

Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen^{1,2}

Department of Physics and Center for Computing in Science Education,
University of Oslo, Norway¹

Department of Physics and Astronomy and Facility for Rare Isotope Beams,
Michigan State University, East Lansing, Michigan, USA²

March 12, 2024

© 1999-2024, Morten Hjorth-Jensen. Released under CC Attribution-NonCommercial 4.0 license

Plans for the week March 11-15

1. Discussion of Autoencoders (AEs)
2. Links between Principal Component Analysis (PCA) and AE

Reading recommendations

1. Goodfellow et al chapter 14.
2. Rashcka et al. Their chapter 17 contains a brief introduction only.
3. Deep Learning Tutorial on AEs from Stanford University
4. Building AEs in Keras
5. Introduction to AEs in TensorFlow
6. Grosse, University of Toronto, Lecture on AEs
7. Bank et al on AEs
8. Baldi and Hornik, Neural networks and principal component analysis: Learning from examples without local minima, Neural Networks 2, 53 (1989)

Autoencoders: Overarching view

Autoencoders are artificial neural networks capable of learning efficient representations of the input data (these representations are called codings) without any supervision (i.e., the training set is unlabeled). These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction.

Autoencoders learn to encode the input data into a lower-dimensional representation, and then decode it back to the original data. The goal of autoencoders is to minimize the reconstruction error, which measures how well the output matches the input. Autoencoders can be seen as a way of learning the latent features or hidden structure of the data, and they can be used for data compression, denoising, anomaly detection, and generative modeling.

Powerful detectors

More importantly, autoencoders act as powerful feature detectors, and they can be used for unsupervised pretraining of deep neural networks.

Lastly, they are capable of randomly generating new data that looks very similar to the training data; this is called a generative model. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces. Surprisingly, autoencoders work by simply learning to copy their inputs to their outputs. This may sound like a trivial task, but we will see that constraining the network in various ways can make it rather difficult. For example, you can limit the size of the internal representation, or you can add noise to the inputs and train the network to recover the original inputs. These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data. In short, the codings are byproducts of the autoencoder's attempt to learn the identity function under some constraints.

First introduction of AEs

Autoencoders were first introduced by Rumelhart, Hinton, and Williams in 1986 with the goal of learning to reconstruct the input observations with the lowest error possible.

Why would one want to learn to reconstruct the input observations? If you have problems imagining what that means, think of having a dataset made of images. An autoencoder would be an algorithm that can give as output an image that is as similar as possible to the input one. You may be confused, as there is no apparent reason of doing so. To better understand why autoencoders are useful we need a more informative (although not yet unambiguous) definition.

An autoencoder is a type of algorithm with the primary purpose of learning an "informative" representation of the data that can be used for different applications (see [Bank, D., Koenigstein, N., and Giryas, R., Autoencoders](#)) by learning to reconstruct a set of input observations well enough.

Autoencoder structure

Autoencoders are neural networks where the outputs are its own inputs. They are split into an **encoder part** which maps the input \mathbf{x} via a function $f(\mathbf{x}, \mathbf{W})$ (this is the encoder part) to a **so-called code part** (or intermediate part) with the result \mathbf{h}

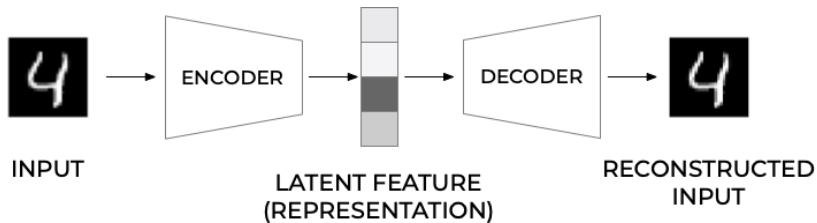
$$\mathbf{h} = f(\mathbf{x}, \mathbf{W}),$$

where \mathbf{W} are the weights to be determined. The **decoder** parts maps, via its own parameters (weights given by the matrix \mathbf{V} and its own biases) to the final output

$$\tilde{\mathbf{x}} = g(\mathbf{h}, \mathbf{V}).$$

The goal is to minimize the construction error.

Schematic image of an Autoencoder



More on the structure

In most typical architectures, the encoder and the decoder are neural networks since they can be easily trained with existing software libraries such as TensorFlow or PyTorch with back propagation.

In general, the encoder can be written as a function g that will depend on some parameters

$$h_i = g(x_i),$$

where $h_i \in \mathbb{R}^q$ (the latent feature representation) is the output of the encoder block where we evaluate it using the input x_i .

Decoder part

Note that we have $g : \mathbb{R}^n \rightarrow \mathbb{R}^q$. The decoder and the output of the network \tilde{x}_i can be written then as a second generic function of the latent features

$$\tilde{x}_i = f(h_i) = f(g(x_i)),$$

where $\tilde{x}_i \in \mathbb{R}^n$.

Training an autoencoder simply means finding the functions $g(\cdot)$ and $f(\cdot)$ that satisfy

$$\arg \min_{f,g} < [\Delta(x_i, f(g(x_i))) > .$$

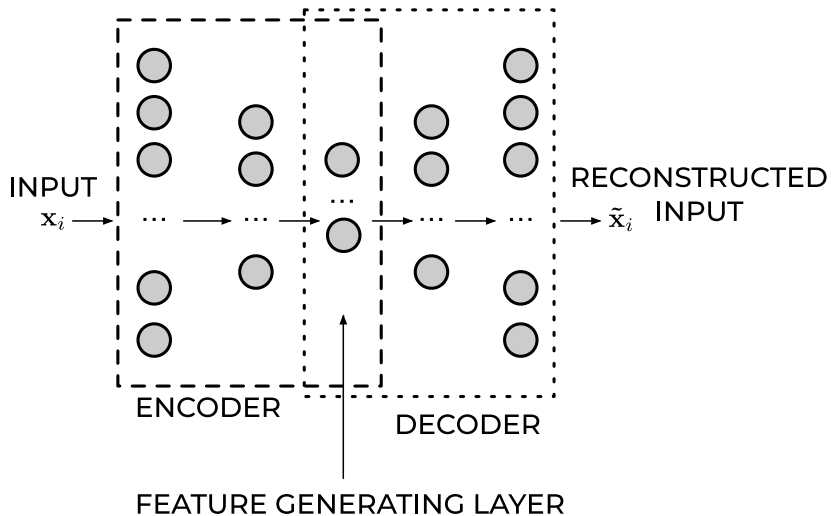
Typical AEs

The standard setup is done via a standard feed forward neural network (FFNN), or what is called a Feed Forward Autoencoder. A typical FFNN architecture has an odd number of layers and is symmetrical with respect to the middle layer.

Typically, the first layer has a number of neurons $n_1 = n$ which equals the size of the input observation x_i .

As we move toward the center of the network, the number of neurons in each layer drops in some measure. The middle layer usually has the smallest number of neurons. The fact that the number of neurons in this layer is smaller than the size of the input, is often called the **bottleneck**.

Feed Forward Autoencoder



Mirroring

In almost all practical applications, the layers after the middle one are a mirrored version of the layers before the middle one. For example, an autoencoder with three layers could have the following numbers of neurons:

$n_1 = 10$, $n_2 = 5$ and then $n_3 = n_1 = 10$ where the input dimension is equal to ten.

All the layers up to and including the middle one, make what is called the encoder, and all the layers from and including the middle one (up to the output) make what is called the decoder.

If the FFNN training is successful, the result will be a good approximation of the input $\tilde{x}_i \approx x_i$.

What is essential to notice is that the decoder can reconstruct the input by using only a much smaller number of features than the input observations initially have.

Output of middle layer

The output of the middle layer h_i are also called a **learned representation** of the input observation x_i .

The encoder can reduce the number of dimensions of the input observation and create a learned representation h_i) of the input that has a smaller dimension $q < n$.

This learned representation is enough for the decoder to reconstruct the input accurately (if the autoencoder training was successful as intended).

Activation Function of the Output Layer

In autoencoders based on neural networks, the output layer's activation function plays a particularly important role. The most used functions are ReLU and Sigmoid.

ReLU

The ReLU activation function can assume all values in the range $[0, \infty]$. As a reminder, its formula is

$$\text{ReLU}(x) = \max(0, x).$$

This choice is good when the input observations x_i assume a wide range of positive values. If the input x_i can assume negative values, the ReLU is, of course, a terrible choice, and the identity function is a much better choice.

The ReLU activation function for the output layer is well suited for cases when the input observations x_i assume a wide range of positive real values.

Sigmoid

The sigmoid function σ can assume all values in the range $[0, 1]$,

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

This activation function can only be used if the input observations x_i are all in the range $[0, 1]$ or if you have normalized them to be in that range. Consider as an example the MNIST dataset. Each value of the input observation x_i (one image) is the gray values of the pixels that can assume any value from 0 to 255. Normalizing the data by dividing the pixel values by 255 would make each observation (each image) have only pixel values between 0 and 1. In this case, the sigmoid would be a good choice for the output layer's activation function.

Cost/Loss Function

If an autoencoder is trying to solve a regression problem, the most common choice as a loss function is the Mean Square Error (MSE):

$$L_{\text{MSE}} = \text{MSE} = \frac{1}{M} \sum_{i=1}^M |x_i - \tilde{x}_i|^2$$

Binary Cross-Entropy

If the activation function of the output layer of the AE is a sigmoid function, thus limiting neuron outputs to be between 0 and 1, and the input features are normalized to be between 0 and 1 we can use as loss function the binary cross-entropy. This cost/loss function is typically used in classification problems, but it works well for autoencoders. The formula for it is

$$L_{\text{CE}} = -\frac{1}{M} \sum_{i=1}^M \sum_{j=1}^n [x_{j,i} \log \tilde{x}_{j,i} + (1 - x_{j,i}) \log(1 - \tilde{x}_{j,i})]$$

Reconstruction Error

The reconstruction error (RE) is a metric that gives you an indication of how good (or bad) the autoencoder was able to reconstruct the input observation x_i . The most typical RE used is the MSE

$$\text{RE} \equiv \text{MSE} = \frac{1}{M} \sum_{i=1}^M |x_i - \tilde{x}_i|^2$$

Dimensionality reduction and links with Principal component analysis

The hope is that the training of the autoencoder can unravel some useful properties of the function f . They are often trained with only single-layer neural networks (although deep networks can improve the training) and are essentially given by feed forward neural networks.

Linear functions

If the function f and g are given by a linear dependence on the weight matrices \mathbf{W} and \mathbf{V} , we can show that for a regression case, by minimizing the mean squared error between \mathbf{x} and $\tilde{\mathbf{x}}$, the autoencoder learns the same subspace as the standard principal component analysis (PCA).

In order to see this, we define then

$$\mathbf{h} = f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x},$$

and

$$\tilde{\mathbf{x}} = g(\mathbf{h}, \mathbf{V}) = \mathbf{V}\mathbf{h} = \mathbf{V}\mathbf{W}\mathbf{x}.$$

AE mean-squared error

With the above linear dependence we can in turn define our optimization problem in terms of the optimization of the mean-squared error, that is we wish to optimize

$$\min_{\mathbf{w}, \mathbf{v} \in \mathbb{R}} \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \tilde{x}_i)^2 = \frac{1}{n} \|\mathbf{x} - \mathbf{VWx}\|_2^2,$$

where we have used the definition of a norm-2 vector, that is

$$\|\mathbf{x}\|_2 = \sqrt{\sum_i x_i^2}.$$

Dimensionality reduction

This is equivalent to our functions learning the same subspace as the PCA method. This means that we can interpret AEs as a dimensionality reduction method. To see this, we need to remind ourselves about the PCA method.

What is the Principal Component Analysis (PCA)?

PCA is a linear transformation that finds the directions of maximum variance in the data, and projects the data onto a lower-dimensional space. These directions are called principal components, and they are orthogonal to each other. PCA can be seen as a way of compressing the data by discarding the components that have low variance and retain the most important ones. PCA can be applied to both supervised and unsupervised learning problems, and it is often used for data visualization, feature extraction, and noise reduction.

A linear autoencoder can be shown to be equal to the PCA. In this lectures we will try to expose these ideas.

Basic ideas of the PCA

The principal component analysis deals with the problem of fitting a low-dimensional affine subspace S of dimension d much smaller than the total dimension D of the problem at hand (our data set). Mathematically it can be formulated as a statistical problem or a geometric problem. In our discussion of the theorem for the classical PCA, we will stay with a statistical approach. Historically, the PCA was first formulated in a statistical setting in order to estimate the principal component of a multivariate random variable.

Ingredients of the PCA

We have a data set defined by a design/feature matrix \mathbf{X} (see below for its definition)

1. Each data point is determined by p extrinsic (measurement) variables
2. We may want to ask the following question: Are there fewer intrinsic variables (say $d \ll p$) that still approximately describe the data?
3. If so, these intrinsic variables may tell us something important and finding these intrinsic variables is what dimension reduction methods do.

A good read is for example [Vidal, Ma and Sastry](#).

Introducing the Covariance and Correlation functions

Before we discuss the PCA theorem, we need to remind ourselves about the definition of the covariance and the correlation function.

These are quantities

Suppose we have defined two vectors \hat{x} and \hat{y} with n elements each. The covariance matrix \mathbf{C} is defined as

$$\mathbf{C}[\mathbf{x}, \mathbf{y}] = \begin{bmatrix} \text{cov}[\mathbf{x}, \mathbf{x}] & \text{cov}[\mathbf{x}, \mathbf{y}] \\ \text{cov}[\mathbf{y}, \mathbf{x}] & \text{cov}[\mathbf{y}, \mathbf{y}] \end{bmatrix},$$

where for example

$$\text{cov}[\mathbf{x}, \mathbf{y}] = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})(y_i - \bar{y}).$$

Covariance matrix

With this definition and recalling that the variance is defined as

$$\text{var}[\mathbf{x}] = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \bar{x})^2,$$

we can rewrite the covariance matrix as

$$\mathbf{C}[\mathbf{x}, \mathbf{y}] = \begin{bmatrix} \text{var}[\mathbf{x}] & \text{cov}[\mathbf{x}, \mathbf{y}] \\ \text{cov}[\mathbf{x}, \mathbf{y}] & \text{var}[\mathbf{y}] \end{bmatrix}.$$

More on the covariance

The covariance takes values between zero and infinity and may thus lead to problems with loss of numerical precision for particularly large values. It is common to scale the covariance matrix by introducing instead the correlation matrix defined via the so-called correlation function

$$\text{corr}[\mathbf{x}, \mathbf{y}] = \frac{\text{cov}[\mathbf{x}, \mathbf{y}]}{\sqrt{\text{var}[\mathbf{x}]\text{var}[\mathbf{y}]}}.$$

The correlation function is then given by values $\text{corr}[\mathbf{x}, \mathbf{y}] \in [-1, 1]$. This avoids eventual problems with too large values. We can then define the correlation matrix for the two vectors \mathbf{x} and \mathbf{y} as

$$\mathbf{K}[\mathbf{x}, \mathbf{y}] = \begin{bmatrix} 1 & \text{corr}[\mathbf{x}, \mathbf{y}] \\ \text{corr}[\mathbf{y}, \mathbf{x}] & 1 \end{bmatrix},$$

In the above example this is the function we constructed using **pandas**.

Reminding ourselves about Linear Regression

In our derivation of the various regression algorithms like **Ordinary Least Squares** or **Ridge regression** we defined the design/feature matrix \mathbf{X} as

$$\mathbf{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & \dots & \dots x_{0,p-1} \\ x_{1,0} & x_{1,1} & x_{1,2} & \dots & \dots x_{1,p-1} \\ x_{2,0} & x_{2,1} & x_{2,2} & \dots & \dots x_{2,p-1} \\ \dots & \dots & \dots & \dots & \dots \\ x_{n-2,0} & x_{n-2,1} & x_{n-2,2} & \dots & \dots x_{n-2,p-1} \\ x_{n-1,0} & x_{n-1,1} & x_{n-1,2} & \dots & \dots x_{n-1,p-1} \end{bmatrix},$$

with $\mathbf{X} \in \mathbb{R}^{n \times p}$, with the predictors/features p referring to the column numbers and the entries n being the row elements.

Rewriting the matrix \mathbf{X}

We can rewrite the design/feature matrix in terms of its column vectors as

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_0 & \mathbf{x}_1 & \mathbf{x}_2 & \dots & \dots & \mathbf{x}_{p-1} \end{bmatrix},$$

with a given vector

$$\mathbf{x}_i^T = \begin{bmatrix} x_{0,i} & x_{1,i} & x_{2,i} & \dots & \dots & x_{n-1,i} \end{bmatrix}.$$

Simple Example

With these definitions, we can now rewrite our 2×2 correlation/covariance matrix in terms of a more general design/feature matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$. This leads to a $p \times p$ covariance matrix for the vectors \mathbf{x}_i with $i = 0, 1, \dots, p-1$

$$\mathbf{C}[\mathbf{x}] = \begin{bmatrix} \text{var}[\mathbf{x}_0] & \text{cov}[\mathbf{x}_0, \mathbf{x}_1] & \text{cov}[\mathbf{x}_0, \mathbf{x}_2] & \dots & \dots & \text{cov}[\mathbf{x}_0, \mathbf{x}_{p-1}] \\ \text{cov}[\mathbf{x}_1, \mathbf{x}_0] & \text{var}[\mathbf{x}_1] & \text{cov}[\mathbf{x}_1, \mathbf{x}_2] & \dots & \dots & \text{cov}[\mathbf{x}_1, \mathbf{x}_{p-1}] \\ \text{cov}[\mathbf{x}_2, \mathbf{x}_0] & \text{cov}[\mathbf{x}_2, \mathbf{x}_1] & \text{var}[\mathbf{x}_2] & \dots & \dots & \text{cov}[\mathbf{x}_2, \mathbf{x}_{p-1}] \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \text{cov}[\mathbf{x}_{p-1}, \mathbf{x}_0] & \text{cov}[\mathbf{x}_{p-1}, \mathbf{x}_1] & \text{cov}[\mathbf{x}_{p-1}, \mathbf{x}_2] & \dots & \dots & \text{var}[\mathbf{x}_{p-1}] \end{bmatrix}$$

The Correlation Matrix

The correlation matrix

$$\mathbf{K}[\mathbf{x}] = \begin{bmatrix} 1 & \text{corr}[\mathbf{x}_0, \mathbf{x}_1] & \text{corr}[\mathbf{x}_0, \mathbf{x}_2] & \dots & \dots & \text{corr}[\mathbf{x}_0, \mathbf{x}_{p-1}] \\ \text{corr}[\mathbf{x}_1, \mathbf{x}_0] & 1 & \text{corr}[\mathbf{x}_1, \mathbf{x}_2] & \dots & \dots & \text{corr}[\mathbf{x}_1, \mathbf{x}_{p-1}] \\ \text{corr}[\mathbf{x}_2, \mathbf{x}_0] & \text{corr}[\mathbf{x}_2, \mathbf{x}_1] & 1 & \dots & \dots & \text{corr}[\mathbf{x}_2, \mathbf{x}_{p-1}] \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \text{corr}[\mathbf{x}_{p-1}, \mathbf{x}_0] & \text{corr}[\mathbf{x}_{p-1}, \mathbf{x}_1] & \text{corr}[\mathbf{x}_{p-1}, \mathbf{x}_2] & \dots & \dots & 1 \end{bmatrix}$$

Numpy Functionality

The Numpy function **np.cov** calculates the covariance elements using the factor $1/(n - 1)$ instead of $1/n$ since it assumes we do not have the exact mean values. The following simple function uses the **np.vstack** function which takes each vector of dimension $1 \times n$ and produces a $2 \times n$ matrix **W**

$$\mathbf{W}^T = \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \dots & \dots \\ x_{n-2} & y_{n-2} \\ x_{n-1} & y_{n-1} \end{bmatrix},$$

which in turn is converted into the 2×2 covariance matrix **C** via the Numpy function **np.cov()**. We note that we can also calculate the mean value of each set of samples **x** etc using the Numpy function **np.mean(x)**. We can also extract the eigenvalues of the covariance matrix through the **np.linalg.eig()** function.

```
# Importing various packages
```

```
import numpy as np
```

```
n = 100
```

Correlation Matrix again

The previous example can be converted into the correlation matrix by simply scaling the matrix elements with the variances. We should also subtract the mean values for each column. This leads to the following code which sets up the correlations matrix for the previous example in a more brute force way. Here we scale the mean values for each column of the design matrix, calculate the relevant mean values and variances and then finally set up the 2×2 correlation matrix (since we have only two vectors).

```
import numpy as np
n = 100
# define two vectors
x = np.random.random(size=n)
y = 4+3*x+np.random.normal(size=n)
#scaling the x and y vectors
x = x - np.mean(x)
y = y - np.mean(y)
variance_x = np.sum(x*x)/n
variance_y = np.sum(y*y)/n
print(variance_x)
print(variance_y)
cov_xy = np.sum(x*y)/n
cov_xx = np.sum(x*x)/n
cov_yy = np.sum(y*y)/n
C = np.zeros((2,2))
```

Using Pandas

We show here how we can set up the correlation matrix using **pandas**, as done in this simple code

```
import numpy as np
import pandas as pd
n = 10
x = np.random.normal(size=n)
x = x - np.mean(x)
y = 4+3*x+np.random.normal(size=n)
y = y - np.mean(y)
X = (np.vstack((x, y))).T
print(X)
Xpd = pd.DataFrame(X)
print(Xpd)
correlation_matrix = Xpd.corr()
print(correlation_matrix)
```

Links with the Design Matrix

We can rewrite the covariance matrix in a more compact form in terms of the design/feature matrix \mathbf{X} as

$$\mathbf{C}[\mathbf{x}] = \frac{1}{n} \mathbf{X}^T \mathbf{X} = \mathbb{E}[\mathbf{X}^T \mathbf{X}].$$

To see this let us simply look at a design matrix $\mathbf{X} \in \mathbb{R}^{2 \times 2}$

$$\mathbf{X} = \begin{bmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_0 & \mathbf{x}_1 \end{bmatrix}.$$

Computing the Expectation Values

If we then compute the expectation value

$$\mathbb{E}[\mathbf{X}^T \mathbf{X}] = \frac{1}{n} \mathbf{X}^T \mathbf{X} = \begin{bmatrix} x_{00}^2 + x_{01}^2 & x_{00}x_{10} + x_{01}x_{11} \\ x_{10}x_{00} + x_{11}x_{01} & x_{10}^2 + x_{11}^2 \end{bmatrix},$$

which is just

$$\mathbf{C}[\mathbf{x}_0, \mathbf{x}_1] = \mathbf{C}[\mathbf{x}] = \begin{bmatrix} \text{var}[\mathbf{x}_0] & \text{cov}[\mathbf{x}_0, \mathbf{x}_1] \\ \text{cov}[\mathbf{x}_1, \mathbf{x}_0] & \text{var}[\mathbf{x}_1] \end{bmatrix},$$

where we wrote

$$\mathbf{C}[\mathbf{x}_0, \mathbf{x}_1] = \mathbf{C}[\mathbf{x}]$$

to indicate that this is the covariance of the vectors \mathbf{x} of the design/feature matrix \mathbf{X} .

It is easy to generalize this to a matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$.

Towards the PCA theorem

We have that the covariance matrix (the correlation matrix involves a simple rescaling) is given as

$$\mathbf{C}[\mathbf{x}] = \frac{1}{n} \mathbf{X}^T \mathbf{X} = \mathbb{E}[\mathbf{X}^T \mathbf{X}].$$

Let us now assume that we can perform a series of orthogonal transformations where we employ some orthogonal matrices \mathbf{S} . These matrices are defined as $\mathbf{S} \in \mathbb{R}^{p \times p}$ and obey the orthogonality requirements $\mathbf{S}\mathbf{S}^T = \mathbf{S}^T\mathbf{S} = \mathbf{I}$. The matrix can be written out in terms of the column vectors \mathbf{s}_i as $\mathbf{S} = [\mathbf{s}_0, \mathbf{s}_1, \dots, \mathbf{s}_{p-1}]$ and $\mathbf{s}_i \in \mathbb{R}^p$.

More details

Assume also that there is a transformation $\mathbf{S}^T \mathbf{C}[\mathbf{x}] \mathbf{S} = \mathbf{C}[\mathbf{y}]$ such that the new matrix $\mathbf{C}[\mathbf{y}]$ is diagonal with elements $[\lambda_0, \lambda_1, \lambda_2, \dots, \lambda_{p-1}]$.

That is we have

$$\mathbf{C}[\mathbf{y}] = \mathbb{E}[\mathbf{S}^T \mathbf{X}^T \mathbf{X} \mathbf{T} \mathbf{S}] = \mathbf{S}^T \mathbf{C}[\mathbf{x}] \mathbf{S},$$

since the matrix \mathbf{S} is not a data dependent matrix. Multiplying with \mathbf{S} from the left we have

$$\mathbf{S} \mathbf{C}[\mathbf{y}] = \mathbf{C}[\mathbf{x}] \mathbf{S},$$

and since $\mathbf{C}[\mathbf{y}]$ is diagonal we have for a given eigenvalue i of the covariance matrix that

$$\mathbf{S}_i \lambda_i = \mathbf{C}[\mathbf{x}] \mathbf{S}_i.$$

More on the PCA Theorem

In the derivation of the PCA theorem we will assume that the eigenvalues are ordered in descending order, that is

$$\lambda_0 > \lambda_1 > \cdots > \lambda_{p-1}.$$

The eigenvalues tell us then how much we need to stretch the corresponding eigenvectors. Dimensions with large eigenvalues have thus large variations (large variance) and define therefore useful dimensions. The data points are more spread out in the direction of these eigenvectors. Smaller eigenvalues mean on the other hand that the corresponding eigenvectors are shrunk accordingly and the data points are tightly bunched together and there is not much variation in these specific directions. Hopefully then we could leave it out dimensions where the eigenvalues are very small. If p is very large, we could then aim at reducing p to $l \ll p$ and handle only l features/predictors.

The Algorithm before the Theorem

Here's how we would proceed in setting up the algorithm for the PCA, see also discussion below here.

- ▶ Set up the datapoints for the design/feature matrix \mathbf{X} with $\mathbf{X} \in \mathbb{R}^{n \times p}$, with the predictors/features p referring to the column numbers and the entries n being the row elements.

$$\mathbf{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & \dots & \dots x_{0,p-1} \\ x_{1,0} & x_{1,1} & x_{1,2} & \dots & \dots x_{1,p-1} \\ x_{2,0} & x_{2,1} & x_{2,2} & \dots & \dots x_{2,p-1} \\ \dots & \dots & \dots & \dots & \dots \\ x_{n-2,0} & x_{n-2,1} & x_{n-2,2} & \dots & \dots x_{n-2,p-1} \\ x_{n-1,0} & x_{n-1,1} & x_{n-1,2} & \dots & \dots x_{n-1,p-1} \end{bmatrix},$$

- ▶ Center the data by subtracting the mean value for each column. This leads to a new matrix $\mathbf{X} \rightarrow \bar{\mathbf{X}}$.
- ▶ Compute then the covariance/correlation matrix $\mathbb{E}[\bar{\mathbf{X}}^T \bar{\mathbf{X}}]$.
- ▶ Find the eigenpairs of \mathbf{C} with eigenvalues $[\lambda_0, \lambda_1, \dots, \lambda_{p-1}]$ and eigenvectors $[\mathbf{s}_0, \mathbf{s}_1, \dots, \mathbf{s}_{p-1}]$.
- ▶ Order the eigenvalue (and the eigenvectors accordingly) in order of decreasing eigenvalues.

Writing our own PCA code

We will use a simple example first with two-dimensional data drawn from a multivariate normal distribution with the following mean and covariance matrix (we have fixed these quantities but will play around with them below):

$$\mu = (-1, 2) \quad \Sigma = \begin{bmatrix} 4 & 2 \\ 2 & 2 \end{bmatrix}$$

Note that the mean refers to each column of data. We will generate $n = 10000$ points $X = \{x_1, \dots, x_N\}$ from this distribution, and store them in the 1000×2 matrix \mathbf{X} . This is our design matrix where we have forced the covariance and mean values to take specific values.

Implementing it

The following Python code aids in setting up the data and writing out the design matrix. Note that the function **multivariate** returns also the covariance discussed above and that it is defined by dividing by $n - 1$ instead of n .

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import display
n = 10000
mean = (-1, 2)
cov = [[4, 2], [2, 2]]
X = np.random.multivariate_normal(mean, cov, n)
```

Now we are going to implement the PCA algorithm. We will break it down into various substeps.

First Step

The first step of PCA is to compute the sample mean of the data and use it to center the data. Recall that the sample mean is

$$\mu_n = \frac{1}{n} \sum_{i=1}^n x_i$$

and the mean-centered data $\bar{X} = \{\bar{x}_1, \dots, \bar{x}_n\}$ takes the form

$$\bar{x}_i = x_i - \mu_n.$$

When you are done with these steps, print out μ_n to verify it is close to μ and plot your mean centered data to verify it is centered at the origin! The following code elements perform these operations using **pandas** or using our own functionality for doing so. The latter, using **numpy** is rather simple through the **mean()** function.

```
df = pd.DataFrame(X)
# Pandas does the centering for us
df = df - df.mean()
# we center it ourselves
X_centered = X - X.mean(axis=0)
```

Scaling

Alternatively, we could use the functions we discussed earlier for scaling the data set. That is, we could have used the **StandardScaler** function in **Scikit-Learn**, a function which ensures that for each feature/predictor we study the mean value is zero and the variance is one (every column in the design/feature matrix). You would then not get the same results, since we divide by the variance. The diagonal covariance matrix elements will then be one, while the non-diagonal ones need to be divided by $2\sqrt{2}$ for our specific case.

Centered Data

Now we are going to use the mean centered data to compute the sample covariance of the data by using the following equation

$$\Sigma_n = \frac{1}{n-1} \sum_{i=1}^n \bar{x}_i^T \bar{x}_i = \frac{1}{n-1} \sum_{i=1}^n (x_i - \mu_n)^T (x_i - \mu_n)$$

where the data points $x_i \in \mathbb{R}^p$ (here in this example $p = 2$) are column vectors and x^T is the transpose of x . We can write our own code or simply use either the functionality of **numpy** or that of **pandas**, as follows

```
print(df.cov())  
print(np.cov(X_centered.T))
```

Note that the way we define the covariance matrix here has a factor $n - 1$ instead of n . This is included in the `cov()` function by **numpy** and **pandas**. Our own code here is not very elegant and asks for obvious improvements. It is tailored to this specific 2×2 covariance matrix.

```
# extract the relevant columns from the centered design matrix of dim  
x = X_centered[:,0]  
y = X_centered[:,1]  
Cov = np.zeros((2,2))
```


Exploring

Depending on the number of points n , we will get results that are close to the covariance values defined above. The plot shows how the data are clustered around a line with slope close to one. Is this expected? Try to change the covariance and the mean values. For example, try to make the variance of the first element much larger than that of the second diagonal element. Try also to shrink the covariance (the non-diagonal elements) and see how the data points are distributed.

Diagonalize the sample covariance matrix to obtain the principal components

Now we are ready to solve for the principal components! To do so we diagonalize the sample covariance matrix Σ . We can use the function `np.linalg.eig` to do so. It will return the eigenvalues and eigenvectors of Σ . Once we have these we can perform the following tasks:

- ▶ We compute the percentage of the total variance captured by the first principal component
- ▶ We plot the mean centered data and lines along the first and second principal components
- ▶ Then we project the mean centered data onto the first and second principal components, and plot the projected data.
- ▶ Finally, we approximate the data as

$$x_i \approx \tilde{x}_i = \mu_n + \langle x_i, v_0 \rangle v_0$$

where v_0 is the first principal component.

Collecting all Steps

Collecting all these steps we can write our own PCA function and compare this with the functionality included in **Scikit-Learn**.

The code here outlines some of the elements we could include in the analysis. Feel free to extend upon this in order to address the above questions.

```
# diagonalize and obtain eigenvalues, not necessarily sorted
EigValues, EigVectors = np.linalg.eig(Cov)
# sort eigenvectors and eigenvalues
#permute = EigValues.argsort()
#EigValues = EigValues[permute]
#EigVectors = EigVectors[:,permute]
print("Eigenvalues of Covariance matrix")
for i in range(2):
    print(EigValues[i])
FirstEigvector = EigVectors[:,0]
SecondEigvector = EigVectors[:,1]
print("First eigenvector")
print(FirstEigvector)
print("Second eigenvector")
print(SecondEigvector)
#thereafter we do a PCA with Scikit-learn
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2Dsl = pca.fit_transform(X)
print("Eigenvector of largest eigenvalue")
print(pca.components_[:,0])
```

Classical PCA Theorem

We assume now that we have a design matrix \mathbf{X} which has been centered as discussed above. For the sake of simplicity we skip the overline symbol. The matrix is defined in terms of the various column vectors $[\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{p-1}]$ each with dimension $\mathbf{x} \in \mathbb{R}^n$. The PCA theorem states that minimizing the above reconstruction error corresponds to setting $\mathbf{W} = \mathbf{S}$, the orthogonal matrix which diagonalizes the empirical covariance(correlation) matrix. The optimal low-dimensional encoding of the data is then given by a set of vectors \mathbf{z}_i with at most l vectors, with $l \ll p$, defined by the orthogonal projection of the data onto the columns spanned by the eigenvectors of the covariance(correlations matrix).

The PCA Theorem

To show the PCA theorem let us start with the assumption that there is one vector \mathbf{s}_0 which corresponds to a solution which minimized the reconstruction error J . This is an orthogonal vector. It means that we now approximate the reconstruction error in terms of \mathbf{w}_0 and \mathbf{z}_0 as

We are almost there, we have obtained a relation between minimizing the reconstruction error and the variance and the covariance matrix. Minimizing the error is equivalent to maximizing the variance of the projected data.

We could trivially maximize the variance of the projection (and thereby minimize the error in the reconstruction function) by letting the norm-2 of \mathbf{w}_0 go to infinity. However, this norm since we want the matrix \mathbf{W} to be an orthogonal matrix, is constrained by $\|\mathbf{w}_0\|_2^2 = 1$. Imposing this condition via a Lagrange multiplier we can then in turn maximize

$$J(\mathbf{w}_0) = \mathbf{w}_0^T \mathbf{C}[\mathbf{x}] \mathbf{w}_0 + \lambda_0(1 - \mathbf{w}_0^T \mathbf{w}_0).$$

Taking the derivative with respect to \mathbf{w}_0 we obtain

Geometric Interpretation and link with Singular Value Decomposition

For a detailed demonstration of the geometric interpretation, see [Vidal, Ma and Sastry, section 2.1.2](#).

Principal Component Analysis (PCA) is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it. The following Python code uses NumPy's `svd()` function to obtain all the principal components of the training set, then extracts the first two principal components. First we center the data using either **pandas** or our own code

```
import numpy as np
import pandas as pd
from IPython.display import display
np.random.seed(100)
# setting up a 10 x 5 vanilla matrix
rows = 10
cols = 5
X = np.random.randn(rows,cols)
df = pd.DataFrame(X)
# Pandas does the centering for us
df = df - df.mean()
display(df)
```

PCA and scikit-learn

Scikit-Learn's PCA class implements PCA using SVD decomposition just like we did before. The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
#thereafter we do a PCA with Scikit-learn  
from sklearn.decomposition import PCA  
pca = PCA(n_components = 2)  
X2D = pca.fit_transform(X)  
print(X2D)
```

After fitting the PCA transformer to the dataset, you can access the principal components using the components variable (note that it contains the PCs as horizontal vectors, so, for example, the first principal component is equal to

```
pca.components_.T[:, 0]
```

Another very useful piece of information is the explained variance ratio of each principal component, available via the *explained_variance_ratio* variable. It indicates the proportion of the dataset's variance that lies along the axis of each principal component.

Example of Cancer Data

We can now repeat the above but applied to real data, in this case the Wisconsin breast cancer data. Here we compute performance scores on the training data using logistic regression.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=0)

logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("Train set accuracy from Logistic Regression: {:.2f}".format(logreg.score(X_train, y_train)))
# We scale the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Then perform again a log reg fit
logreg.fit(X_train_scaled, y_train)
print("Train set accuracy scaled data: {:.2f}".format(logreg.score(X_train_scaled, y_train)))
# thereafter we do a PCA with Scikit-learn
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
```


Incremental PCA

One problem with the preceding implementation of PCA is that it requires the whole training set to fit in memory in order for the SVD algorithm to run. Fortunately, Incremental PCA (IPCA) algorithms have been developed: you can split the training set into mini-batches and feed an IPCA algorithm one minibatch at a time. This is useful for large training sets, and also to apply PCA online (i.e., on the fly, as new instances arrive).

Randomized PCA. Scikit-Learn offers yet another option to perform PCA, called Randomized PCA. This is a stochastic algorithm that quickly finds an approximation of the first d principal components. Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$, so it is dramatically faster than the previous algorithms when d is much smaller than n .

Kernel PCA. The kernel trick is a mathematical technique that implicitly maps instances into a very high-dimensional space (called the feature space), enabling nonlinear classification and regression with Support Vector Machines. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the original space. It turns

Other techniques

There are many other dimensionality reduction techniques, several of which are available in Scikit-Learn.

Here are some of the most popular:

- ▶ **Multidimensional Scaling (MDS)** reduces dimensionality while trying to preserve the distances between the instances.
- ▶ **Isomap** creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the geodesic distances between the instances.
- ▶ **t-Distributed Stochastic Neighbor Embedding (t-SNE)** reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space (e.g., to visualize the MNIST images in 2D).
- ▶ **Linear Discriminant Analysis (LDA)** is actually a classification algorithm, but during training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data. The benefit is that the projection will keep classes as far apart as