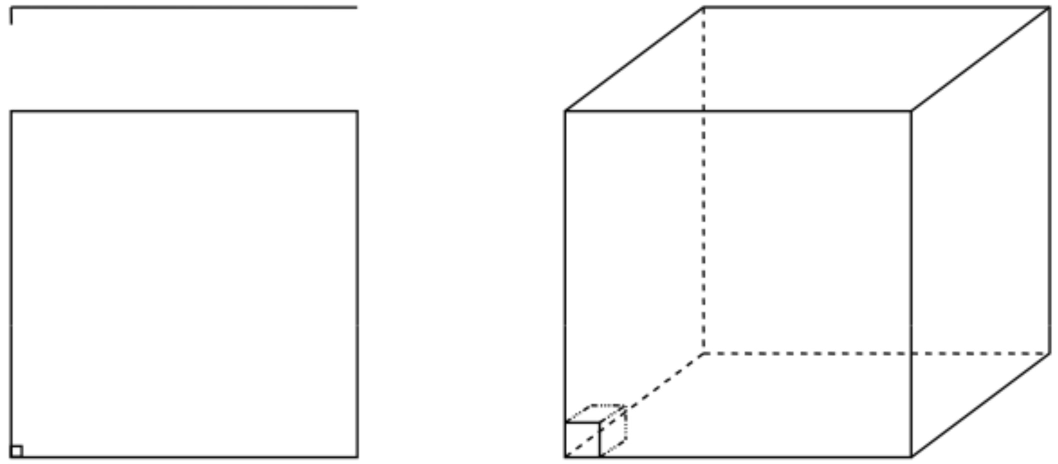
Lecture: Face Recognition and Feature Reduction

Juan Carlos Niebles and Ranjay Krishna
Stanford Vision and Learning Lab

Recap - Curse of dimensionality

- Assume 5000 points uniformly distributed in the unit hypercube and we want to apply 5-NN. Suppose our query point is at the origin.
  - In 1-dimension, we must go a distance of 5/5000=0.001 on the average to capture 5 nearest neighbors.
  - In 2 dimensions, we must go $\sqrt{0.001}$ to get a square that contains 0.001 of the volume.
  - In d dimensions, we must go $(0.001)^{1/d}$



What we will learn today

- Singular value decomposition
- Principal Component Analysis (PCA)
- Image compression

What we will learn today

- Singular value decomposition
- Principal Component Analysis (PCA)
- Image compression

Singular Value Decomposition (SVD)

- There are several computer algorithms that can "factorize" a matrix, representing it as the product of some other matrices
- The most useful of these is the Singular Value Decomposition.
- Represents any matrix **A** as a product of three matrices: $\mathbf{U \Sigma V^T}$
- Python command:
  - [U,S,V]= numpy.linalg.svd(A)

Singular Value Decomposition (SVD)

$$\mathbf{U\Sigma V^T = A}$$

- Where **U** and **V** are rotation matrices, and **Σ** is a scaling matrix. For example:

$$\overset{U}{\begin{bmatrix} -.40 & .916 \\ .916 & .40 \end{bmatrix}} \times \overset{\Sigma}{\begin{bmatrix} 5.39 & 0 \\ 0 & 3.154 \end{bmatrix}} \times \overset{V^T}{\begin{bmatrix} -.05 & .999 \\ .999 & .05 \end{bmatrix}} = \overset{A}{\begin{bmatrix} 3 & -2 \\ 1 & 5 \end{bmatrix}}$$

Singular Value Decomposition (SVD)

- Beyond 2x2 matrices:
  - In general, if **A** is *m* x *n*, then **U** will be *m* x *m*, **Σ** will be *m* x *n*, and $\mathbf{V}^T$ will be *n* x *n*.
  - (Note the dimensions work out to produce *m* x *n* after multiplication)

$$\overset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \overset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \overset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \overset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

Singular Value Decomposition (SVD)

- **U** and **V** are always rotation matrices.
  - Geometric rotation may not be an applicable concept, depending on the matrix. So we call them "unitary" matrices – each column is a unit vector.
- **Σ** is a diagonal matrix
  - The number of nonzero entries = rank of **A**
  - The algorithm always sorts the entries high to low

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

SVD Applications

- We’ve discussed SVD in terms of geometric transformation matrices
- But SVD of an image matrix can also be very useful
- To understand this, we’ll look at a less geometric interpretation of what SVD is doing

SVD Applications

$$\overset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \overset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \overset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \overset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

- Look at how the multiplication works out, left to right:
- Column 1 of **U** gets scaled by the first value from **Σ**.

$$\overset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \overset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \overset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

- The resulting vector gets scaled by row 1 of **V**^{T} to produce a contribution to the columns of **A**

SVD Applications

decomposition USigma

$$\begin{array}{c} U\Sigma \\ \begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix} \end{array} \times \begin{array}{c} V^T \\ \begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix} \end{array} \quad \begin{array}{c} A_{partial} \\ \begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix} \end{array}$$

$$+ \begin{array}{c} U\Sigma \\ \begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix} \end{array} \times \begin{array}{c} V^T \\ \begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix} \end{array} \quad \begin{array}{c} A_{partial} \\ \begin{bmatrix} -.6 & -.1 & .4 \\ .2 & 0 & -.2 \end{bmatrix} \end{array}$$

$$= \qquad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- Each product of (*column i of **U***)·(*value i from **Σ***)·(*row i of **V**^{T}*) produces a component of the final **A**.

SVD Applications

$$\underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \underset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

$$\underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

$$\underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \underset{A_{partial}}{\begin{bmatrix} -.6 & -.1 & .4 \\ .2 & 0 & -.2 \end{bmatrix}}$$

- We’re building **A** as a linear combination of the columns of ***U***
- Using all columns of ***U***, we’ll rebuild the original matrix perfectly
- But, in real-world data, often we can just use the first few columns of ***U*** and we’ll get something close (e.g. the first ***A*$_{partial}$**, above)

SVD Applications

$$\overset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \overset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \overset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

$$\overset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

$$\overset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \overset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \overset{A_{partial}}{\begin{bmatrix} -.6 & -.1 & .4 \\ .2 & 0 & -.2 \end{bmatrix}}$$

- We can call those first few columns of ***U*** the *Principal Components* of the data
- They show the major patterns that can be added to produce the columns of the original matrix
- The rows of **V**$^{\mathbf{T}}$ show how the *principal components* are mixed to produce the columns of the matrix

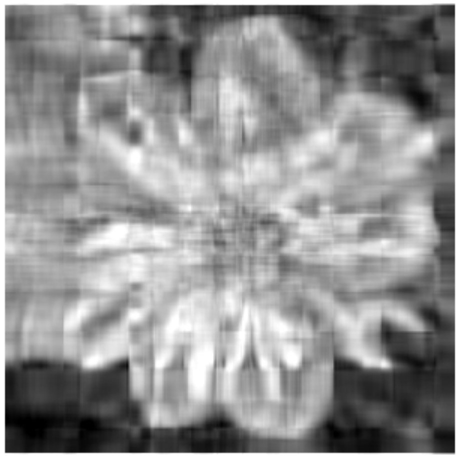
SVD Applications

$$\overset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \overset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \overset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \overset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

We can look at **Σ** to see that the first column has a large effect

while the second column has a much smaller effect in this example

SVD Applications



- For this image, using **only the first 10** of 300 principal components produces a recognizable reconstruction U matrix
- So, SVD can be used for image compression

SVD for symmetric matrices

- If A is a symmetric matrix, it can be decomposed as the following:

$$A = \Phi \Sigma \Phi^T$$

- Compared to a traditional SVD decomposition, ~~U = V^T~~ and is an orthogonal matrix.

U = V

Principal Component Analysis

$$\overset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \overset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \overset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

- Remember, columns of ***U*** are the *Principal Components* of the data: the major patterns that can be added to produce the columns of the original matrix
- One use of this is to construct a matrix where each column is a separate data sample
- Run SVD on that matrix, and look at the first few columns of ***U*** to see patterns that are common among the columns
- This is called *Principal Component Analysis* (or PCA) of the data samples

Principal Component Analysis

$$\overset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \overset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \qquad \overset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

- Often, raw data samples have a lot of redundancy and patterns
- PCA can allow you to represent data samples as weights on the principal components, rather than using the original raw form of the data
- By representing each sample as just those weights, you can represent just the “meat” of what’s different between samples.
- This minimal representation makes machine learning and other algorithms much more efficient

How is SVD computed?

- For this class: tell PYTHON to do it. Use the result.
- But, if you're interested, one computer algorithm to do it makes use of Eigenvectors!

Eigenvector definition

- Suppose we have a square matrix **A**. We can solve for vector x and scalar λ such that $Ax = \lambda x$
- In other words, find vectors where, if we transform them with **A**, the only effect is to scale them with no change in direction.
- These vectors are called eigenvectors (German for "self vector" of the matrix), and the scaling factors λ are called eigenvalues
- An m x m matrix will have $\leq m$ eigenvectors where λ is nonzero

Finding eigenvectors

- Computers can find an x such that Ax= λx using this iterative algorithm:

  - X = random unit vector
  - while(x hasn't converged)
    - X = Ax
    - normalize x

- x will quickly converge to an eigenvector
- Some simple modifications will let this algorithm find all eigenvectors

Finding SVD

- Eigenvectors are for square matrices, but SVD is for all matrices

Gram matrix : V.T * V

- To do svd(A), computers can do this:
  - Take eigenvectors of AA^T (matrix is always square).
    - These eigenvectors are the columns of **U**.
    - Square root of eigen*values* are the singular values (the entries of **Σ**).
  - Take eigenvectors of A^TA (matrix is always square).
    - These eigenvectors are columns of **V** (or rows of $\mathbf{V}^T$)

assumed A (- Rm*n
Then U (- Rm*m
V (- Rn*n
This way, AA.T (- Rm*m
A.TA (- Rn*n

Finding SVD

- Moral of the story: SVD is fast, even for large matrices
- It's useful for a lot of stuff
- There are also other algorithms to compute SVD or part of the SVD
  - Python's np.linalg.svd() command has options to efficiently compute only what you need, if performance becomes an issue

A detailed geometric explanation of SVD is here:
http://www.ams.org/samplings/feature-column/fcarc-svd

What we will learn today

- Introduction to face recognition
- Principal Component Analysis (PCA)
- Image compression

Covariance

- Variance and Covariance are a measure of the “spread” of a set of points around their center of mass (mean)
- Variance – measure of the deviation from the mean for points in one dimension e.g. heights
- Covariance as a measure of how much each of the dimensions vary from the mean with respect to each other.
- Covariance is measured between 2 dimensions to see if there is a relationship between the 2 dimensions e.g. number of hours studied & marks obtained.
- The covariance between one dimension and itself is the variance

Covariance

$$\text{covariance}(X,Y) = \frac{\sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y})}{(n-1)}$$

$$\mathrm{cov}(X, Y) = E\{[X - E(X)][Y - E(Y)]\}$$
$$\mathrm{cov}(X, Y) = E(XY) - E(X)E(Y)$$

- So, if you had a 3-dimensional data set (x,y,z), then you could measure the covariance between the x and y dimensions, the y and z dimensions, and the x and z dimensions. Measuring the covariance between x and x , or y and y , or z and z would give you the variance of the x , y and z dimensions respectively

Covariance matrix

- Representing Covariance between dimensions as a matrix e.g. for 3 dimensions

$$C = \begin{bmatrix} \mathrm{cov}(x,x) & \mathrm{cov}(x,y) & \mathrm{cov}(x,z) \\ \mathrm{cov}(y,x) & \mathrm{cov}(y,y) & \mathrm{cov}(y,z) \\ \mathrm{cov}(z,x) & \mathrm{cov}(z,y) & \mathrm{cov}(z,z) \end{bmatrix}$$

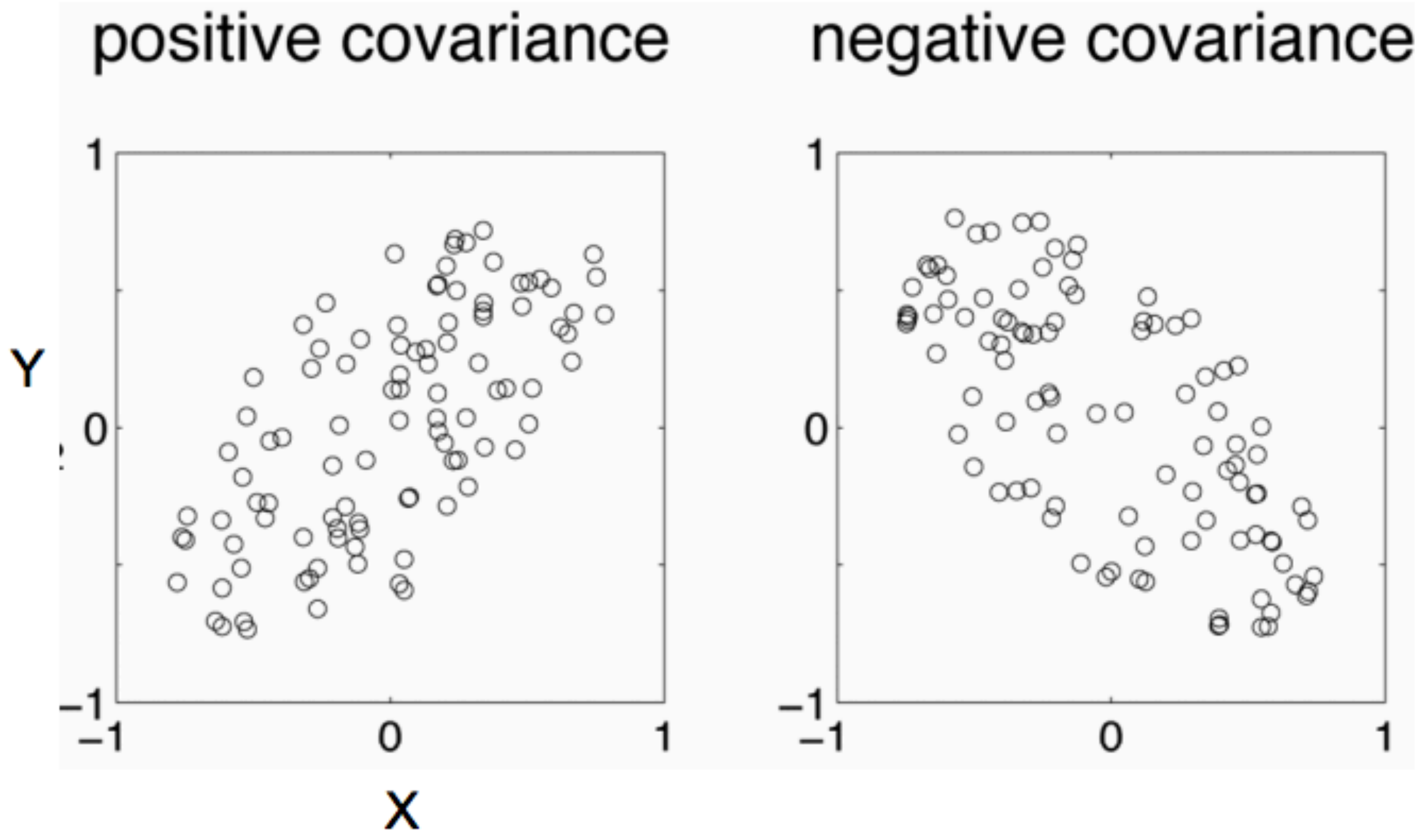
Variances

- Diagonal is the variances of x, y and z
- cov(x,y) = cov(y,x) hence matrix is symmetrical about the diagonal
- N-dimensional data will result in NxN covariance matrix

Covariance

- What is the interpretation of covariance calculations?
  - e.g.: 2 dimensional data set
  - x: number of hours studied for a subject
  - y: marks obtained in that subject
  - covariance value is say: 104.53
  - what does this value mean?

Covariance interpretation

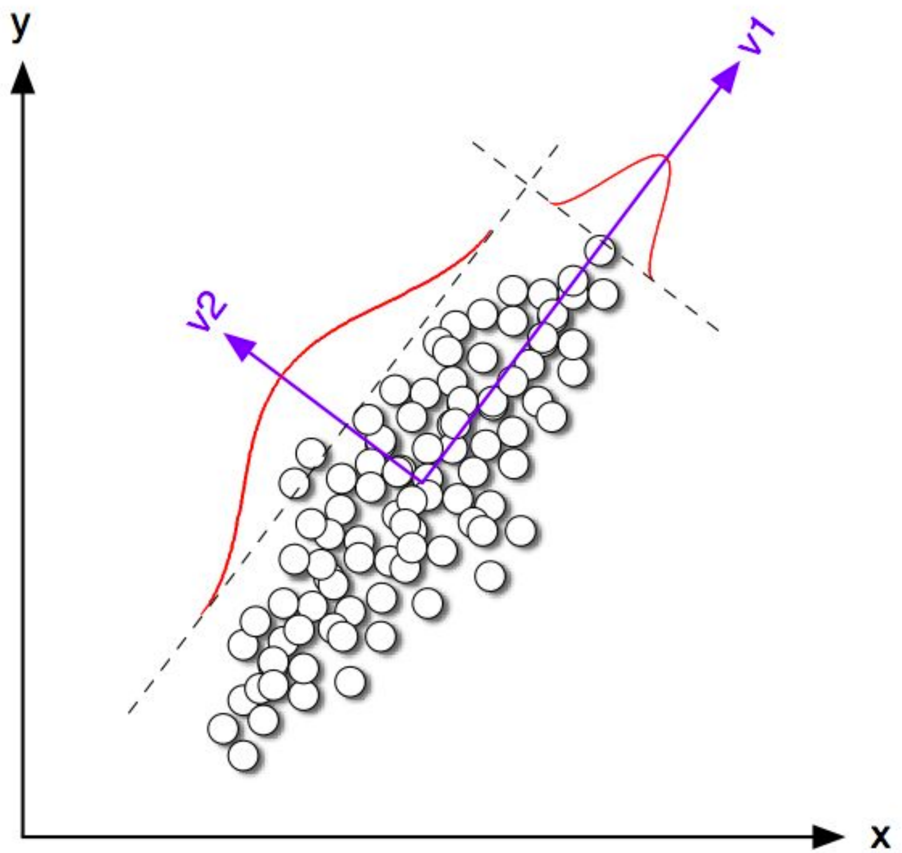


Covariance interpretation

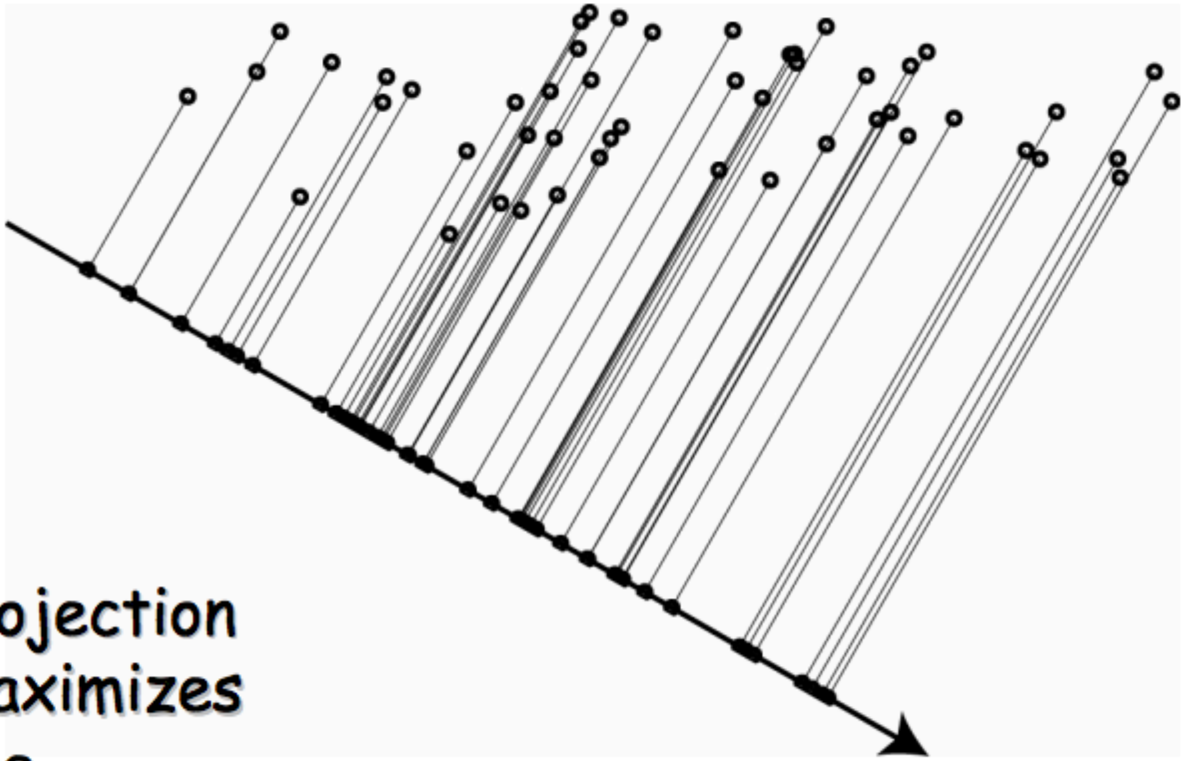
- Exact value is not as important as it’s sign.
- A **positive value** of covariance indicates both dimensions increase or decrease together e.g. as the number of hours studied increases, the marks in that subject increase.
- A **negative value** indicates while one increases the other decreases, or vice-versa e.g. active social life at PSU vs performance in CS dept.
- If **covariance is zero**: the two dimensions are independent of each other e.g. heights of students vs the marks obtained in a subject

Example data

Covariance between the two axis is high. Can we reduce the number of dimensions to just 1?



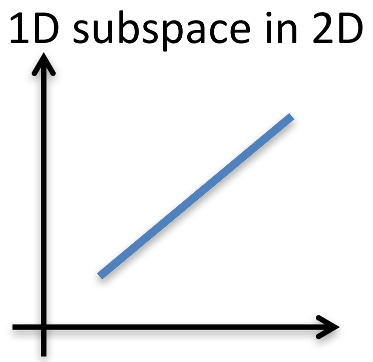
Geometric interpretation of PCA



find projection
that maximizes
variance

Geometric interpretation of PCA

- Let's say we have a set of 2D data points x. But we see that all the points lie on a line in 2D.
- So, 2 dimensions are redundant to express the data. We can express all the points with just one dimension.

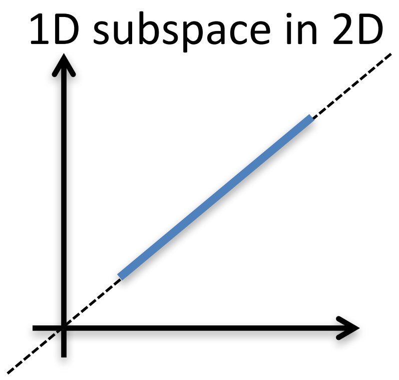


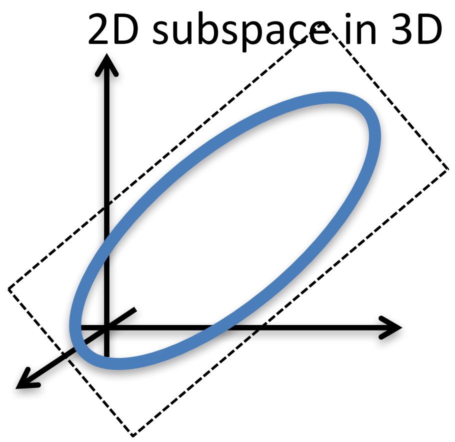
PCA: Principle Component Analysis

- Given a set of points, how do we know if they can be compressed like in the previous example?
  - The answer is to look into the correlation between the points
  - The tool for doing this is called PCA

PCA Formulation

- Basic idea:
  - If the data lives in a subspace, it is going to look very flat when viewed from the full space, e.g.

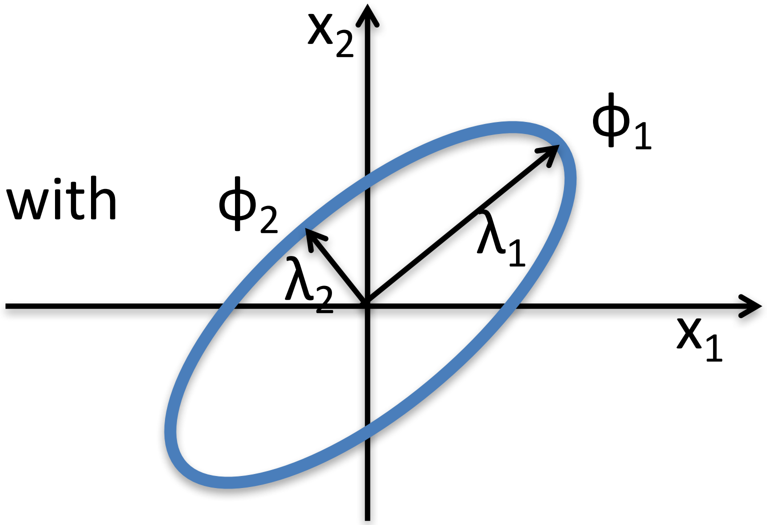




Slide inspired by N. Vasconcelos

PCA Formulation

- Assume x is Gaussian with covariance Σ.



- Recall that a gaussian is defined with it's mean and variance:

$$\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

- Recall that µ and Σ of a gaussian are defined as:

$$\boldsymbol{\mu} = \mathbf{E}[\mathbf{X}] = [\mathbf{E}[X_1], \mathbf{E}[X_2], \ldots, \mathbf{E}[X_k]]^{\mathrm{T}}$$

$$\boldsymbol{\Sigma} =: \mathbf{E}[(\mathbf{X} - \boldsymbol{\mu})(\mathbf{X} - \boldsymbol{\mu})^{\mathrm{T}}] = [\mathrm{Cov}[X_i, X_j]; 1 \leq i, j \leq k]$$

PCA formulation

- Since gaussians are symmetric, it's covariance matrix is also a symmetric matrix. So we can express it as:
  - $\boldsymbol{\Sigma} = \mathbf{U\Lambda U}^T = \mathbf{U\Lambda}^{1/2}(\mathbf{U\Lambda}^{1/2})^T$

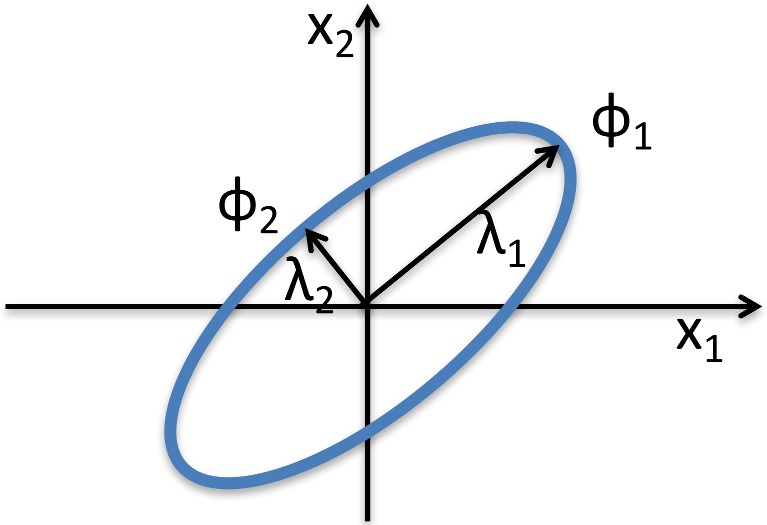
$$\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \iff \mathbf{X} \sim \boldsymbol{\mu} + \mathbf{U}\boldsymbol{\Lambda}^{1/2}\mathcal{N}(0, \mathbf{I})$$

$$\iff \mathbf{X} \sim \boldsymbol{\mu} + \mathbf{U}\mathcal{N}(0, \boldsymbol{\Lambda}).$$

U.T * (X - mu) ~ N(0, /\)
U.T * X ~ N(U.T * mu, /\)

PCA Formulation

- If x is Gaussian with covariance Σ,
  - Principal components ϕ_i are the eigenvectors of Σ
  - Principal lengths λ_i are the eigenvalues of Σ



- by computing the eigenvalues we know the data is
  - Not flat if $\lambda_1 \approx \lambda_2$
  - Flat if $\lambda_1 >> \lambda_2$

Slide inspired by N. Vasconcelos

PCA Algorithm (training)

- Given sample $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}, \; x_i \in \mathcal{R}^d$

  - $\boxed{\text{compute sample mean: } \hat{\mu} = \frac{1}{n}\sum_i(\mathbf{x}_i)}$
  - compute sample covariance: $\hat{\Sigma} = \frac{1}{n}\sum_i(\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^T$
  - compute eigenvalues and eigenvectors of $\hat{\Sigma}$

$$\hat{\Sigma} = \Phi\Lambda\Phi^T, \;\; \Lambda = diag(\sigma_1^2, \ldots, \sigma_n^2) \;\; \Phi^T\Phi = I$$

  - order eigenvalues $\sigma_1^2 > \ldots > \sigma_n^2$
  - if, for a certain k, $\sigma_k << \sigma_1$ eliminate the eigenvalues and eigenvectors above k.

Slide inspired by N. Vasconcelos

PCA Algorithm (testing)

- Given principal compoenents $\phi_i, i \in 1, \ldots, k$ and a test sample $\mathcal{T} = \{\mathbf{t}_1, \ldots, \mathbf{t}_n\}, \ t_i \in \mathcal{R}^d$

  - $\boxed{\text{subtract mean to each point } \mathbf{t}'_i = \mathbf{t}_i - \hat{\mu}}$

  - project onto eigenvector space $\mathbf{y}_i = \mathbf{A}\mathbf{t}'_i$ where

$$\mathbf{A} = \begin{bmatrix} \phi_1^T \\ \vdots \\ \phi_k^T \end{bmatrix}$$

  - use $\mathcal{T}' = \{\mathbf{y}_1, \ldots \mathbf{y}_n\}$ to estimate class conditional densities and do all further processing on $\mathbf{y}$.

Slide inspired by N. Vasconcelos

PCA by SVD

- An alternative manner to compute the principal components, based on singular value decomposition
- Quick reminder: SVD
  - Any real n x m matrix (n>m) can be decomposed as

$$\boxed{A = M\Pi N^T}$$

reduced singular value decomposition

  - Where M is an (n x m) column orthonormal matrix of left singular vectors (columns of M)
  - Π is an (m x m) diagonal matrix of singular values
  - N^T is an (m x m) row orthonormal matrix of right singular vectors (columns of N)

$$\boxed{M^T M = I \qquad N^T N = I}$$

Slide inspired by N. Vasconcelos

PCA by SVD

- To relate this to PCA, we consider the data matrix

$$\boxed{X = \begin{bmatrix} | & & | \\ x_1 & \dots & x_n \\ | & & | \end{bmatrix}}$$

- The sample mean is

$$\boxed{\mu = \frac{1}{n}\sum_i x_i = \frac{1}{n}\begin{bmatrix} | & & | \\ x_1 & \dots & x_n \\ | & & | \end{bmatrix}\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} = \frac{1}{n}X1}$$

Slide inspired by N. Vasconcelos

PCA by SVD

- Center the data by subtracting the mean to each column of X
- The centered data matrix is

$$X_c = \begin{bmatrix} | & & | \\ x_1 & \dots & x_n \\ | & & | \end{bmatrix} - \begin{bmatrix} | & & | \\ \mu & \dots & \mu \\ | & & | \end{bmatrix}$$

$$= X - \mu 1^T = X - \frac{1}{n} X 1 1^T = X\left(I - \frac{1}{n} 1 1^T \right)$$

Slide inspired by N. Vasconcelos

PCA by SVD

- The sample covariance matrix is

$$\boxed{\Sigma = \frac{1}{n}\sum_i (x_i - \mu)(x_i - \mu)^T = \frac{1}{n}\sum_i x_i^c \left(x_i^c\right)^T}$$

where x_i^c is the ith column of X_c

- This can be written as

$$\boxed{\Sigma = \frac{1}{n}\begin{bmatrix} | & & | \\ x_1^c & \dots & x_n^c \\ | & & | \end{bmatrix}\begin{bmatrix} - & x_1^c & - \\ & \vdots & \\ - & x_n^c & - \end{bmatrix} = \frac{1}{n} X_c X_c^T}$$

Slide inspired by N. Vasconcelos

PCA by SVD

- The matrix

$$X_c^T = \begin{bmatrix} - & x_1^c & - \\ & \vdots & \\ - & x_n^c & - \end{bmatrix}$$

in form of reduced svd
PI is diagonal

is real (n x d). Assuming n>d it has SVD decomposition

$$X_c^T = \mathrm{M}\Pi\mathrm{N}^T \qquad \mathrm{M}^T\mathrm{M} = I \qquad \mathrm{N}^T\mathrm{N} = I$$

and

$$\Sigma = \frac{1}{n} X_c X_c^T = \frac{1}{n} \mathrm{N}\Pi\mathrm{M}^T\mathrm{M}\Pi\mathrm{N}^T = \frac{1}{n} \mathrm{N}\Pi^2\mathrm{N}^T$$

Slide inspired by N. Vasconcelos

PCA by SVD

$$\boxed{\Sigma = \mathrm{N}\left(\frac{1}{n}\Pi^2\right)\mathrm{N}^T}$$

- Note that N is (d x d) and orthonormal, and Π^2 is diagonal. This is just the eigenvalue decomposition of Σ
- It follows that
  - The eigenvectors of Σ are the columns of N
  - The eigenvalues of Σ are

$$\boxed{\lambda_i = \frac{1}{n}\pi_i^2}$$

- This gives an alternative algorithm for PCA

Slide inspired by N. Vasconcelos

PCA by SVD

- In summary, computation of PCA by SVD
- Given X with one example per column
  - Create the centered data matrix

$$\boxed{X_c^T = \left(I - \frac{1}{n} 1 1^T\right) X^T}$$

  - Compute its SVD

$$\boxed{X_c^T = \mathrm{M}\Pi\mathrm{N}^T}$$

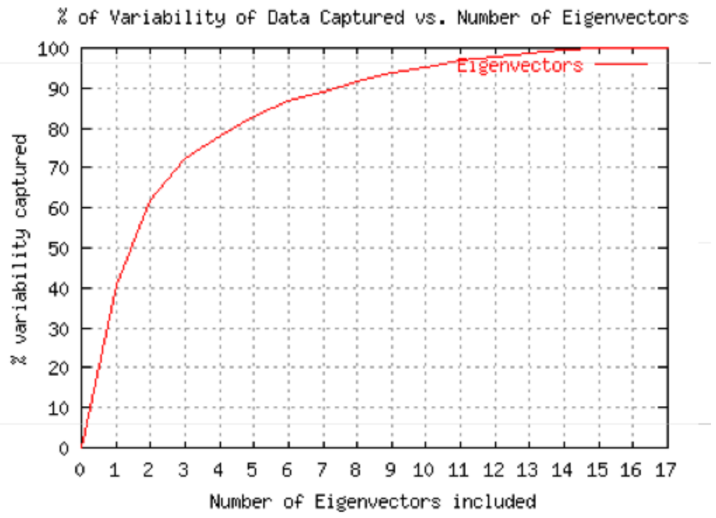
  - Principal components are columns of N, eigenvalues are

$$\boxed{\lambda_i = \frac{1}{n}\pi_i^2}$$

Slide inspired by N. Vasconcelos

Rule of thumb for finding the number of PCA components

- A natural measure is to pick the eigenvectors that explain p% of the data variability
  - Can be done by plotting the ratio r_k as a function of k



$$r_k = \sum_{i=1}^{k} \lambda_i^2 \Big/ \sum_{i=1}^{n} \lambda_i^2$$

  - E.g. we need 3 eigenvectors to cover 70% of the variability of this dataset

Slide inspired by N. Vasconcelos

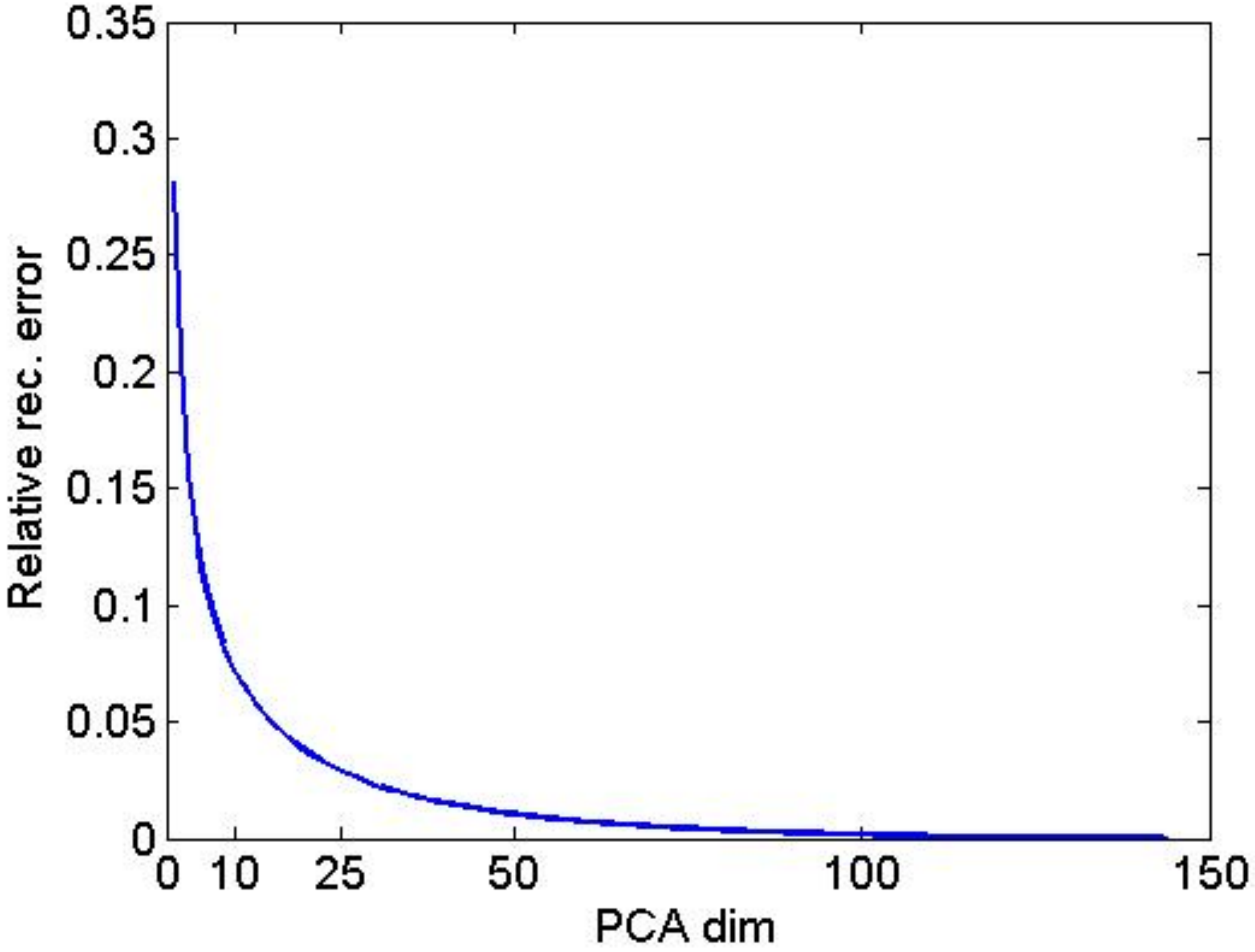
What we will learn today

- Introduction to face recognition
- Principal Component Analysis (PCA)
- Image compression

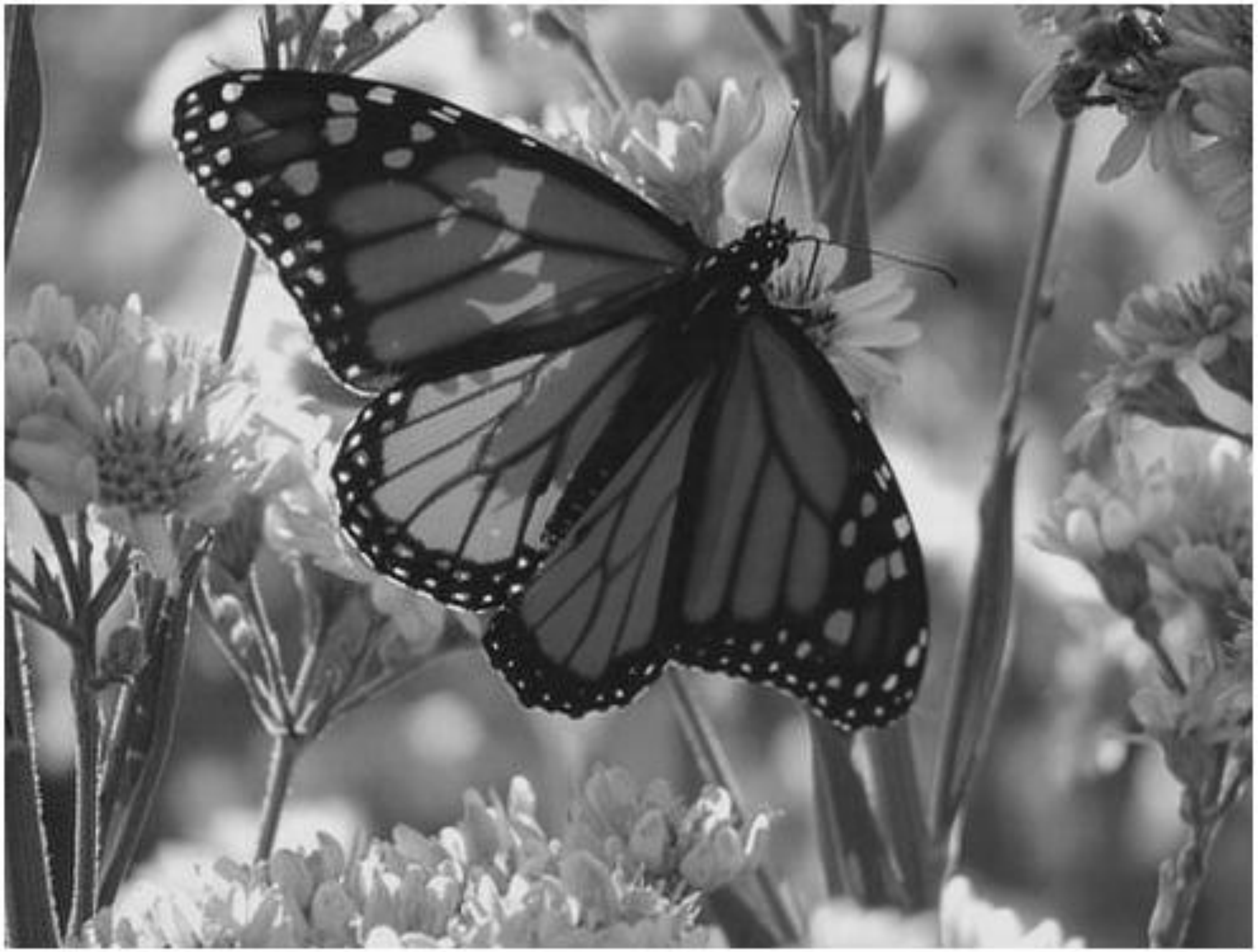
Original Image

- Divide the original 372x492 image into patches:
  - Each patch is an instance that contains 12x12 pixels on a grid
- View each as a 144-D vector

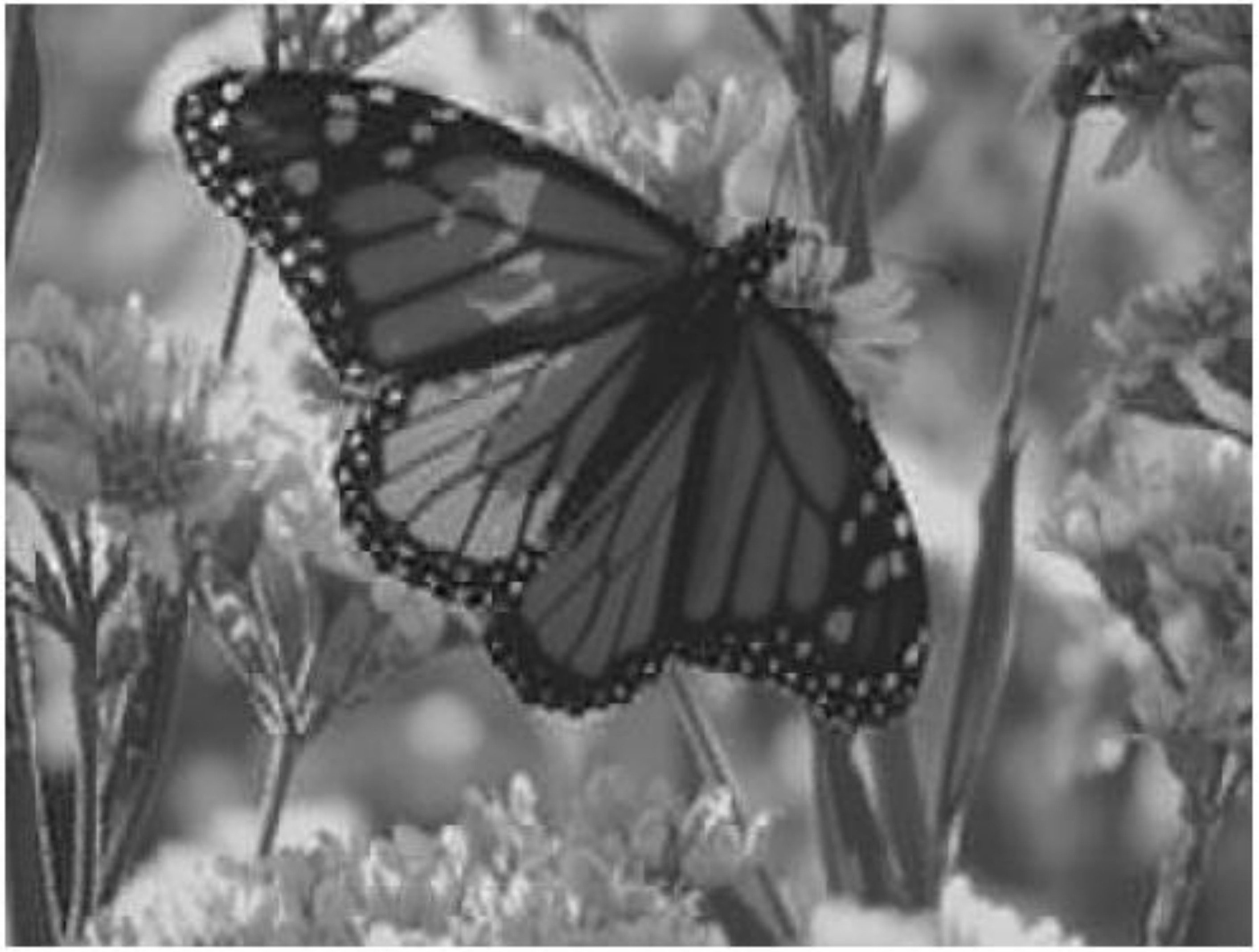
L_2 error and PCA dim



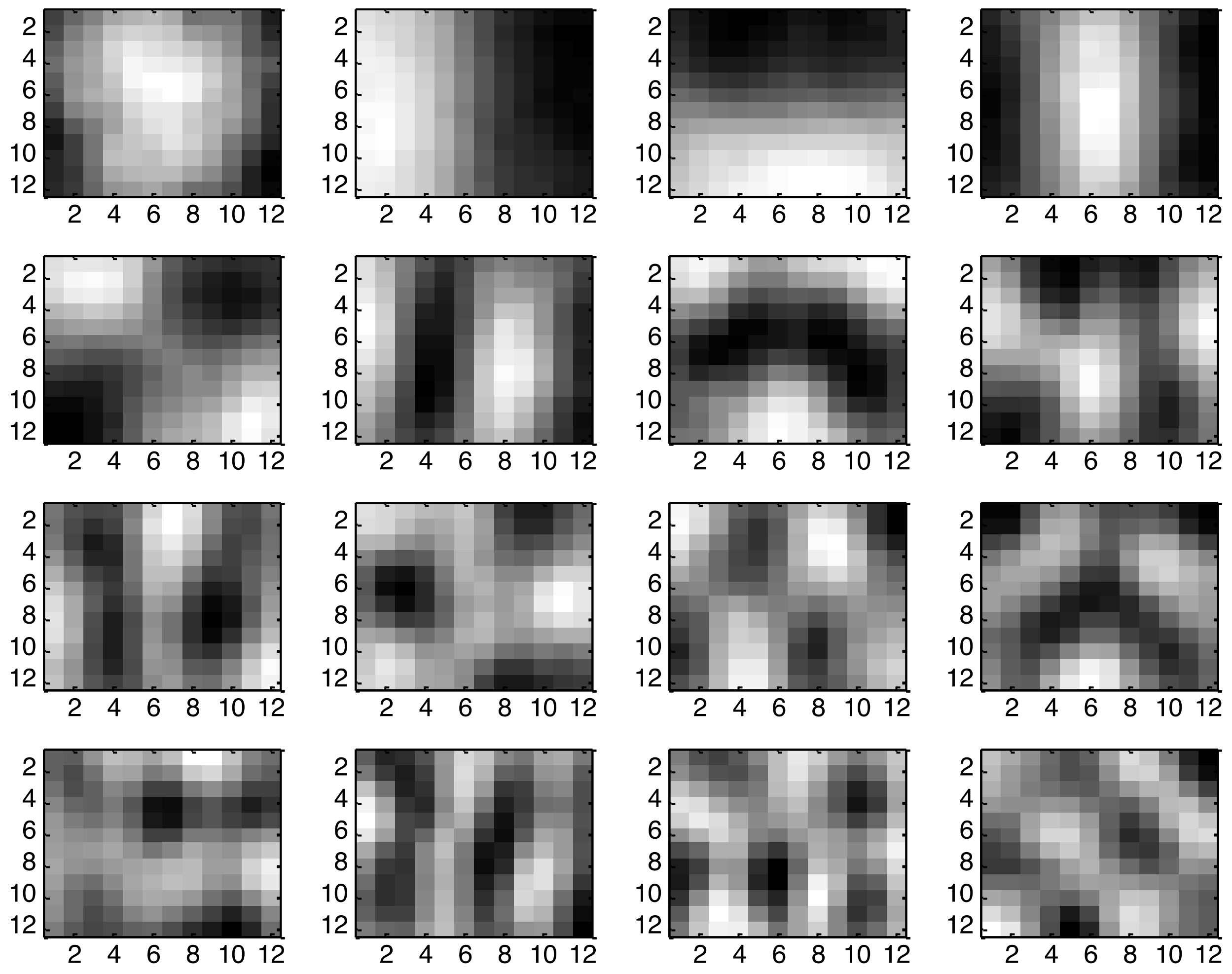
PCA compression: 144D) 60D



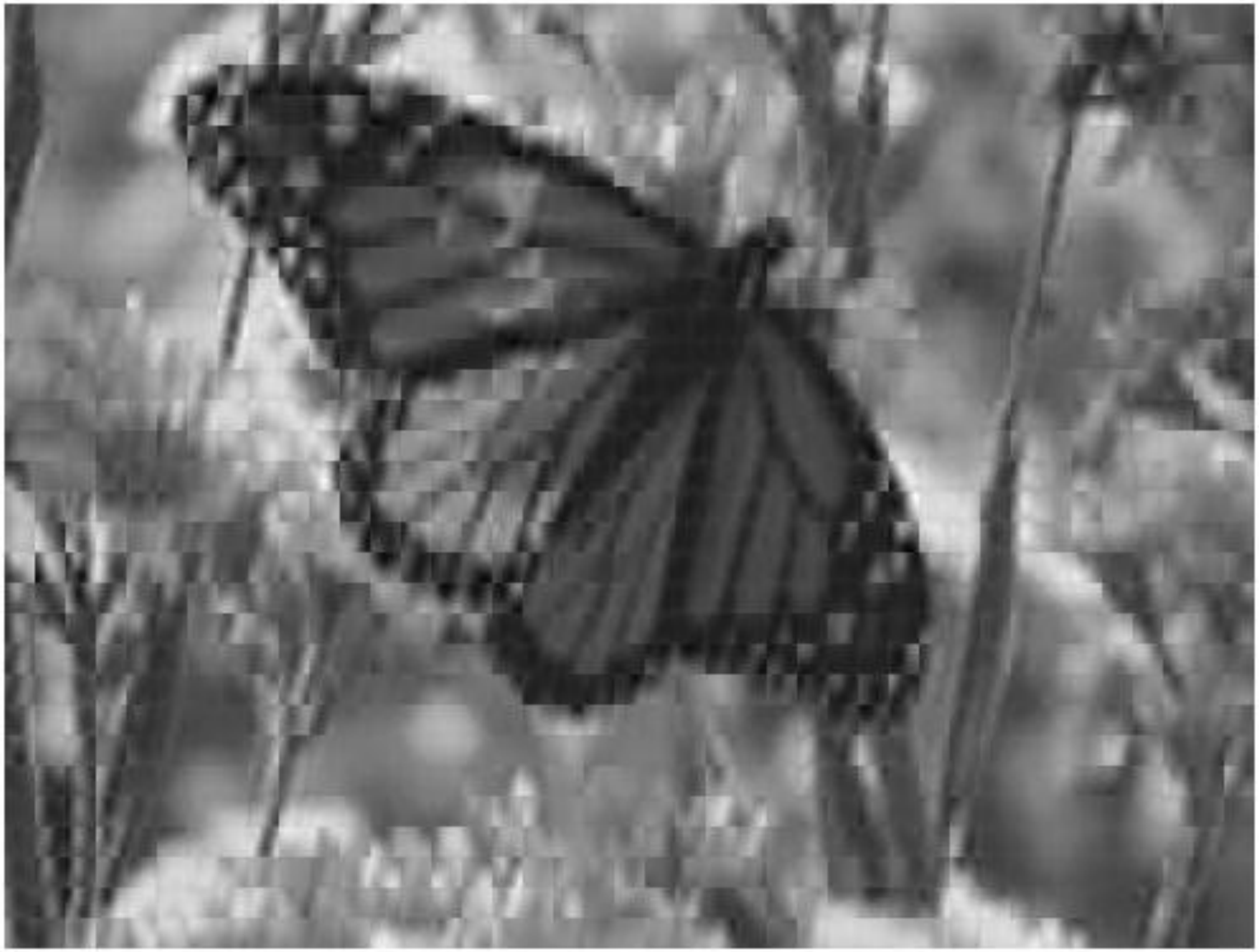
PCA compression: 144D) 16D



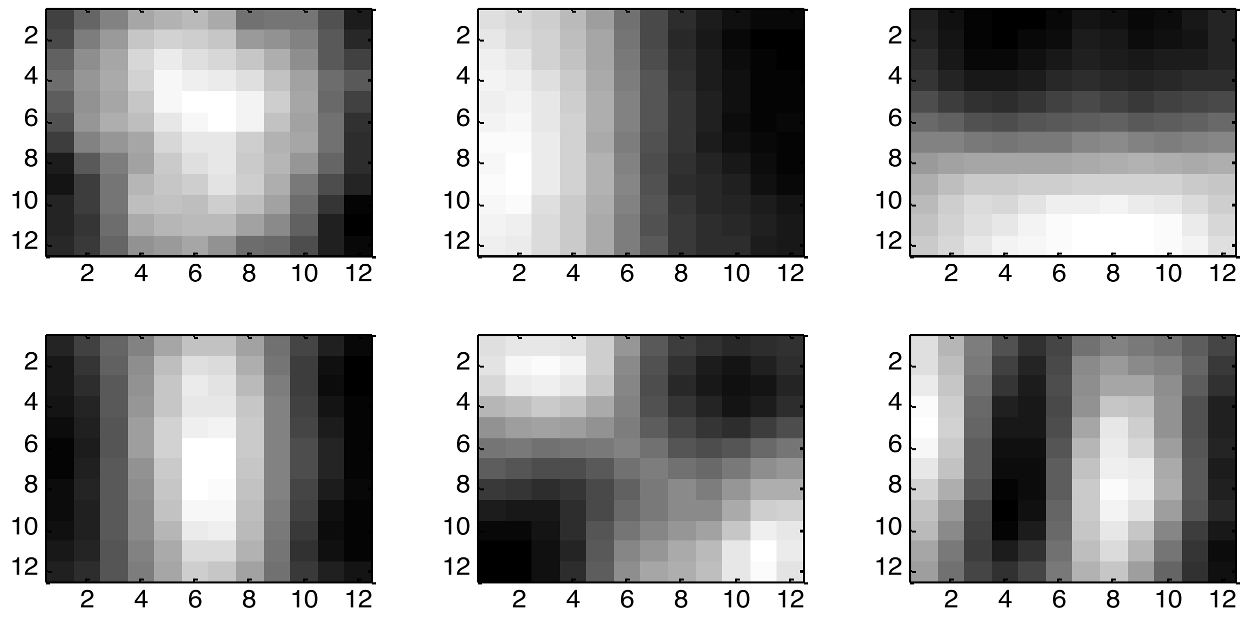
16 most important eigenvectors



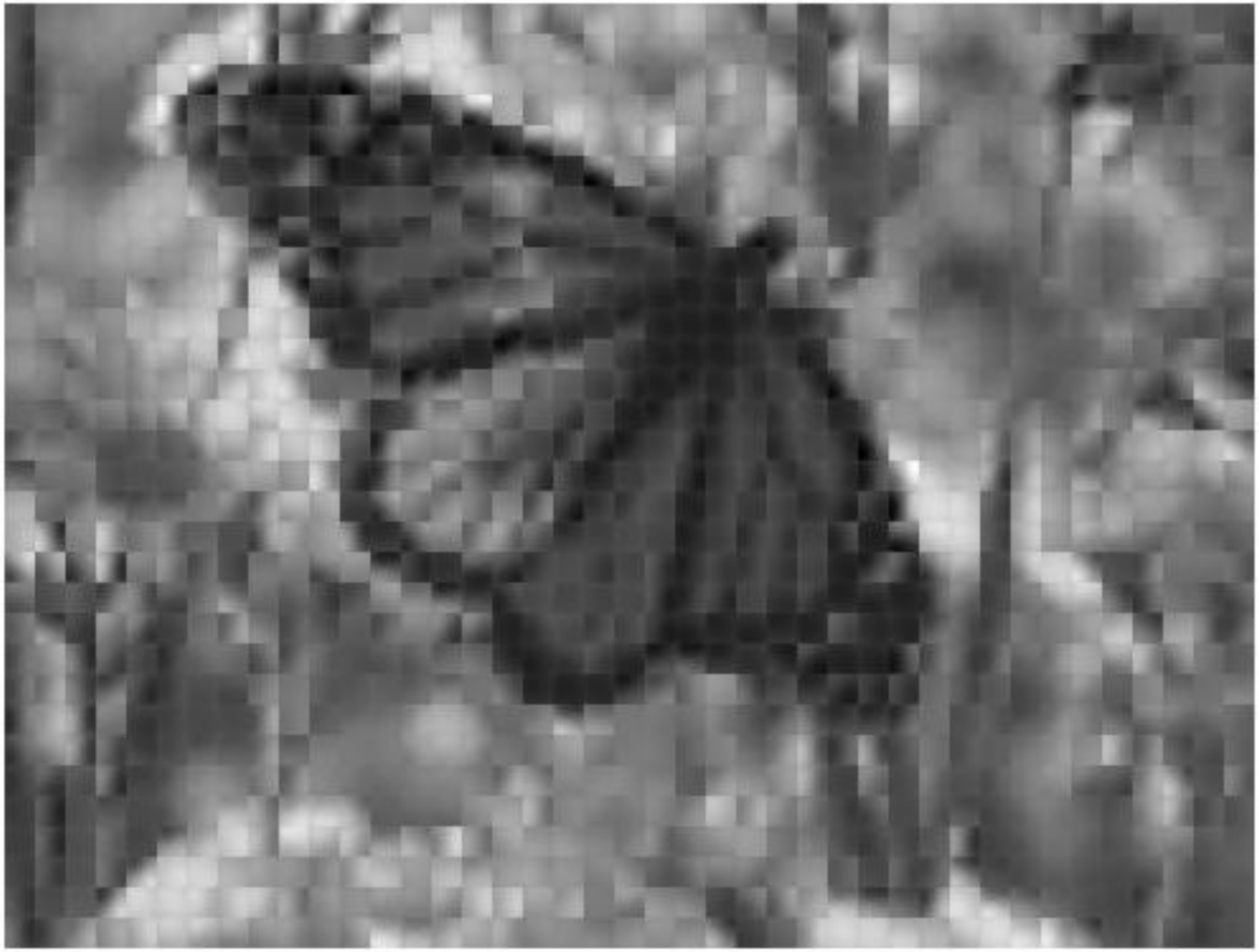
PCA compression: 144D) 6D



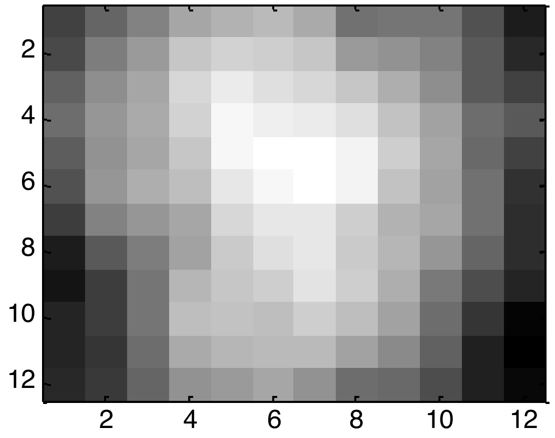
6 most important eigenvectors

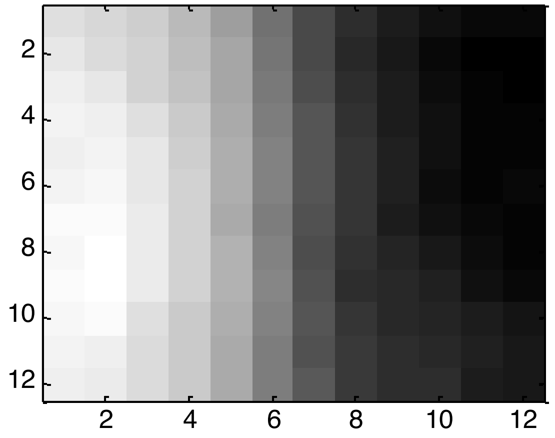


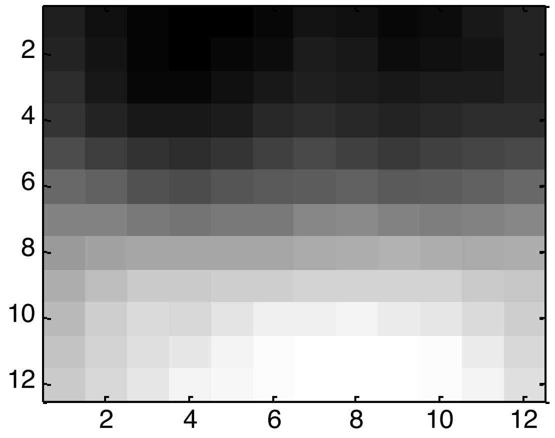
PCA compression: 144D) 3D



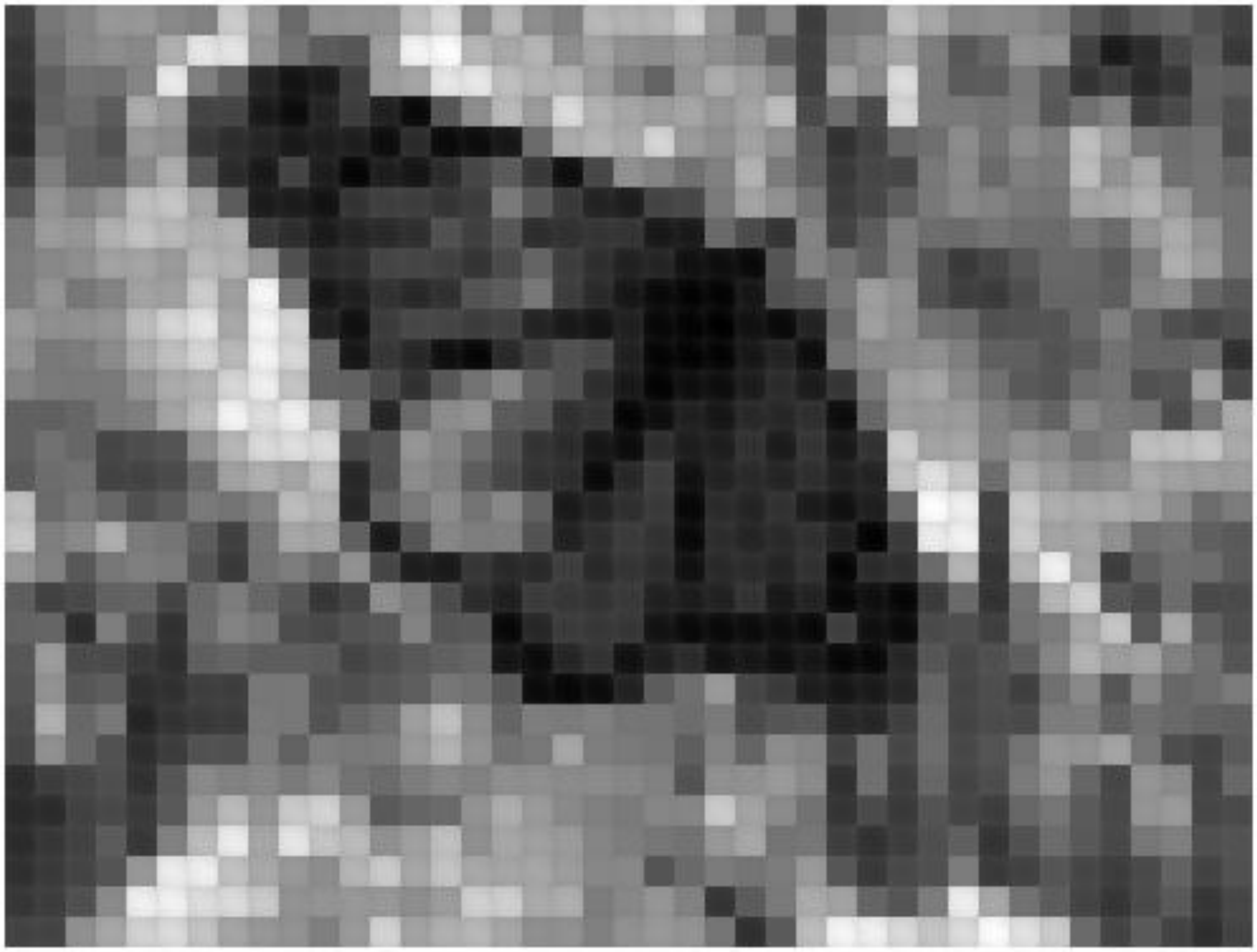
3 most important eigenvectors







PCA compression: 144D) 1D



What we have learned today

- Introduction to face recognition
- Principal Component Analysis (PCA)
- Image compression