasets를 이용하여 mnist dataset을 load한 뒤 10000장의 data를 뽑아낸다. K-means의 input x는 28x28짜리 mnist image 벡터이다. 0~9 까지의 label을 가진 mnist data를 group화 하기 위해서 0~9의 label을 가진 image data를 group의 대표로 지정해준다.
- 대표 벡터와 input 벡터의 거리를 비교하면서 k-means 군집화를 진행한다.

K=10	K : 10 / accuracy : 0.5159
-------------	----------------------------

```
means[0],means[1],means[2],means[3],means[4],means[5],means[6],means[7],means[8],means[9] =
data[1],data[3],data[5],data[7],data[2],data[0],data[13],data[15],data[17],data[4]
```

다음과 같이 means(군집의 대표)를 (군집 번호):(label) = {0:0,1:1,2:2,3:3,4:4,5:5,6:6,7:7,8:8,9:9} 으로 initialize 하였기 때문에 결과 accuracy는 0.5159 로 고정되었다.

Analysis:

Option1,2에서는 k개를 무작위로 뽑았다. Option3과 다르게 군집의 label은 0~9 사이의 label의 전부가 포함되지 않았을 가능성이 있어 분류 정확도가 실행 될 때마다 random 값에 따라 낮아질 수 있다. k를 증가시킬수록 label의 전부가 포함될 수 있을 가능성이 증가하고 분류가 k가 작을 때보다 섬세히 진행되므로 정확도도 같이 증가한다. 이와 다르게 Option3처럼 사용자가 직접 대푯값을 적절히 지정해 준다면 정확도가 고정되며 사용자의 선택에 따라 성능이 개선 될 가능성이 있다.

3) Divide the input 28×28 image into four 14×14 sub-blocks, and compute the histogram of orientations for each sub-block as below.

- histogram of orientation을 구하기 위해서 먼저 dx, dy map을 구한다.
- dx, dy map을 sobel mask를 사용하여 계산한 후 strength map과 orient map을 계산한다.

```
for y in range(0,28):
    for x in range(0,28):
        grad_map_dx[y,x]=np.sum(sobel_mx[0:3,0:3]*X[y:y+3,x:x+3])
        grad_map_dy[y,x]=np.sum(sobel_my[0:3,0:3]*X[y:y+3,x:x+3])
        strength_map[y,x]=math.sqrt(grad_map_dx[y,x]**2+grad_map_dy[y,x]**2)
        # division by zero 를 피하기 위하여 실시
        if(grad_map_dy[y,x]==0):
            grad_map_dy[y,x]=1
        if(grad_map_dx[y,x]==0):
            grad_map_dx[y,x]=1
        # orient map 구하기
        orient_map[y,x]=(math.atan(grad_map_dy[y,x]/grad_map_dx[y,x])*(180/3.14))+90
```

* image 속 숫자가 아닌 화소는 모두 0이므로 dx, dy 가 0이 되고 orient_map을 구할 때 division by zero 오류가 생겨 수행 불가능

→ 바탕의 dx, dy를 모두 1로 임시로 두고 orient map을 계산 후 dx, dy 를 다시 0으로 고친다.

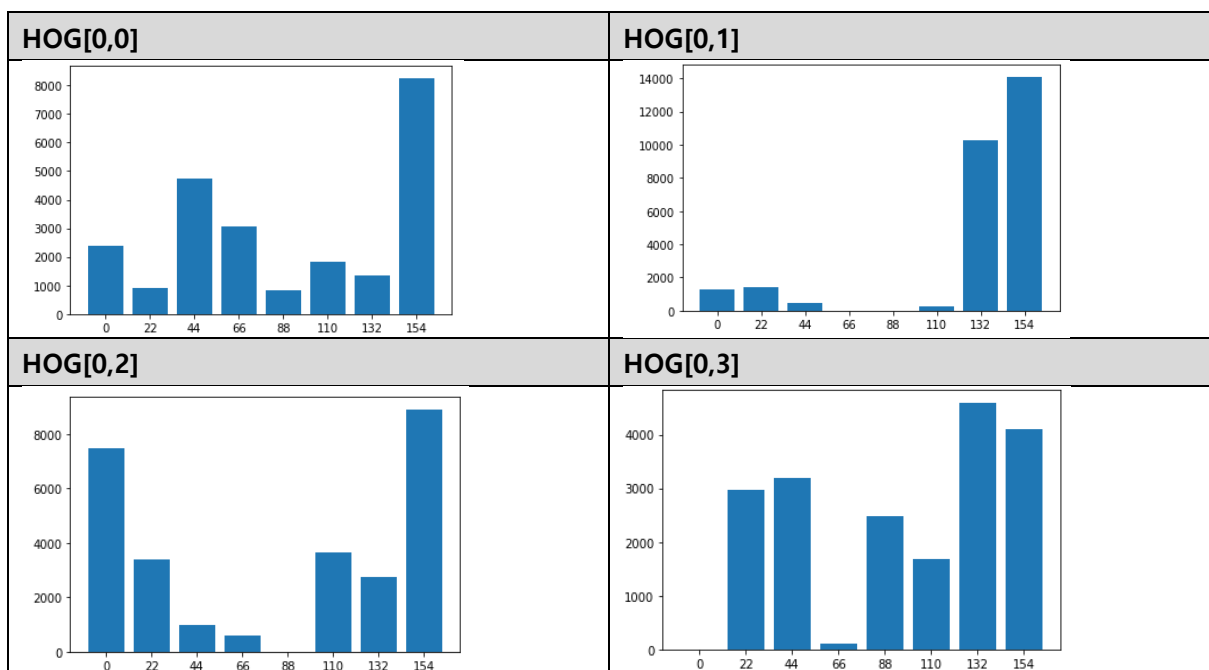
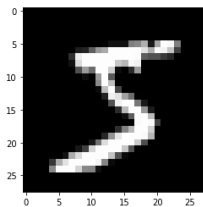
→ strength map의 바탕은 모두 0이므로 HOG에 영향을 주지 않음.

```
# 10000장에 대한 4x8 vector
# HOG=np.zeros((10000,4,8)) 위에서 선언함
for y in range(0,28):
    for x in range(0,28):
        # 0~14 x 0~14 부분
        if(y < 14 and x<14):
            # orient map 에서 orient 를 확인하고 strength map에서 값을 그 위치에 sum.
            if(orient_map[y,x]>0 and orient_map[y,x]<22):
                HOG[i][0][0]+=strength_map[y,x]
            if(orient_map[y,x]>22 and orient_map[y,x]<44):
                HOG[i][0][1]+=strength_map[y,x]
            if(orient_map[y,x]>44 and orient_map[y,x]<66):
                HOG[i][0][2]+=strength_map[y,x]
            if(orient_map[y,x]>66 and orient_map[y,x]<88):
                HOG[i][0][3]+=strength_map[y,x]
            if(orient_map[y,x]>88 and orient_map[y,x]<110):
                HOG[i][0][4]+=strength_map[y,x]
            if(orient_map[y,x]>110 and orient_map[y,x]<132):
                HOG[i][0][5]+=strength_map[y,x]
            if(orient_map[y,x]>132 and orient_map[y,x]<154):
                HOG[i][0][6]+=strength_map[y,x]
            if(orient_map[y,x]>154 and orient_map[y,x]<180):
                HOG[i][0][7]+=strength_map[y,x]
```

- orient map과 strength map을 이용하여 slice 된 4개의 14x14의 sub-block의 histogram of orientations를 구한다. → 4x8 vector

Histogram (HOG)

Target image : data[0]

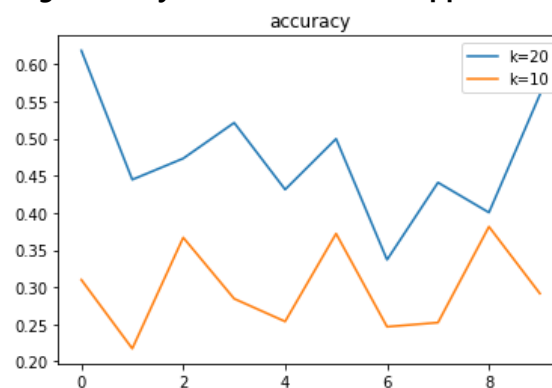


HOG는 numpy array이며 shape가 10000*4*8이다. 10000장의 MNIST data 에서 특징점을 추출한 것이다. 한 장에 대하여 HOG를 확인해보면 위의 표와 같다.

4) Apply k-means (k=10) algorithm again using feature, where input dimension is $8 \times 4 = 32$.

- 1000개의 4x8 vector를 input으로 하는 k-means algorithm을 구현한다.
- 먼저 대푯값을 initialize 하기 위해 random하게 뽑고 뽑힌 대표의 HOG를 구한다. 구한 HOG가 대푯값 z이다.
- 앞에서 설명한 K-means 알고리즘에서 input, 대푯값을 4*8 vectors로 대체,

5) Measure the clustering accuracy for feature-based approach, and analyze the results.



K=10 (4*8 vectors)	K=20 (4*8 vectors)
iteration: 1 / K : 10 / accuracy : 0.31	iteration: 1 / K : 20 / accuracy : 0.6183
iteration: 2 / K : 10 / accuracy : 0.2174	iteration: 2 / K : 20 / accuracy : 0.4447
iteration: 3 / K : 10 / accuracy : 0.3668	iteration: 3 / K : 20 / accuracy : 0.4732
iteration: 4 / K : 10 / accuracy : 0.2845	iteration: 4 / K : 20 / accuracy : 0.5212
iteration: 5 / K : 10 / accuracy : 0.2539	iteration: 5 / K : 20 / accuracy : 0.4314
iteration: 6 / K : 10 / accuracy : 0.3721	iteration: 6 / K : 20 / accuracy : 0.4996
iteration: 7 / K : 10 / accuracy : 0.2468	iteration: 7 / K : 20 / accuracy : 0.3369
iteration: 8 / K : 10 / accuracy : 0.2524	iteration: 8 / K : 20 / accuracy : 0.4408
iteration: 9 / K : 10 / accuracy : 0.3814	iteration: 9 / K : 20 / accuracy : 0.4005
iteration: 10 / K : 10 / accuracy : 0.2915	iteration: 10 / K : 20 / accuracy : 0.559
비교자료: 28*28 vectors K=10	비교자료: 28*28 vectors K=20
iteration: 1 / K : 10 / accuracy : 0.3838	iteration: 1 / K : 20 / accuracy : 0.5629
iteration: 2 / K : 10 / accuracy : 0.3541	iteration: 2 / K : 20 / accuracy : 0.4232
iteration: 3 / K : 10 / accuracy : 0.4413	iteration: 3 / K : 20 / accuracy : 0.3848
iteration: 4 / K : 10 / accuracy : 0.3516	iteration: 4 / K : 20 / accuracy : 0.4575
iteration: 5 / K : 10 / accuracy : 0.4354	iteration: 5 / K : 20 / accuracy : 0.5988
iteration: 6 / K : 10 / accuracy : 0.436	iteration: 6 / K : 20 / accuracy : 0.4146
iteration: 7 / K : 10 / accuracy : 0.4911	iteration: 7 / K : 20 / accuracy : 0.5643
iteration: 8 / K : 10 / accuracy : 0.4318	iteration: 8 / K : 20 / accuracy : 0.4925
iteration: 9 / K : 10 / accuracy : 0.36	iteration: 9 / K : 20 / accuracy : 0.575
iteration: 10 / K : 10 / accuracy : 0.2991	iteration: 10 / K : 20 / accuracy : 0.5706

Accuracy를 구했을 때 위와 같이 나온다. 정확도를 구할 때, 앞에서 설명했던 것과 동일하게 군집화된 input X의 target(label)이 군집 대표의 label과 같다면 정답으로 간주하였다.

그 결과 K=10 일때 0.2~0.4 정도의 정확도가 나왔다. Input이 28*28 vectors 일 때의 정확도 0.3~0.45 보다 작았다. 하지만 28*28 vector가 아닌 특징을 뽑아낸 4*8 vectors를 사용하였기 때문에 수행시간이 28*28 vectors보다 훨씬 빨랐으며 메모리 소모도 작았다. K=20 일때, 눈에 띄게 좋은 성능을 내었다. 정확도가 가장 높을 때 61퍼 정도의 성능을 내었으며 28*28 vectors input일 때의 결과와 비슷한 결과를 볼 수 있었다. 역시 28*28 vectors보다 크기가 작은 4*8 vectors를 사용했음에도 불구하고 좋은 결과를 낼 수 있었다.

K-means 알고리즘을 사용하여 MNIST를 분류할 때, 속도와 메모리 소모에 중점을 두고 있다면 28*28 input vector를 사용하기 보다 4*8 input vector를 사용하는 것이 더 적절할 것으로 예상된다.

3. (30%) (Back propagation) Design the simple single-layer perceptron (SLP) network for IRIS dataset.

1) Use python library to extract images, and separate them into test and training set.

2) Select any two datasets for training and test.

```
import sklearn.datasets as dataset
import numpy as np
#mnist=dataset.fetch_openml('mnist_784')
iris=dataset.load_iris()

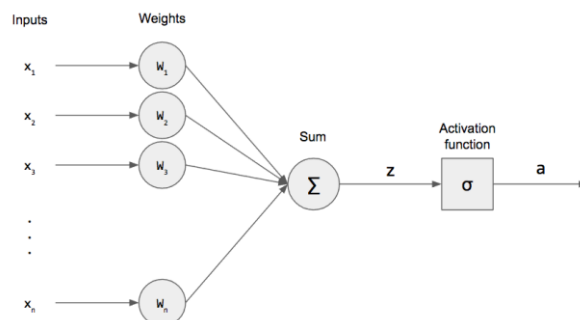
iris_data=iris.data[0:100]
iris_target=iris.target[0:100]
print('iris data shape:',iris_data.shape)
print('iris target shape:', iris_target.shape)

iris_target=iris_target.reshape(100,1)
iris=np.concatenate((iris_data,iris_target), axis=1)

np.random.shuffle(iris)
train_set,test_set=iris[0:80],iris[80:100]
x_train,y_train = np.hsplit(train_set,[4])
y_train=y_train.reshape(80)
x_test,y_test = np.hsplit(test_set,[4])
y_test=y_test.reshape(20)
print(y_train)
```

- Sklearn으로부터 Iris data와 target을 먼저 불러온다.
- Iris data set은 150 개의 sample로 이루어져 있다. 0~50 까지는 class 0, 50~100은 class 1, 100~150은 class 2로 이루어져 있다. Class 0,1 만을 classification하기 위해서 iris data set을 0~100으로 slicing한다.
- Iris data set을 다시 train set과 test set으로 나눌 때 slicing만을 진행한다면 class가 한쪽으로 치우칠 수 있기 때문에, 먼저 dataset shuffle을 진행하고 slicing한다.
- 적절한 Shuffle을 위하여 iris_data와 iris_target을 np.hsplit을 사용하여 axis 1 방향으로 합친 후 set를 shuffle한다. shuffle 이후에 다시 data와 target 을 split한다. (박스 안 코드)

3) Design SLP network to classify them. Use the sigmoid function for activation function.



- single layer perceptron의 activation function으로 sigmoid를 사용하므로 다음과 같이 sigmoid 함수를 definition 한다.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

 여기서 함수 input x는 위 그림의 z와 같다.

```
def train(x,y,epoch):
    n=y.shape[0]
    weight=np.random.randn(5)
    output=np.zeros((n))
    for k in range(epoch):
        for i in range(n):
            t=np.insert(x[i],0,1)
            z=np.dot(t,weight)
            output=sigmoid(z)
            e=y[i]-output
            weight=weight-0.01*e*d_sigmoid(t)
    return weight
```

SLP의 train 알고리즘은 위 코드와 같다. 가장 train에 있어서 가장 중요한 weight를 random으로 initial 해주고 train iteration에 들어간다. Epoch 수만큼 반복하며 iris data input과 weight를 np.dot을 통하여 feed forward를 진행한다.

4) Use the back propagation method to train the network parameters.

- back propagation을 구현하기 위하여 weight 업데이트를 빨간색 박스와 같이 한다. d_sigmoid 함수는 sigmoid의 derivate 함수이다. 즉 update되는 weight는 이전 weight - (learning rate)*(predict error)*(sigmoid derivate(input))으로 update된다.

5) Compute the training and test error for every epoch.

```
def test(x,y,weight):
    n=y.shape[0]
    error=np.zeros((n))
    output=np.zeros((n))
    right=0
    for i in range(n):
        t=np.insert(x[i],0,1)
        z=np.dot(t,weight)
        output[i]=sigmoid(z)
        error[i]=(output[i]-y[i])**2
        if(output[i]<0.5):
            output[i]=0
        else:
            output[i]=1
        if(output[i]==y[i]):
            right+=1

    cost=np.sum(error)/n
    acc=right/n*100
    print("accuracy :",acc,"%", " error :",cost)
    return acc,cost
```

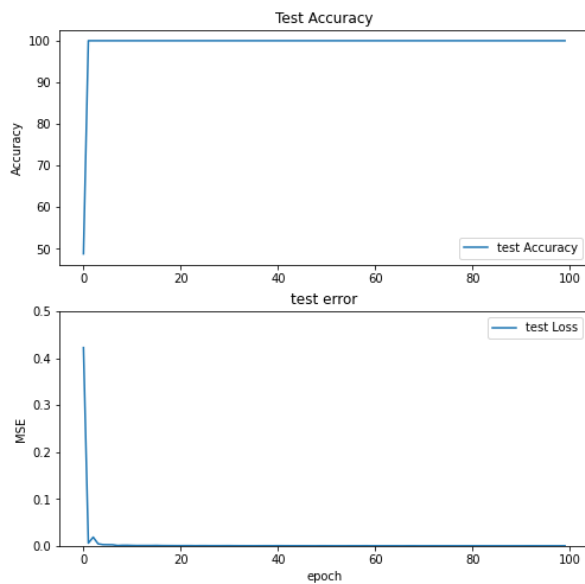
- 학습된 SLP 모델의 weights를 test를 진행하는데 사용한다. Train과 feedforward하는 부분은 동일하나 output을 이용하여 error를 구할 때 MSE를 사용하여 계산하였다. Sigmoid function으로부터 나온 output이 0.5보다 크면 class 1으로, 0.5보다 작으면 class 0으로 classify하였다.

결과는 다음과 같다.

```
acc=[]
loss=[]
for epoch in range(100):
    weight=train(x_train,y_train,epoch)
    accuracy,cost=test(x_train,y_train,weight)
    acc.append(accuracy)
    loss.append(cost)
```

```
def mean_squared_error(y, t):
    return ((y-t)**2).mean(axis=None)
```

Training epoch를 100으로 하고 accuracy와 error를 계산하였다. 여기서 error는 MSE로 구하였다.



```
accuracy : 48.75 %    error : 0.4227225955005398
accuracy : 100.0 %   error : 0.005741832271406984
accuracy : 100.0 %   error : 0.01848892820210584
accuracy : 100.0 %   error : 0.004311730654204309
accuracy : 100.0 %   error : 0.0025209082168864833
accuracy : 100.0 %   error : 0.002348636626835545
accuracy : 100.0 %   error : 0.002222033555277321
accuracy : 100.0 %   error : 0.0006204003535769358
accuracy : 100.0 %   error : 0.0010624984709918486
accuracy : 100.0 %   error : 0.001099649377037045
accuracy : 100.0 %   error : 0.0007880014011550557
accuracy : 100.0 %   error : 0.0005657760212140286
accuracy : 100.0 %   error : 0.0006134235646595865
accuracy : 100.0 %   error : 0.0006068177944237821
accuracy : 100.0 %   error : 0.0006150943322053798
accuracy : 100.0 %   error : 0.000743791938028751
accuracy : 100.0 %   error : 0.00048560546739053937
accuracy : 100.0 %   error : 0.0003869084666494386
accuracy : 100.0 %   error : 0.0003580305794538961
accuracy : 100.0 %   error : 0.00029633176521358086
accuracy : 100.0 %   error : 0.00027083547230937675
```

정확도는 1 epoch 에서 48.75%가 나오고 이후에 100%가 된다.

실제로 다음과 같이 predict 한 값과 실제 target 을 비교하면 모두 일치하는 것을 볼 수 있었다.

```
print("predict :",output[i],"/ target :",y[i])
```

```
predict : 1.0 / target : 1.0
predict : 0.0 / target : 0.0
predict : 1.0 / target : 1.0
predict : 1.0 / target : 1.0
predict : 1.0 / target : 1.0
predict : 0.0 / target : 0.0
predict : 0.0 / target : 0.0
predict : 1.0 / target : 1.0
predict : 1.0 / target : 1.0
predict : 0.0 / target : 0.0
predict : 0.0 / target : 0.0
predict : 1.0 / target : 1.0
predict : 0.0 / target : 0.0
predict : 0.0 / target : 0.0
predict : 1.0 / target : 1.0
predict : 1.0 / target : 1.0
predict : 0.0 / target : 0.0
predict : 1.0 / target : 1.0
predict : 0.0 / target : 0.0
predict : 0.0 / target : 0.0
predict : 1.0 / target : 1.0
predict : 0.0 / target : 0.0
predict : 0.0 / target : 0.0
predict : 1.0 / target : 1.0
predict : 0.0 / target : 0.0
predict : 0.0 / target : 0.0
predict : 1.0 / target : 1.0
predict : 1.0 / target : 1.0
predict : 0.0 / target : 0.0
predict : 1.0 / target : 1.0
predict : 1.0 / target : 1.0
predict : 0.0 / target : 0.0
predict : 1.0 / target : 1.0
accuracy : 100.0 %    error : 0.00022360362795565206
```