BauerMeiners04

Unknown Author

November 11, 2013

Part I

GROUP: Felix Bauer (331805) Jens Meiners (332697)

Exercise 4

In this exercise, you will experiment with different techniques to compute principal components in the context of modeling handwritten characters (small images of 28×28 pixels). A handwritten digit can also be seen as $28 \times 28 = 784$-dimensional signal. Conversely, the 784-dimensional principal components that are learned can also be visualized as 28×28 pixel images.

In [2]:

```
import scipy,scipy.io,numpy,numpy.linalg,Image,IPython.core.display,io

# Load the handwritten characters dataset (only 2500 first samples and subsampled)
#
# input:  none
# output: a matrix containing the data
#
def load():
    X = scipy.io.loadmat('characters.mat')['X'].reshape([-1,56,56]).transpose([0,2,1])
    X = 0.25*(X[:,::2,::2]+X[:,1::2,::2]+X[:,::2,1::2]+X[:,1::2,1::2])
    return X

# Display the data or principal components nicely
#
#  input:  an array of size n*28*28
#  output: a function that displays the data when it is called
#
def show(x):
    n = len(x)
    x = x - x.min()
    x = x / x.max()
    z = numpy.ones([n,30,30])
    z[:,1:-1,1:-1] = x
    x = z.reshape([1,n,30,30]).transpose([0,2,1,3]).reshape([30,n*30])
    b = io.BytesIO()
    img = Image.fromarray((x*255).astype('byte'),'L').save(b, format='png')

    return lambda: IPython.core.display.Image(data=b.getvalue(),format='png', embed=Tr

# Example of execution
X = load()[:30]
show(X)()
```

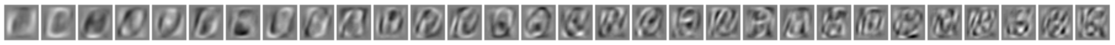
Out [2]:

(a) Compute the principal components using the built-in method for computing eigenvalues and eigenvectors (numpy.linalg.eigh) and visualize the 30 leading principal components using the function show(). Note that the function numpy.linalg.eigh does not sort eigenvalues/eigenvectors automatically. Also, please use the following naming conventions: S = scatter matrix, E = matrix of eigenvectors, $\boldsymbol{e}$ = eigenvector.

In [3]:
```
def calculate_scatter(X):
    n = X.shape[0] # n = 2500
    h = X.shape[1] # h = 28
    w = X.shape[2] # w = 28

    # Rearange Images to be 28*28 dim vectors
    X = X.reshape([n, h * w])

    # Find mean of all images
    mean = X.mean(axis=0)

    # For testing: Show mean
    #meanT = mean.reshape(1,28,28)
    #show(meanT)()

    # Subsract mean from ALL vectors in X
    X -= mean

    # Calculating the outer product for all verctors in X. This is vec * vec.T and giv
    # And sum this into the Scatter Matrix
    S = numpy.zeros([w * h, w * h])
    for vec in X:
        S += numpy.outer(vec, vec)
    return S
```

In [8]:
```
X = load()

n = X.shape[0] # n = 2500
h = X.shape[1] # h = 28
w = X.shape[2] # w = 28

S = calculate_scatter(X)

import time
start = time.time()

# Calculating the Eigenvectors for S
lambd, E = numpy.linalg.eigh(S)

# Sort lambd. argsort will give us the sorted indices. ([::-1 will reverse the array)
lambd = lambd.argsort()[::-1][:30]

# Choose the right Eigenvectors, and reshape them to the picture format
pictures = E[:,lambd].T.reshape(30, h, w)
print "Time for calculating the eigenvectors: " + str(time.time() - start) + " sec"

show(pictures)()
```

```
Time for calculating the eigenvectors: 1.02190995216 sec
```

Out [8]: 

(b) See exercise sheet

ANSWER: The advantages of this kind of classifier would be that it is easy to understand and implement. Also we think that if we have only a few possible classes, it would yield acceptable results. Another advantage is that the algorithm would be very fast (just calulating some dot products, finding the maximum and calculating some posteriors).

On the other hand we think that this kind of classifier is not appropriate when we deal with a lot of classes. It could be that more then one class match best one component. (For exampe the second component above could best match a ‘O’ or a ‘C’ at the same time). So discriminative power of each individual component can be low and lead to bad results.

Also we observe, that the not-so-significant componetents (For example component 25-30 above) look noisy, and are not
appropriate components for this kind of Bayes classifier. But the significant components (1-24 above) are not enough to classify all of our 26 letters.

The classifier wouldn’t be robust against noise in the data.

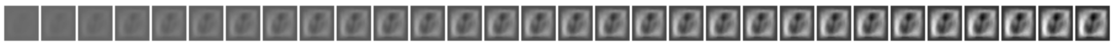
(c) Compute the first principal components without using numpy.linalg.eigh, but instead, iteratively, using repeatedly a gradient descent on the objective function in Eq. 85 of Duda&Hart ($\boldsymbol{e} \leftarrow \boldsymbol{e} - \gamma \cdot \frac{\partial J}{\partial \boldsymbol{e}}$) and a normalization step where $\boldsymbol{e}$ is projected orthogonally onto the subspace defined by $||\boldsymbol{e}||^2 = 1$, (i.e. by normalizing $\boldsymbol{e}$ to have unit norm). The parameter γ is the learning rate and has to be choosen appropriately. Display the vector $\boldsymbol{e}$ at 30 successive iterations for appropriate learning rate using the function show().

In [7]:

```
X = load()

n = X.shape[0] # n = 2500
h = X.shape[1] # h = 28
w = X.shape[2] # w = 28

S = calculate_scatter(X)

import time
start = time.time()
LAMBD = 0.00000000000001
# Because it was not given in the task, we initialize e with ones.
e = numpy.ones([h * w])
e = e / numpy.linalg.norm(e)

pictures = numpy.zeros([30, 28, 28])

for i in range(30):
    # J = - e.T * S * e (+ some unimportant stuff)
    # We use a formula from the matrix cookbook (Eq. 73) to transform the derivative
    e = e + (LAMBD * (S + S.T).dot(e))
    pictures[i] = e.reshape([h, w])
    e = e / numpy.linalg.norm(e)
print "Time for iterative learning: " + str(time.time() - start) + " sec"
show(pictures)()
```

```
Time for iterative learning: 0.357540130615
```

Out [7]:



Note: The resulting image is the same as the first component above. The image above seems to be more grey because show() will do some normalisations. Also note that we don’t see a large impact from LAMBD. A small ‘push’ in the right direction in the first iterations will be enough. show() will increase the contrast. We chose a very small LAMBD to better present the learning effect. LAMBD = 1. would also work.(d) See exercise sheet

ANSWER: The main difference is that with the iterative methode we get the first component only. If we only need the first component the iterative method is totally fine and runs faster.

We are not sure if it is possible to modify the scatter matrix somehow to obtain the next components. It may be possible, but the implementation efford would not be worth the outcome.

If the objectives($J(\omega)$) are differnt, we may not be able to use Eigenvectors to get the solution. In that case finding the maximum with the iterative method is more flexible (although we have to make sure that the local maximum is indeed the global maximum).

From a programming perspective it is always better to use already implemented methods like numpy.linalg.eigh(). It is faster and less error prone. Even If we are just interessted in the first component and runtime is an issue, we should use already implemented functions like scipy.optimize.newton() and not try to implement numerical algorithms ourself.

Submission guidelines

To facilitate grading, please export the notebook to PDF format. This can be done easily by installing the required packages and running

```
ipython nbconvert –to latex sheet04.ipynb && pdflatex sheet04.tex
```