*P. Koumoutsakos*
*ETH Zentrum, CLT E 13*
*CH-8092 Zürich*

Fall semester 2020

# Set 3 - Principal Component Analysis and Oja's rule

Issued: October 30, 2020
Hand in (optional): November 13, 2020 08:00am

## Question 1: The Covariance Method (60 points)

Principal Component Analysis (PCA) is a classical method to perform dimensionality reduction and uncover structure in data. The method learns an orthogonal transformation that eliminates linear correlations and tries to capture as much variance in the data as possible. The principal components can be computed using the covariance method, by constructing the covariance matrix of the data, $C \in \mathbb{R}^{d\times d}$ and identifying its eigenvalue decomposition. The covariance matrix is given by

$$C = \frac{1}{n-1} X^\top X \,, \tag{1}$$

where $X \in \mathbb{R}^{n\times d}$ is constructed by stacking the dataset. We assume that the data are independent and identically distributted. The eigenvalue decomposition of the symmetric matrix $C$ is given by

$$C = V \Lambda V^{-1}, \tag{2}$$

where $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_d)$ with $\lambda_1 \geq \ldots \geq \lambda_d$. Due to the symmetry of the real covariance matrix, the eigenvectors are orthogonal to each other and they form an orthonormal basis, $V^{-1} = V^\top$. The PCA components are the columns of the eigenvector matrix $V = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d]$. The transformed data are given by $\boldsymbol{y} = V^\top \boldsymbol{x}$ and are linearly uncorrelated.

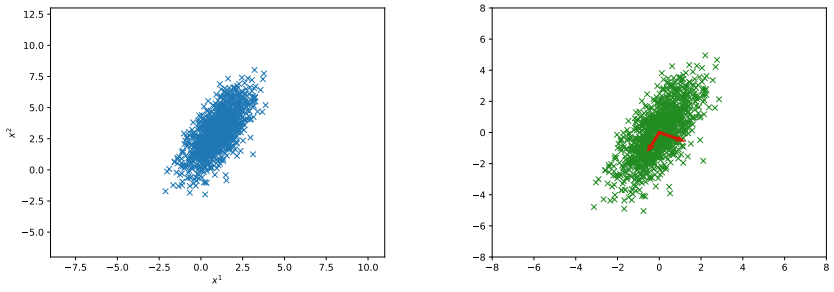


Figure 1: Two dimensional dataset along with its PCA components.

a) Use an appropriate routine provided by the LAPACK software library to compute the PCA of the dataset based on the covariance method. The Intel Math Kernel Library (MKL) includes a high-performance implementation of LAPACK. In order to use the MKL on Euler,

you have to load the module with `module load mkl`. After loading the module, you can include the header `#include<mkl_lapack.h>` to access the LAPACK routines.

Complete the steps in the skeleton code provided in the file `main_pca.cpp`. PCA is sensitive to the relative scaling of the input. For this exercise, **we consider relative input scaling to be relevant**, so the data should **not** be standardized (scaled with zero mean and unit variance). However, the data need to be centered. Write a program that performs the following tasks:

1. read the provided dataset (saved in a memory allocation of $n \times d$, where $n$ is the data-set size and $d$ the data dimension),
2. center the data,
3. compute the covariance matrix of the data,
4. call the `dsyev_()` routine of LAPACK to compute the eigenvalues of the matrix.

You are provided with a two dimensional dataset (`2D_dataset.txt`) with $n = 1024$ samples plotted in Figure 1. Can you recover the principal components plotted in Figure 1? Report the computed eigenvalues.

b) In the following, we apply the PCA routine to a scenario closer to the real world. You are provided with a dataset with $n = 1280$ images of faces. Each face consists of a grayscale image $\boldsymbol{I} \in \mathbb{R}^{h \times w}$ that is flattened to a single data point $\boldsymbol{x} \in \mathbb{R}^d$, with $d = hw$, ignoring spatial structure. For this exercise $h = 50, w = 37$ so $d = 1850$. The dataset is saved in the file `faces_dataset.txt` and each the images are stored row-wise.

Perform PCA on this dataset and save only the $m = 10$ principal components (corresponding to the highest eigenvalues) in a `.txt` file. You have to save a matrix $V_m \in \mathbb{R}^{d \times m}$, obtained from the first $m$ columns of the eigenvectors matrix $V \in \mathbb{R}^{d \times d}$ computed by PCA. Use the provided `python` routine to plot the principal components you computed.

c) The components computed by PCA can be used to compress the dataset. The transformation to the compressed form is given by $Y = XV_m$, where $Y \in \mathbb{R}^{n \times m}$ and $X \in \mathbb{R}^{n \times d}$ is the original image database. Use the computed $m = 10$ components to compress the data and report the compression ratio by measuring the number of floating point numbers needed to represent the original dataset and the compressed one (hint: for the purpose of compression you need to take into account the cost of the scaler, e.g. standardization).

d) Try to reconstruct the original dataset using the compressed data, the $m = 10$ PCA components and the data scaling factors and save the result in a `.txt` file. The reconstruction can be obtained by $X = YV_m^\top$. A `python` routine is provided that can be used to plot the components computed by your method, plot the reconstruction and benchmark (qualitatively) against a reference `python` implementation of PCA. Validate your result with the provided `python` routines.

## Question 2: Principal Component Analysis with Oja's rule (40 points)

Consider a linear network which maps each element of a set of input vectors $\boldsymbol{x}^n \in \mathbb{R}^d, n = 1, \ldots, N$ to some features $y^n \in \mathbb{R}$ through a weight vector $\boldsymbol{w} \in \mathbb{R}^d$,

$$y^n = \boldsymbol{w}^\top \boldsymbol{x}^n. \tag{3}$$

This operation (computing the output of the neurons in a neural network) is commonly referred to as prediction or forward propagation. We have seen that the Hebbian rule for updating the weights $\Delta \boldsymbol{w} = \beta y^n \boldsymbol{x}^n$ can lead to unstable growth or decay of the weight matrix. Oja's algorithm [1, 2], which regularises the Hebbian learning rule, is given by (for small learning rates $\beta \ll 1$):

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \beta y^n (\boldsymbol{x}^n - \boldsymbol{w}_k y^n). \tag{4}$$

In equation (4), $k$ is the learning iteration, $n$ is indexing the current sample processed, and the weight vector $\boldsymbol{w}$ is increased whenever the output $y^n \in \mathbb{R}$ correlates with the regularized quantity $\boldsymbol{x}^n - \boldsymbol{w}_k y^n$. The update rule imposes to the weight vector to have unit length (due to the regularization term $-\boldsymbol{w}_k y^n$) and the iterative learning has a fixed point for $\boldsymbol{w}$ equal to the first eigenvector of the set of input vectors $\lim_{k\to\infty} \boldsymbol{w}^k = \boldsymbol{v}_1$.

In the original formulation of the algorithm in [1] the data was provided as a stream ($k \hat{=} n$) but the convergence analysis also holds for multiple sweeps over a fixed dataset of size $N$. The data can also be provided in batches at the input of the perceptron, similar to classical neural networks training, the update is the update In this case, the update $\Delta \boldsymbol{w}$ is the expected update (average over the batches).

a) Follow the guidelines in the code:

- Implement the forward pass through the perceptron.
- Implement Oja's rule.
- Compute the eigenvalues learned by the Oja's method.
- Then use the learned component to reconstruct the data from the principal component of the Oja's rule, in a similar fashion as PCA.

The eigenvalues can be computed from the standard deviation of the output of the perceptron. Try to learn the principal component of the faces dataset with Hebb's rule. Do you manage to recover it? Why? Implement Oja's rule in the code and identify the principal component in the faces dataset. Compare the computed component with the one recovered with the covariance method and the reference `python` implementation provided. Is the component exactly the same? Is this always the case? What about the computed eigenvalue? Although in theory Oja's rule enjoys convergence properties at equilibrium, in practice it is a heuristic and usually diverges if the parameters are not tuned properly due to overflows (nans and infs in the code). In order to alleviate this problem you can tune/experiment with the following hyper-parameters:

- Weight initialization
- Learning rate and batch size
- Gradient normalization

In order to find the first $M$ eigenvectors of a dataset, we extend the scalar output of the network $y \in \mathbb{R}$, to a vector of neurons $\boldsymbol{y} = [y_1, \ldots, y_M]^\top$ with $\boldsymbol{y} \in \mathbb{R}^M$, each satisfying $y_i = \boldsymbol{w}_i^\top \boldsymbol{x}$. Note that the index $i$ here denotes the component of the multivariate output $\boldsymbol{y}$, and is different from $n$ in relation (4) where it denotes the sample number. For an input sample $\boldsymbol{x}^n$ from the data, the output reads:

$$\boldsymbol{y}^n = \begin{pmatrix} y_1^n \\ \vdots \\ y_M^n \end{pmatrix} = \begin{pmatrix} \boldsymbol{w}_1^\top \boldsymbol{x}^n \\ \vdots \\ \boldsymbol{w}_M^\top \boldsymbol{x}^n \end{pmatrix} = W \boldsymbol{x}^n, \tag{5}$$

where $\boldsymbol{y}_n \in \mathbb{R}^M$, $W \in \mathbb{R}^{M \times D}$, $W = [\boldsymbol{w}_1^\top, \ldots, \boldsymbol{w}_M^\top]$. If trained with equation 4, all weight vectors would converge to the same dominant eigenvector. In order to ensure orthonormality of the weight vectors and enable the learning of more than one principal component, a generalization of Oja's rule, termed generalized Hebbian Learning or Sanger's rule, was proposed in [2, 3, 4], that reads:

$$\boldsymbol{w}_m^{k+1} = \boldsymbol{w}_m^k + \beta \boldsymbol{y}_m \left( \boldsymbol{x}_n - \sum_{l \leq m} \boldsymbol{w}_l^k \boldsymbol{y}_l \right) . \tag{6}$$

b) Based on your implementation of Oja's rule, extent it and implement Sanger's rule. Follow the guidelines in the skeleton code. Use your code to compute the $M = 5$ principal components of the faces data set and compare it with PCA. Start with the computation of one component, and iteratively check how the method works increasing the number of components. What do you observe? Are the eigenfaces (eigenvectors) corresponding to high eigenvalues (e.g. $\boldsymbol{v}_1$ corresponding to $\lambda_1$) as easy to learn as the eigenfaces correspoding to lower eigenvalues (e.g. $\boldsymbol{v}_8$ corresponding to $\lambda_8$)? Can you explain the behavior you observe? What do you observe regarding the stability of the algorithm?

**Note 1:** The theoretical analysis shows that the Oja's and Sanger's rule converge if $\sum_k \beta_k \to \infty$ and $\lim_{k \to \infty} \beta_k = 0$ for an adaptive learning rate $\beta$. In practice, we use a fixed small learning rate, and the methods might take **many iterations** to converge. These rules are mainly useful in computing components of data-streams.

**Note 2:** Note that in order to simplify the notation we are omitting the notion of a bias vector. Since we standardize the data, we do not need to take into account the bias vector. If the data were not centered, each neuron should behave according to $y_m^n = \boldsymbol{w}_m^\top \boldsymbol{x}^n + b_m$, where $m$ is the output neuron (component number) and $n$ is the sample index (from the data). All aforementioned relations hold true by using the convention of hiding bias into the weight vectors according to $\boldsymbol{x}^n \leftarrow \{\boldsymbol{x}^n, 1\}$ and $\boldsymbol{w}_m \leftarrow \{\boldsymbol{w}_m, b_m\}$.

# References


[1] Oja, Erkki Simplified neuron model as a principal component analyzer, Journal of mathematical biology, 1982.

[2] Bruno, A. Olshausen, Linear Hebbian learning and PCA.

[3] Oja, Erkki, Principal components, minor components, and linear neural networks, Neural networks, 1992.

[4] Terence, D. Sanger, Optimal unsupervised learning in a single-layer linear feedforward neural network, Neural Networks, 1989.

[5] Baldi, Pierre and Hornik, Kurt, Neural networks and principal component analysis: Learning from examples without local minima, Neural networks, 1989.