Artificial neural networks - Exercise session 3

Unsupervised learning: PCA and SOM

2016-2017

1 Introduction

1.1 Principal Component Analysis

Principal Component Analysis (PCA) involves projecting onto the eigenvectors of the covariance matrix. The basic idea behind PCA is to map a vector $x = (x_1, x_2, \ldots, x_p)$ of a p dimensional space to a lower-dimensional vector $z = (z_1, z_2, \ldots, z_q)$ in a q dimensional space (where $q < p$). We are going to only consider linear mappings, these are ones where $z_d = e_d^T x$ for some unknown column vectors e_d and T denotes vector (and later on, matrix) transposition. In other words, z can be obtained from x by simple matrix multiplication:

$$z = E^T x \tag{1}$$

where E is a p by q matrix whose column are e_d: $E = [e_1, e_2, \ldots, e_q]$. Our goal is then to reconstruct as well as possible the original vectors by using another matrix F so that

$$\hat{x} = Fz = FE^T x \tag{2}$$

where F is a p by q matrix, and $\hat{x}$ resembles x as good as possible. The idea is that any low dimensional intermediate representation z which allows the original data x to be well reconstructed should have captured a lot of the important structure of the data and so will be interesting or useful to work with.

PCA projects the data onto the subspace spanned by the q eigenvectors corresponding to the q largest eigenvalues of the correlation matrix. More concrete, in order to perform a PCA reduction of a dataset containing N datapoints of dimension p, one can use the following algorithm:

- Zero-mean the data by subtracting the mean of the dataset from each datapoint.
- Calculate the $p \times p$ dimensional covariance matrix of the zero-mean dataset.
- Calculate the eigenvectors and eigenvalues of this covariance matrix.
- Determine the dimension q of the reduced dataset by looking at the largest eigenvalues. The quality of the reduction depends on how close the sum of the largest q eigenvalues is to the sum of all p eigenvalues.
- Create the $q \times p$ projection matrix E^T from the eigenvectors corresponding to the q largest eigenvalues, and reduce the dataset by multiplying it with this matrix.
- To obtain the corresponding p-dimensional datapoints multiply the new data with the transpose of the projection matrix. Notice that this corresponds to choosing F in (2) such that $F = E$. If q is well chosen these regenerated datapoints should be fairly similar to the original datapoints, thus capturing most of the information in the dataset. Remember to add the mean again when comparing with the original data instead of the zero-mean data.

1.2 Self-organizing Maps

The purpose of Self-organizing Maps (SOMs) is to recognize groups of similar input vectors in an unsupervised fashion. You can create a self organizing map with `newsom`. The neurons in the layer of a SOM are originally arranged according to a predefined topology. The function `gridtop`, `hextop`, or `randtop` can arrange the neurons in a grid, hexagonal, or random topology. At step t each neuron competes to respond to an input vector x. The neuron i^* that is closer to it according to the pre-specified distance function wins. Available distance functions in the Matlab Neural Network Toolbox are `dist`, `boxdist`, `linkdist`, and `mandist`. All the neurons in the neighborhood $N(i^*)$ of the neuron i^* are then updated according to the Kohonen rule. That is, for each index $i \in N(i^*)$ we have:

$$w^i_{(t)} = (1-\alpha)w^i_{(t-1)} + \alpha x \tag{3}$$

where α is a small scalar and $w^i_{(t)}$ denotes the vector of weights representing neuron i at step t. Hence when a vector x is presented, the weights of the winning neuron and its close neighbors move towards x. Consequently, after many presentations, neighboring neurons have learned vectors similar to each other. Notice that the rule to define the neighborhood of each neuron can also vary across steps. In fact, the neighborhood of neuron i is normally defined as a ball whose radius shrinks with increasing t.

2 Exercises

2.1 Redundancy and Random Data

The idea of this exercise is to implement the PCA algorithm in Matlab and apply this to different datasets. The above algorithm can be easily programmed in Matlab with the following functions:

- `cov(x)` calculates the average of the dataset x, subtracts it from each datapoint, and returns the covariance matrix of the result. For matrix input x, it assumes that each row is an observation, and each column is a variable.
- `[v,d]=eig(x)` returns the eigenvectors (v) and eigenvalues (d) of the square matrix x as square matrices.
- `diag(x)` returns the diagonal of the square matrix x as a column vector.
- `[v,d]=eigs(x,k)` returns the k largest eigenvalues and corresponding eigenvectors of the square matrix x.
- `transpose(x)` returns the transpose of the matrix x. The command x' has the same effect.
- `help` command will show the available documentation on command.

Generate a 50×500 matrix of Gaussian random numbers (`randn(50,500)`) and try to reduce dimensions with PCA (interpret this as 500 datapoints of dimension 50). Examine different reduced datasets for different dimensions. Try to reconstruct the original matrix. Estimate the error, e.g. by calculating the root mean square difference between the reconstructed and the original data (`sqrt(mean(mean((x-xhat).^2)))`).

Do the same using the data file `choles_all` (standard in Matlab, can be loaded with `load choles_all`). For the exercise, use only the p component, which is a 21×264 matrix. How does the reduction of random data compare to the reduction of highly correlated data?
Now experiment with the functions `mapstd(x)` and `processpca(x,maxfrac)` (try for example `maxfrac=0.001`), and see whether you can obtain similar results. Note that `mapstd` must be used before `processpca` for the normalization of the matrix and that you can use `processpca('reverse',z,PS)` to get the reconstructed dataset.

2.2 Principal Component Analysis on Handwritten Digits

Perform PCA on handwritten images of the digit 3 taken from the US Postal Service database. To access these images, load the Matlab data called `threes.mat` by typing `load threes -ascii`. This loads a 2 megabyte matrix called `threes`. Each line of this matrix is a single 16 by 16 image of a handwritten 3 that has been expanded out into a 256 long vector. You can look at the i-th image by typing the command `imagesc(reshape(threes(i,:),16,16),[0,1])`. To have a black-white picture use the command `colormap('gray')` first.

- Compute the mean 3 and display it. Take a look at the command mean for this.
- Compute the covariance matrix of the whole dataset of 3s (note that the Matlab function `cov` subtracts the mean automatically, subtracting it beforehand is not incorrect however). Compute the eigenvalues and eigenvectors of this covariance matrix. Plot the eigenvalues (`plot(diag(D))` where D is the diagonal matrix of eigenvalues).
- Compress the dataset by projecting it onto one, two, three, and four principal components. You can use the manual approach via `cov` and `eigs` or the build-in methods. To choose the number of components, you can play with the argument `maxfrac` of `processpca`. Now reconstruct the image from these compressions and plot some pictures of the four reconstructions.
- Write a function which compresses the entire dataset by projecting it onto k principal components, then reconstructs it and measures the reconstruction error. Note that by choosing how many eigenvectors we use to reconstruct the image we are fixing the number of components, and the quality of the reconstruction. Now call this function for values of k from 1 to 50 (here you probably want to use a loop) and plot the reconstruction error as a function of k.
- What should the reconstruction error be if $k = 256$? What is it if you actually try it? Why?
- Use the Matlab function `cumsum` to create a vector whose i-th element is the sum of all but the i largest eigenvalues for $i = 1 : 256$. Compare the first 50 elements of this vector to the vector of reconstruction error versus k. What do you notice?

The last question should have shown you a very interesting and important fact: the squared reconstruction error induced by not using a certain principal component is proportional to its eigenvalue. That is why if the eigenvalues fall off quickly then projecting onto the first few gives very small errors because the sum of the eigenvalues that we are not using is not very large.

2.3 Competitive learning with SOM's

- Use `SOM_concentric_cylinders` as a template for generating data uniformly distributed within simple shapes in the plane or in the 3-D space. Then as in SOM concentric cylinders initialize multiple SOMs playing with the different topology and distance functions. Check how the prototypes distribute in the feature space before training. Then execute training and observe how the competitive rule determines the final placement according to the data distribution.
- Load the Iris sample dataset available within Matlab and use SOM to perform clustering . You can get an idea by running the script `example_SOM_iris`. Try a few different values for the grid size, topology and number of epochs. You can evaluate the performance by looking at the Adjusted Rand Index (ARI) [1]. To see what is happening you might also take a look at the plots: `SOM Topology`, `SOM Sample Hits` and `SOM Weight Positions` on the `nntraintool`.

3 Report

Write a report of maximum 2 pages (text + figures) to discuss:

- Handwritten Digits PCA and reconstruction
- SOM applied to the cylinder and Iris datasets.

[1] `https://en.wikipedia.org/wiki/Rand_index`