Exercises for the course
Machine Learning for Data Science
Winter Semester 2022/23

G. Montavon
Institute of Computer Science
Department of Mathematics and Computer Science
Freie Universität Berlin

Exercise Sheet 3 (programming part)

In [92]:

```
import numpy,sklearn,sklearn.datasets,utils
%matplotlib inline
```

In the following, we will experiment with two different techniques to compute the PCA components of a dataset: Singular Value Decomposition (SVD) and Power Iteration. We consider a random subset of the Labeled Faces in the Wild (LFW) dataset, readily accessible from sklearn, and we apply some basic preprocessing to discount strong variations of luminosity and contrast.

In [93]:

```
D = sklearn.datasets.fetch_lfw_people(resize=0.5, slice_=(slice(70, 195), slice(78, 172)))[
D = D[numpy.random.mtrand.RandomState(1).permutation(len(D))[:2000]]*1.0
D = D - D.mean(axis=(1,2),keepdims=True)
D = D / D.std(axis=(1,2),keepdims=True)
print(D.shape)
```

```
(2000, 62, 47)
```

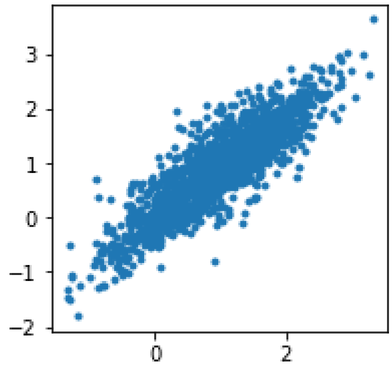
Two functions are provided for your convenience and are available in `utils.py` that is included in the zip archive. The functions are the following:

- **`utils.scatterplot`** produces a scatter plot from a two-dimensional data set.
- **`utils.render`** takes an array of data points or objects of similar shape, and renders them in the IPython notebook.

Some demo code that makes use of these functions is given below.

In [94]:

```
utils.scatterplot(D[:,32,20],D[:,32,21]) # Plot relation between adjacent pixels
utils.render(D[:30],15,2,vmax=5)         # Display first 10 examples in the data
```



Exercise 4 (15 P)

Principal components can be found computing a singular value decomposition. Specifically, we assume a matrix X whose columns contain the data points represented as vectors, and where the data points have been centered (i.e. we have substracted to each of them the mean of the dataset). The matrix X is of size $d \times N$ where d is the number of input features and N is the number of data points. This matrix, more specifically, the rescaled matrix $Z = \frac{1}{\sqrt{N}} X$ is then decomposed using singular value decomposition:

$$U \Lambda V = Z$$

The k principal components can then be found in the first k columns of the matrix U.

(a) Compute the principal components of the data using the function `numpy.linalg.svd` .

(b) Measure the computational time required to find the principal components. Use the function `time.time()` for that purpose. Do *not* include in your estimate the computation overhead caused by loading the data, plotting and rendering.

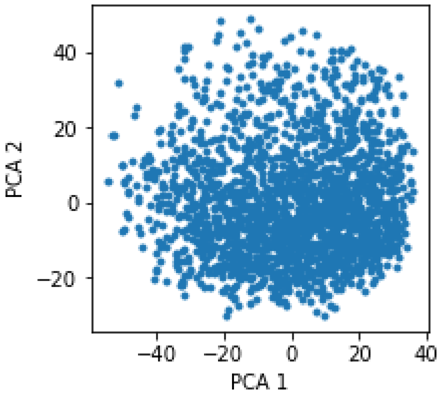
(c) Plot the projection of the dataset on the first two principal components using the function `utils.scatterplot` and visualize the 60 leading principal components using the function `utils.render` .

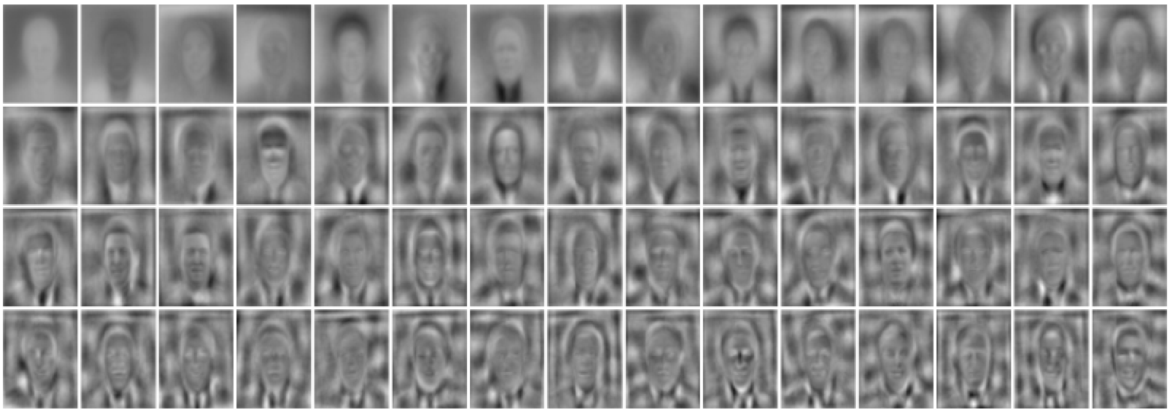
Note that if the algorithm runs for more than 3 minutes, there may be some error in your implementation.

In [126]:

```
import time

X = numpy.array([D[i].flatten() for i in range(D.shape[0])]).T
X -= numpy.mean(X, axis=1).reshape((-1,1))
Z = X / numpy.sqrt(X.shape[1])

start_time = time.time()
U, L, V = numpy.linalg.svd(Z, full_matrices=False)
X_svd = U.T @ X
print(f"Time: {time.time() - start_time: 3.3f} seconds")

utils.scatterplot(X_svd[0,:], X_svd[1,:], xlabel='PCA 1', ylabel='PCA 2')
utils.render(U.T[:60],15,4)
```

```
Time:  7.824 seconds
```





When looking at the scatter plot, we observe that much more variance is expressed in the first two principal components than in individual dimensions as it was plotted before. When looking at the principal components themselves which we render as images, we can see that the first principal components correspond to low-frequency filters that select for coarse features, and the following principal components capture progressively higher-frequency information and are also becoming more noisy.

Exercise 5 (15 P)

The first PCA algorithm based on singular value decomposition is quite expensive to compute. Instead, the power iteration algorithm looks only for the first component and finds it using an iterative procedure. It starts with an initial weight vector $\boldsymbol{w} \in \mathbb{R}^d$, and repeatedly applies the update rule

$$\boldsymbol{w} \leftarrow S\boldsymbol{w} \Big/ \|S\boldsymbol{w}\|.$$

where S is the covariance matrix defined as $S = \frac{1}{N} X X^\top$. Like for standard PCA, the objective that iterative PCA optimizes is $J(\boldsymbol{w}) = \boldsymbol{w}^\top S \boldsymbol{w}$ subject to the unit norm constraint for $\boldsymbol{w}$. We can therefore keep track of the progress of the algorithm after each iteration.

(a) Implement the power iteration algorithm. Use as a stopping criterion the value of $J(\boldsymbol{w})$ between two iterations increasing by less than 0.01. Print the value of the objective function $J(\boldsymbol{w})$ at each iteration.

(b) Measure the time taken to find the principal component.

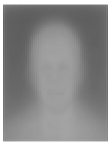
(c) Visualize the the eigenvector $\boldsymbol{w}$ obtained after convergence using the function `utils.render` .

Note that if the algorithm runs for more than 1 minute, there may be some error in your implementation.

In [125]:

```
### REPLACE BY YOUR CODE
EPS = 0.01

X = numpy.array([D[i].flatten() for i in range(D.shape[0])]).T
X -= numpy.mean(X, axis=1).reshape((-1,1))
N = X.shape[1]
S = 1 / N * (X @ X.T)

start_time = time.time()
w = numpy.random.normal(0,1, 62 * 47).reshape((-1,1))
w /= numpy.linalg.norm(w)

iter = 0
J = (w.T @ S @ w)[0, 0]
print(f"iteration {iter:2d}   J(w) = {J:9.3f}")
while True:
    iter += 1
    w = S @ w
    w /= numpy.linalg.norm(w)
    J_new = (w.T @ S @ w)[0, 0]
    print(f"iteration {iter:2d}   J(w) = {J_new:9.3f}")
    if numpy.abs(J - J_new) < EPS:
        print("stopping criterion satisfied")
        break
    J = J_new

print(f"Time: {time.time() - start_time:3.3f} seconds")
utils.render(w, 1, 1)
###
```

```
iteration  0   J(w) =     0.906
iteration  1   J(w) =   133.742
iteration  2   J(w) =   248.450
iteration  3   J(w) =   339.028
iteration  4   J(w) =   364.745
iteration  5   J(w) =   370.268
iteration  6   J(w) =   371.649
iteration  7   J(w) =   372.061
iteration  8   J(w) =   372.196
iteration  9   J(w) =   372.243
iteration 10   J(w) =   372.260
iteration 11   J(w) =   372.266
stopping criterion satisfied
Time: 0.677 seconds
```



We observe that the computation time has decreased significantly. The difference of performance becomes larger as the number of dimensions and data points increases. We can observe that the principal component is the same (sometimes up to a sign flip) as the one obtained by the SVD algorithm.