PCA-based Object Recognition

Textbook: T&V Section 10.4

Slide material:

Octavia Camps, PSU

S. Narasimhan, CMU

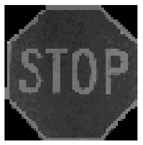
Template Matching

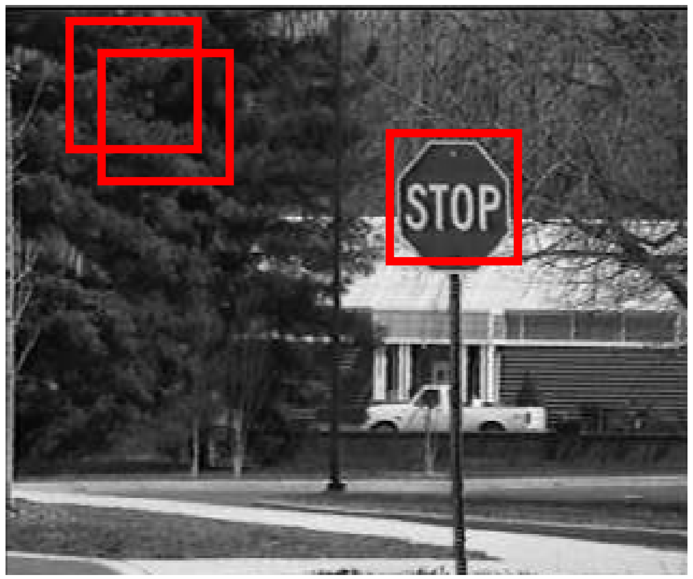
Objects can be represented by storing sample images or "templates"

Stop sign template

Hypotheses from Template Matching

- Place the template at every location on the given image.
  - Compare the pixel values in the template with the pixel values in the underlying region of the image.
  - If a "good" match is found, announce that the object is present in the image.

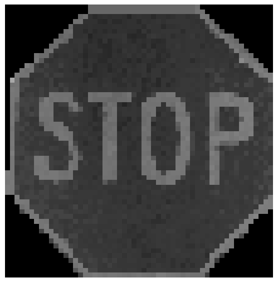


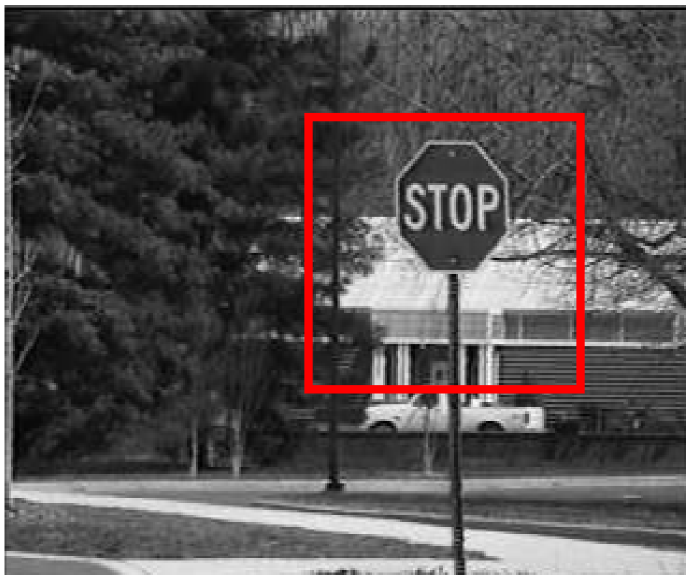


- Possible measures are: SSD, SAD, Cross-correlation, Normalized Cross-correlation, max difference, etc.

Limitations of Template Matching

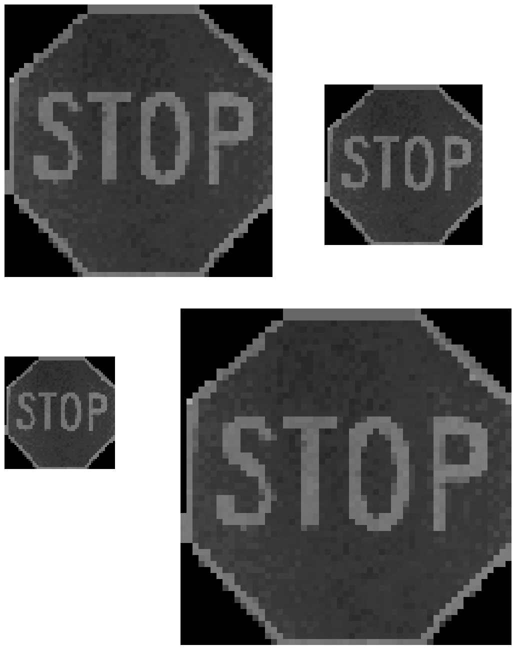
- If the object appears scaled, rotated, or skewed on the image, the match will not be good.





Solution:

- Search for the template and possible transformations of the template:

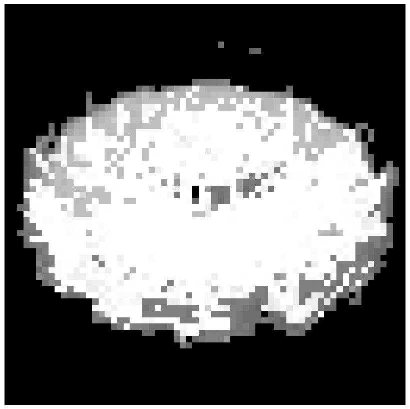


Not very efficient! (but doable …)

Using Eigenspaces

- The appearance of an object in an image depends on several things:
  - Viewpoint
  - Illumination conditions
  - Sensor
  - The object itself (ex: human facial expression)
- In principle, these variations can be handled by increasing the number of templates.

Eigenspaces: Using multiple templates



- The number of templates can grow very fast!
- We need:
  - An efficient way to store templates
  - An efficient way to search for matches

- **Observation:** while each template is different, there exist many similarities between the templates.

Efficient Image Storage

Toy Example: Images with 3 pixels

Consider the following 3x1 templates:

1	2	4	3	5	6
2	4	8	6	10	12
3	6	12	9	15	18

If each pixel is stored in a byte, we need 18 = 3 x 6 bytes

Efficient Image Storage

Looking closer, we can see that all the images are very similar to each other: they are all the same image, scaled by a factor:

$$\begin{bmatrix}1\\2\\3\end{bmatrix} = 1 * \begin{bmatrix}1\\2\\3\end{bmatrix} \quad \begin{bmatrix}2\\4\\6\end{bmatrix} = 2 * \begin{bmatrix}1\\2\\3\end{bmatrix} \quad \begin{bmatrix}4\\8\\12\end{bmatrix} = 4 * \begin{bmatrix}1\\2\\3\end{bmatrix}$$

$$\begin{bmatrix}3\\6\\9\end{bmatrix} = 3 * \begin{bmatrix}1\\2\\3\end{bmatrix} \quad \begin{bmatrix}5\\10\\15\end{bmatrix} = 5 * \begin{bmatrix}1\\2\\3\end{bmatrix} \quad \begin{bmatrix}6\\12\\18\end{bmatrix} = 6 * \begin{bmatrix}1\\2\\3\end{bmatrix}$$

Efficient Image Storage

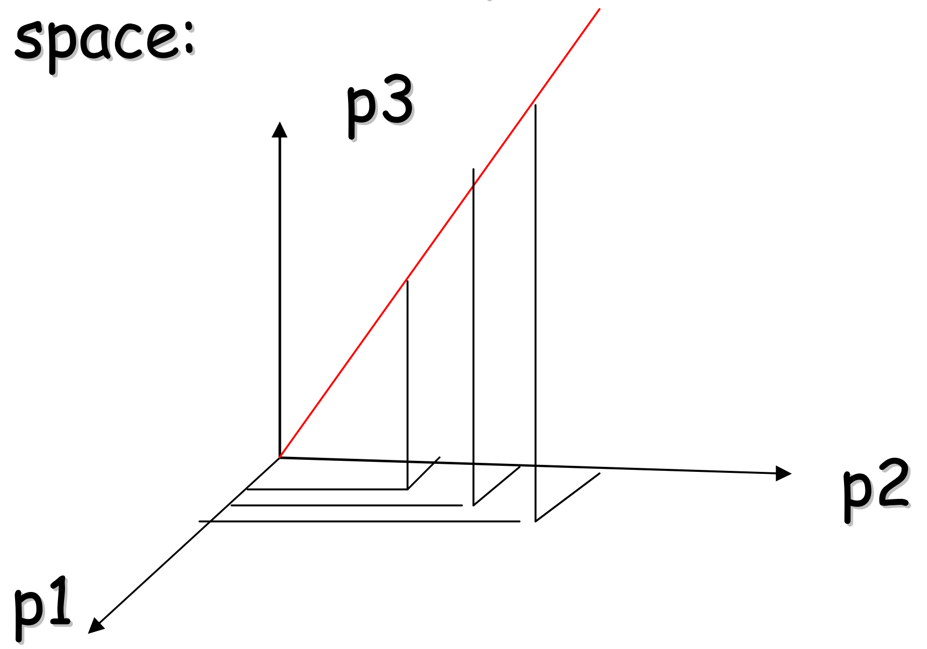
$$\begin{bmatrix}1\\2\\3\end{bmatrix} = 1 * \begin{bmatrix}1\\2\\3\end{bmatrix} \qquad \begin{bmatrix}2\\4\\6\end{bmatrix} = 2 * \begin{bmatrix}1\\2\\3\end{bmatrix} \qquad \begin{bmatrix}4\\8\\12\end{bmatrix} = 4 * \begin{bmatrix}1\\2\\3\end{bmatrix}$$

$$\begin{bmatrix}3\\6\\9\end{bmatrix} = 3 * \begin{bmatrix}1\\2\\3\end{bmatrix} \qquad \begin{bmatrix}5\\10\\15\end{bmatrix} = 5 * \begin{bmatrix}1\\2\\3\end{bmatrix} \qquad \begin{bmatrix}6\\12\\18\end{bmatrix} = 6 * \begin{bmatrix}1\\2\\3\end{bmatrix}$$

They can be stored using only 9 bytes (50% savings!):
Store one image (3 bytes) + the multiplying constants (6 bytes)

Geometrical Interpretation:

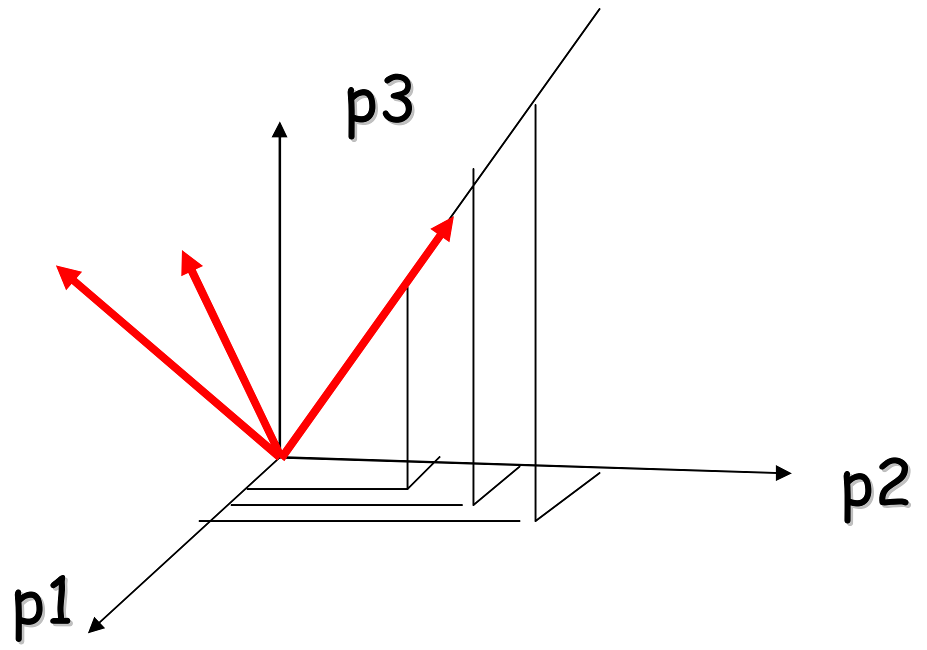
Consider each pixel in the image as a coordinate in a vector space. Then, each 3x1 template can be thought of as a point in a 3D space:



But in this example, all the points happen to belong to a line: a 1D subspace of the original 3D space.

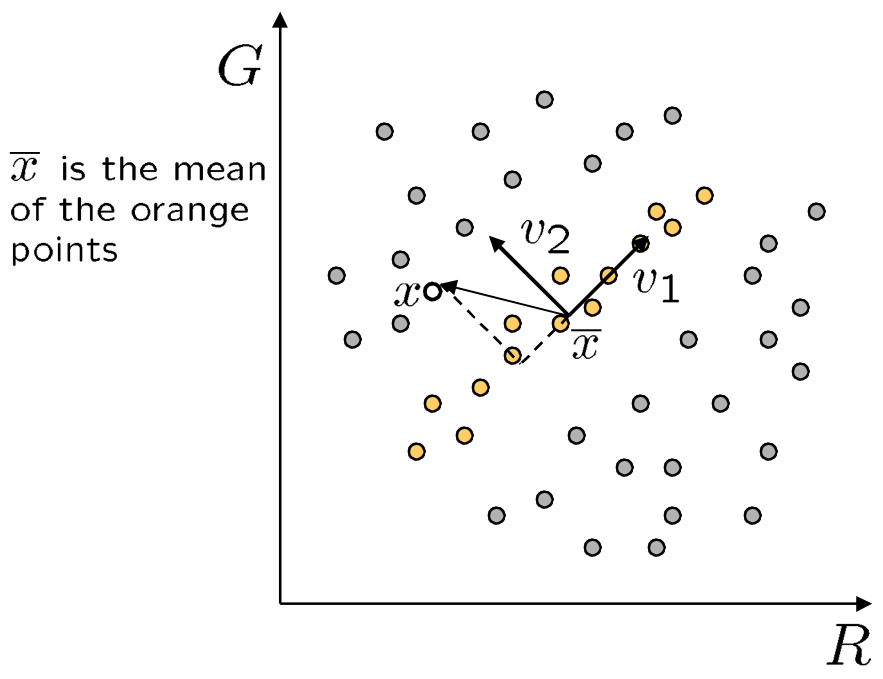
Geometrical Interpretation:

Consider a new coordinate system where one of the axes is along the direction of the line:



In this coordinate system, every image has only one non-zero coordinate: we *only* need to store the direction of the line (a 3 bytes image) and the non-zero coordinate for each of the images (6 bytes).

Linear Subspaces



convert **x** into **v**$_1$, **v**$_2$ coordinates

$$\mathbf{x} \to ((\mathbf{x} - \overline{x}) \cdot \mathbf{v_1}, (\mathbf{x} - \overline{x}) \cdot \mathbf{v_2})$$

What does the **v**$_2$ coordinate measure?

- distance to line
- use it for classification—near 0 for orange pts

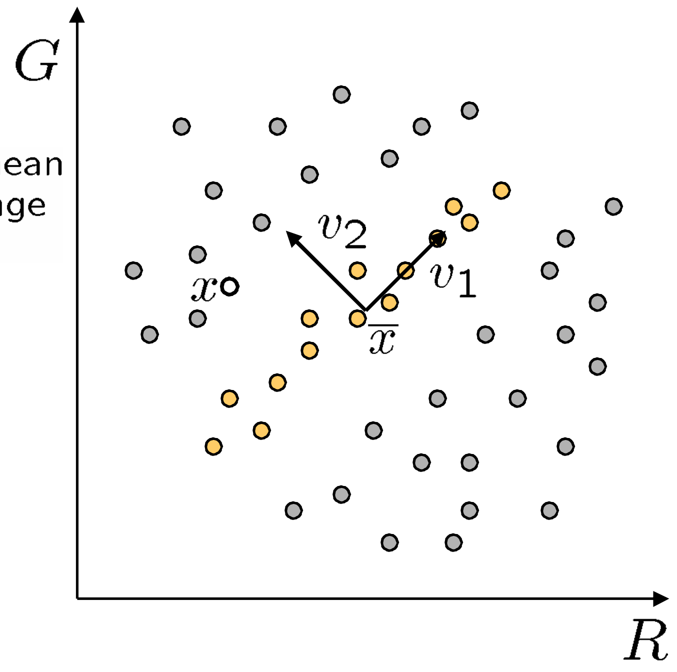
What does the **v**$_1$ coordinate measure?

- position along line
- use it to specify which orange point it is

- Classification can be expensive
  - Must either search (e.g., nearest neighbors) or store large probability density functions.
- Suppose the data points are arranged as above
  - Idea—fit a line, classifier measures distance to line

Dimensionality Reduction

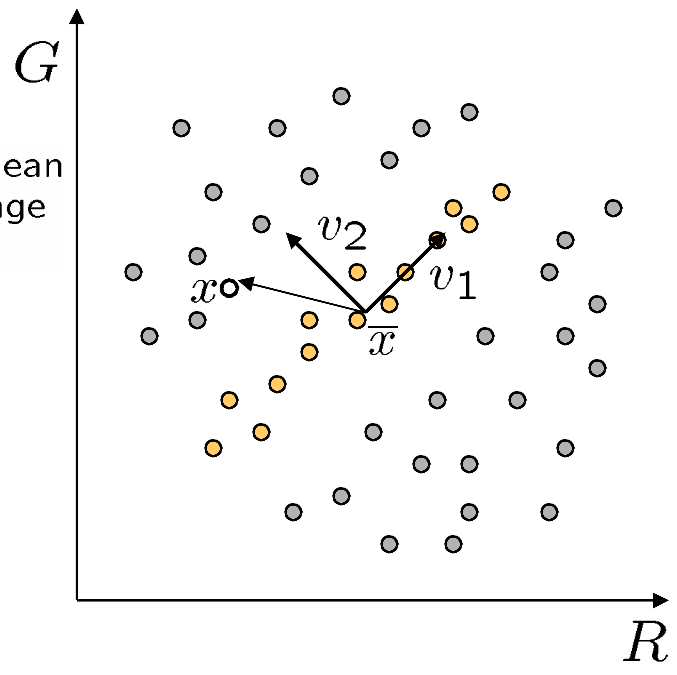
$\overline{x}$ is the mean of the orange points



- Dimensionality reduction
  - We can represent the orange points with *only* their **$\mathbf{v_1}$** coordinates
    - since **$\mathbf{v_2}$** coordinates are all essentially 0
  - This makes it much cheaper to store and compare points
  - A bigger deal for higher dimensional problems

Linear Subspaces

$\overline{x}$ is the mean of the orange points



Consider the variation along direction **v** among all of the orange points:

$$var(\mathbf{v}) = \sum_{\text{orange point } \mathbf{x}} \|(\mathbf{x}-\overline{\mathbf{x}})^{\mathrm{T}} \cdot \mathbf{v}\|^2$$

What unit vector **v** minimizes *var*?

$$\mathbf{v}_2 = min_{\mathbf{v}} \ \{var(\mathbf{v})\}$$

What unit vector **v** maximizes *var*?

$$\mathbf{v}_1 = max_{\mathbf{v}} \ \{var(\mathbf{v})\}$$

$$\begin{aligned} var(\mathbf{v}) &= \sum_{\mathbf{x}} \|(\mathbf{x}-\overline{\mathbf{x}})^{\mathrm{T}} \cdot \mathbf{v}\| \\ &= \sum_{\mathbf{x}} \mathbf{v}^{\mathrm{T}}(\mathbf{x}-\overline{\mathbf{x}})(\mathbf{x}-\overline{\mathbf{x}})^{\mathrm{T}}\mathbf{v} \\ &= \mathbf{v}^{\mathrm{T}}\left[\sum_{\mathbf{x}}(\mathbf{x}-\overline{\mathbf{x}})(\mathbf{x}-\overline{\mathbf{x}})^{\mathrm{T}}\right]\mathbf{v} \\ &= \mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} \quad \text{where } \mathbf{A} = \sum_{\mathbf{x}}(\mathbf{x}-\overline{\mathbf{x}})(\mathbf{x}-\overline{\mathbf{x}})^{\mathrm{T}} \end{aligned}$$

Solution: $\mathbf{v}_1$ is eigenvector of **A** with *largest* eigenvalue
$\mathbf{v}_2$ is eigenvector of **A** with *smallest* eigenvalue

Principal Component Analysis (PCA)

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

PCA Theorem

Let x_1 x_2 … x_n be a set of n N^2 x 1 vectors and let $\bar{x}$ be their average:

$$\mathbf{x}_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{iN^2} \end{bmatrix} \qquad \bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{i=n} \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{iN^2} \end{bmatrix}$$

Note: Each N x N image template can be represented as a N^2 x 1 vector whose elements are the template pixel values.

PCA Theorem

Let X be the N^2 x n matrix with columns $x_1 - \bar{x}$, $x_2 - \bar{x}$,… $x_n - \bar{x}$:

$$X = \left[\begin{array}{cccc} \mathbf{x}_1 - \bar{\mathbf{x}} & \mathbf{x}_2 - \bar{\mathbf{x}} & \cdots & \mathbf{x}_n - \bar{\mathbf{x}} \end{array} \right]$$

Note: subtracting the mean is equivalent to translating the coordinate system to the location of the mean.

PCA Theorem

Let Q = X X^T be the N^2 x N^2 matrix:

$$Q = XX^T = \begin{bmatrix} \mathbf{x}_1 - \bar{\mathbf{x}} & \mathbf{x}_2 - \bar{\mathbf{x}} & \cdots & \mathbf{x}_n - \bar{\mathbf{x}} \end{bmatrix} \begin{bmatrix} (\mathbf{x}_1 - \bar{\mathbf{x}})^T \\ (\mathbf{x}_2 - \bar{\mathbf{x}})^T \\ \vdots \\ (\mathbf{x}_n - \bar{\mathbf{x}})^T \end{bmatrix}$$

Notes:

1. Q is square
2. Q is symmetric
3. Q is the *covariance* matrix [aka scatter matrix]
4. Q can be very large (remember that N^2 is the number of pixels in the template)

PCA Theorem

Theorem:

Each x_j can be written as: $$\mathrm{x}_j = \bar{\mathrm{x}} + \sum_{i=1}^{i=n} g_{ji}\mathrm{e}_i$$

where e_i are the n eigenvectors of Q with non-zero eigenvalues.

Notes:

1. The eigenvectors e_1 e_2 … e_n span an ***eigenspace***
2. e_1 e_2 … e_n are N^2 x 1 orthonormal vectors (N x N images).
3. The scalars g_{ji} are the coordinates of x_j in the space.
4. $$g_{ji} = (\mathrm{x}_j - \bar{\mathrm{x}}).\mathrm{e}_i$$

Using PCA to Compress Data

- Expressing x in terms of $e_1 \ldots e_n$ has not changed the size of the data
- However, if the templates are highly correlated many of the coordinates of x will be zero or closed to zero.

note: this means they lie in a lower-dimensional linear subspace

Using PCA to Compress Data

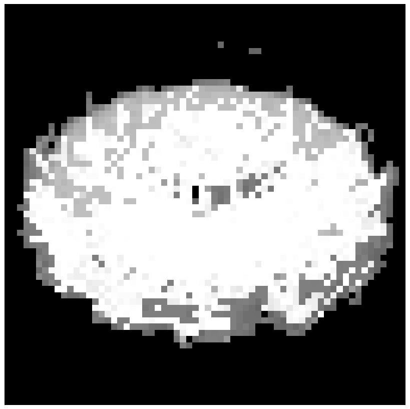
- Sort the eigenvectors e_i according to their eigenvalue:

$$\lambda_1 \geq \lambda_2 \geq \ldots \lambda_n$$

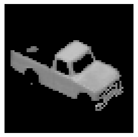
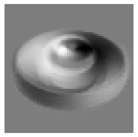
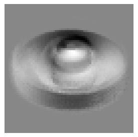
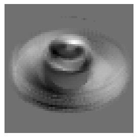
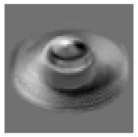
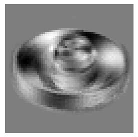
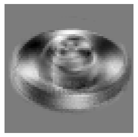
- Assuming that $\lambda_i \approx 0 \text{ if } i > k$

- Then $$\boxed{\mathbf{x}_j \approx \bar{\mathbf{x}} + \sum_{i=1}^{i=k} g_{ji}\mathbf{e}_i}$$

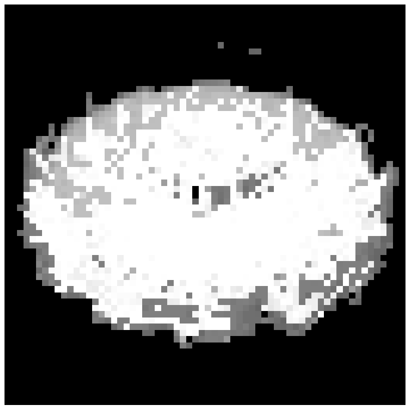
Eigenspaces: Efficient Image Storage



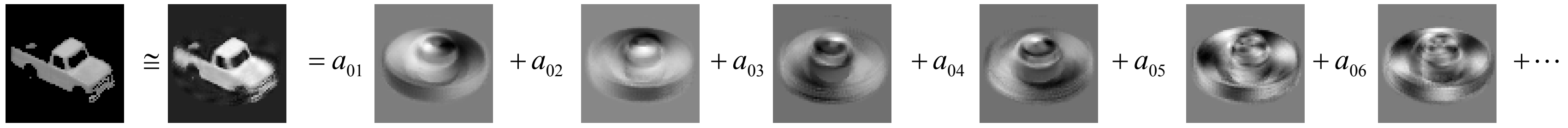
- Use PCA to compress the data:
  - each image is stored as a k-dimensional vector
  - Need to store k N x N eigenvectors
- k << n << N^2

 $\cong$ $= a_{01}$  $+ a_{02}$  $+ a_{03}$  $+ a_{04}$  $+ a_{05}$  $+ a_{06}$  $+ \cdots$

Eigenspaces: Efficient Image Comparison



- Use the same procedure to compress the given image to a k-dimensional vector.
- Compare the compressed vectors:
  - Dot product of k-dimensional vectors
  - k << n << N²



Implementation Details:

- Need to find "first" k eigenvectors of Q:

$$Q = XX^T = \left[\begin{array}{cccc} \mathbf{x}_1 - \bar{\mathbf{x}} & \mathbf{x}_2 - \bar{\mathbf{x}} & \cdots & \mathbf{x}_n - \bar{\mathbf{x}} \end{array} \right] \left[\begin{array}{c} (\mathbf{x}_1 - \bar{\mathbf{x}})^T \\ (\mathbf{x}_2 - \bar{\mathbf{x}})^T \\ \vdots \\ (\mathbf{x}_n - \bar{\mathbf{x}})^T \end{array} \right]$$

Q is $N^2 \times N^2$ where N^2 is the number of pixels in each image. For a 256 x 256 image, $N^2 = 65536$!!

Finding ev of Q

$Q=XX^T$ is very large. Instead, consider the matrix $P=X^TX$

- Q and P are both symmetric, but $Q \neq P^T$
- Q is $N^2 \times N^2$, P is $n \times n$
- n is the number of training images, typically $n \ll N$

Finding ev of Q

Let e be an eigenvector of P with eigenvalue λ:

$$Pe = \lambda e$$

$$X^T X e = \lambda e$$

$$X X^T X e = \lambda X e$$

$$Q(Xe) = \lambda (Xe)$$

Xe is an eigenvector of Q also with eigenvalue λ!

Singular Value Decomposition (SVD)

Any m x n matrix X can be written as the product of 3 matrices:

$$X = UDV^T$$

Where:

- U is m x m and its columns are orthonormal vectors
- V is n x n and its columns are orthonormal vectors
- D is m x n diagonal and its diagonal elements are called the singular values of X, and are such that:

$$\sigma_1 \geq \sigma_2 \geq \ldots \sigma_n \geq 0$$

SVD Properties

$$X = UDV^T$$

- The columns of U are the eigenvectors of XX^T
- The columns of V are the eigenvectors of X^TX
- The squares of the diagonal elements of D are the eigenvalues of XX^T and X^TX

Algorithm EIGENSPACE_LEARN

Assumptions:

1. Each image contains one object only.
2. Objects are imaged by a fixed camera .
3. Images are normalized in size N x N:
   - The image frame is the <u>minimum</u> rectangle enclosing the object.
4. Energy of pixels values is normalized to 1:
   - $\sum_i \sum_j I(i,j)^2 = 1$
5. The object is completely visible and unoccluded in all images.

Algorithm EIGENSPACE_LEARN

Getting the data:

For each object o to be represented, o = 1, …,O

1. Place o on a turntable, acquire a set of n images by rotating the table in increments of 360°/n
2. For each image p, p = 1, …, n:
    1. Segment o from the background
    2. Normalize the image size and energy
    3. Arrange the pixels as vectors $\mathbf{x}^o_p$

Algorithm EIGENSPACE_LEARN

Storing the data:

1. Find the average image vector $\bar{\mathrm{x}} = \frac{1}{n.o} \sum_{o=1}^{O} \sum_{p=1}^{n} x_p^o$

2. Assemble the matrix X:

$$X = \left[\begin{array}{cccc} \mathrm{x}_1^1 - \bar{\mathrm{x}} & \mathrm{x}_2^1 - \bar{\mathrm{x}} & \cdots & \mathrm{x}_n^o - \bar{\mathrm{x}} \end{array} \right]$$

3. Find the first k eigenvectors of XX^T: $e_1, \ldots, e_k$
   (use X^TX or SVD)

4. For each object o, each image p:
   - Compute the corresponding k-dimensional point:

$$\mathrm{g}_p^o = \left[\begin{array}{cccc} \mathrm{e}_1 & \mathrm{e}_2 & \cdots & \mathrm{e}_k \end{array} \right] (\mathrm{x}_p^o - \bar{\mathrm{x}})$$

Algorithm EIGENSPACE_IDENTIF

Recognizing an object from the DB:

1. Given an image, segment the object from the background

2. Normalize the size an energy, write it as a vector **i**

3. Compute the corresponding k-dimensional point:

$$\mathrm{g} = \left[\begin{array}{cccc} \mathrm{e}_1 & \mathrm{e}_2 & \cdots & \mathrm{e}_k \end{array}\right](\mathbf{i} - \bar{\mathrm{x}})$$

4. Find the closest $\boldsymbol{g}^{o}_{p}$ k-dimensional point to ***g***

Key Property of Eigenspace Representation

Given

- 2 images $\hat{x}_1, \hat{x}_2$ that are used to construct the Eigenspace
- $\hat{g}_1$ is the eigenspace projection of image $\hat{x}_1$
- $\hat{g}_2$ is the eigenspace projection of image $\hat{x}_2$

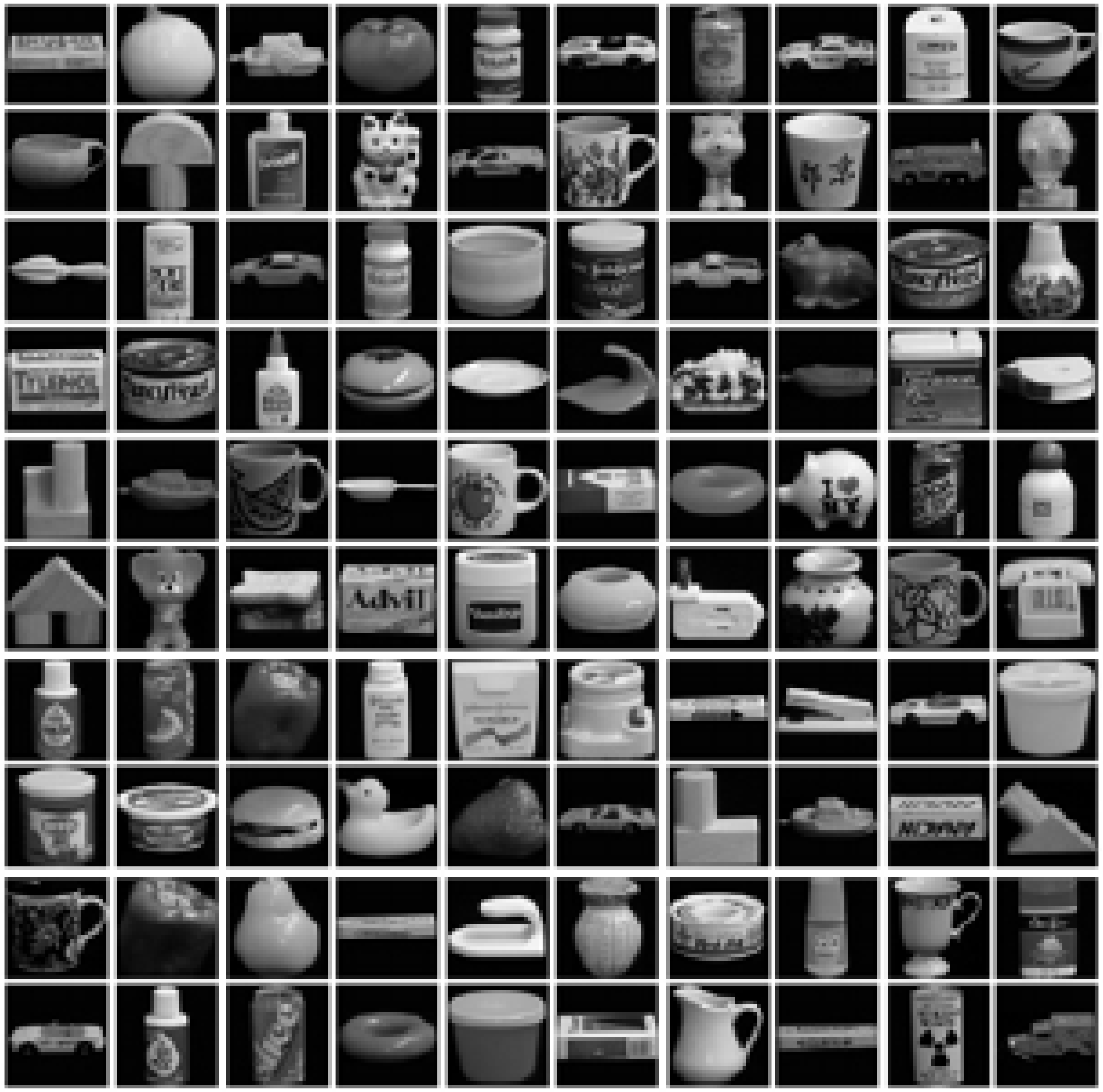
Then,

$$\boxed{\| \hat{g}_2 - \hat{g}_1 \| \approx \| \hat{x}_2 - \hat{x}_1 \|}$$

That is, distance in Eigenspace is approximately equal to the correlation between two images.

Example: Murase and Nayar, 1996

Database of objects. No background clutter or occlusion



Murase and Nayar, 1996

- *Acquire* models of object appearances using a turntable

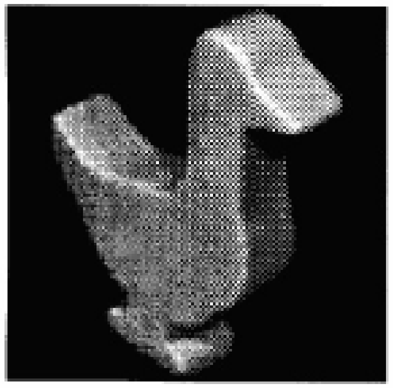
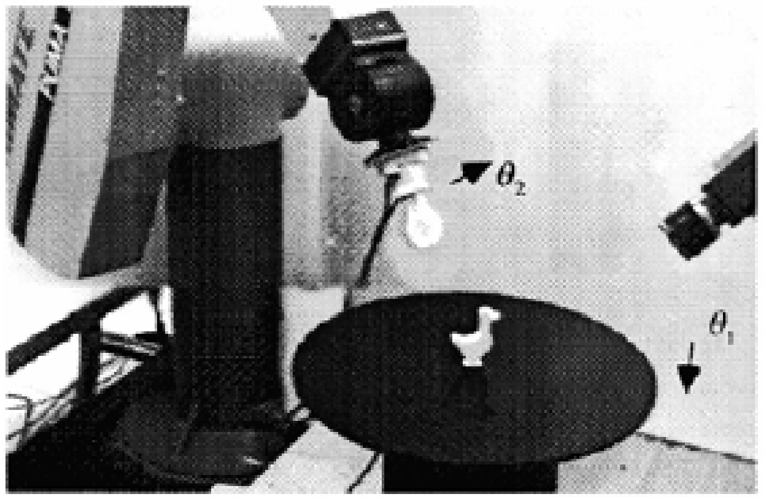
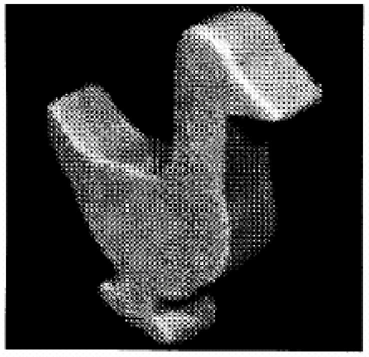
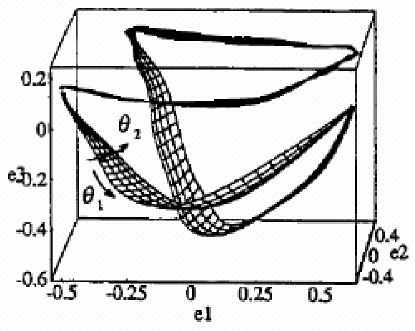



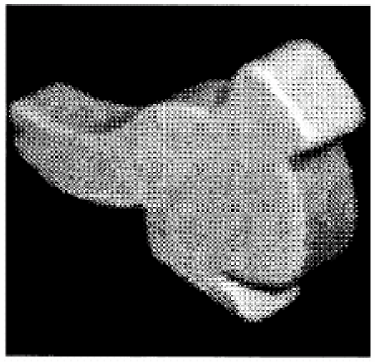
Figure from S. K. Nayar, et al, "Parametric Appearance Representation" 1996

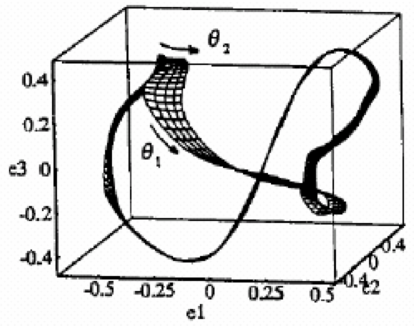
Appearance Manifolds


A

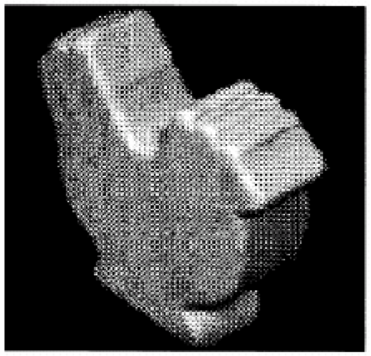


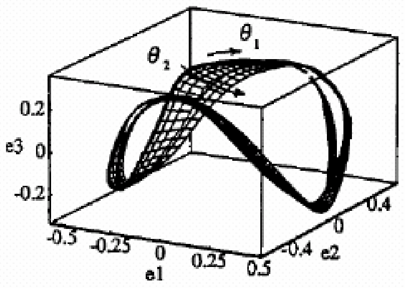
A


B

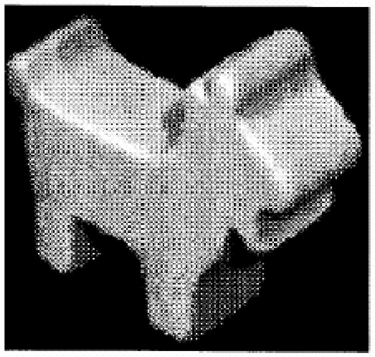


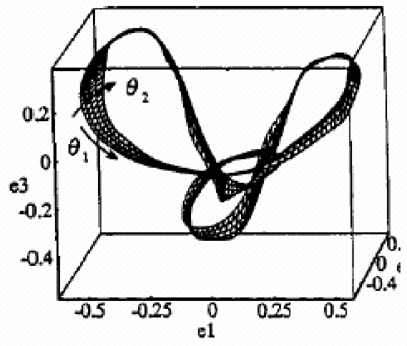
B


C



C

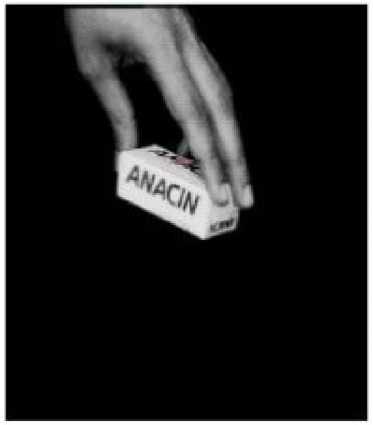

D

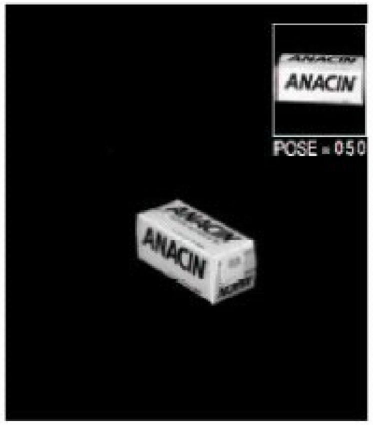


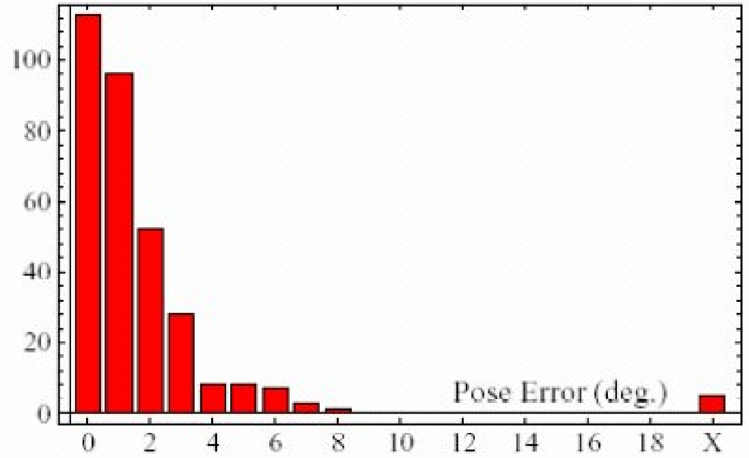
D

[Murase & Nayar, 1996]

Appearance Manifolds



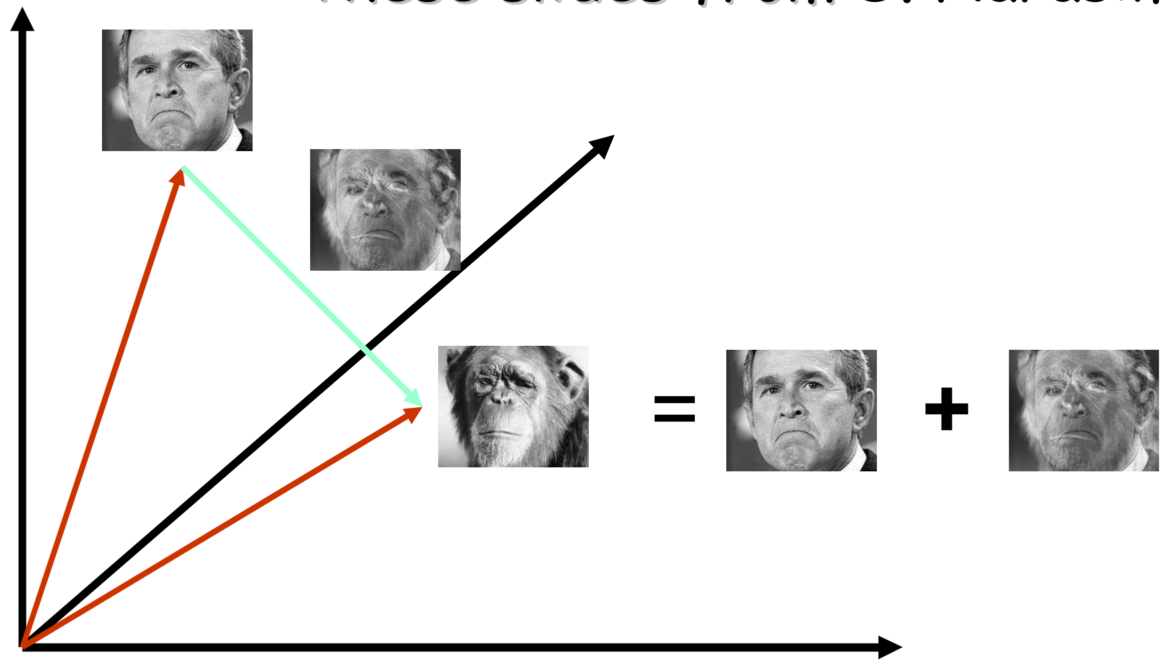




Murase & Nayar

Example: EigenFaces

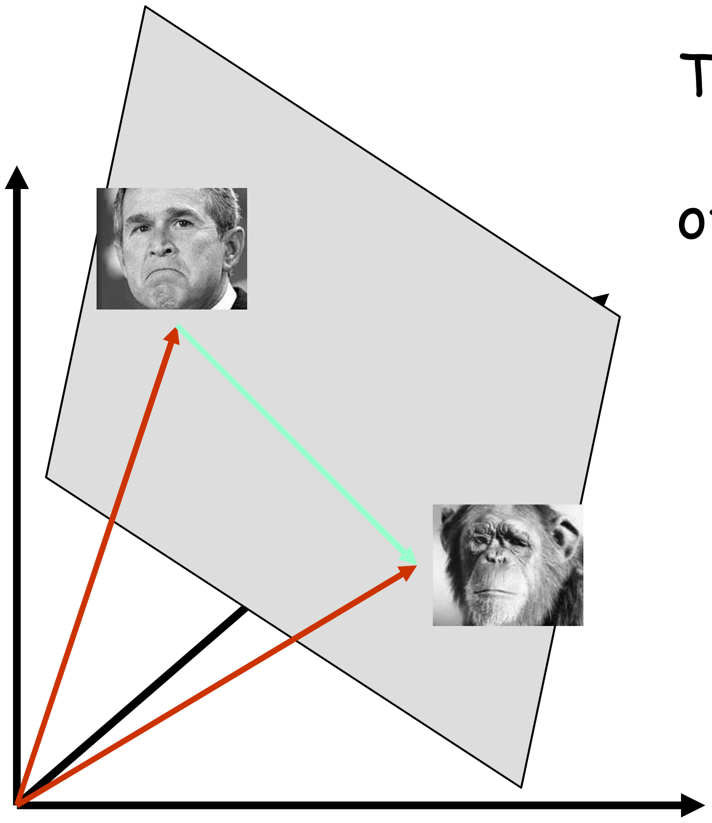
These slides from S. Narasimhan, CMU



- An image is a point in a high dimensional space
  - An N x M image is a point in R^{NM}
  - We can define vectors in this space as we did in the 2D case

[Thanks to Chuck Dyer, Steve Seitz, Nishino]

Dimensionality Reduction



The set of faces is a "subspace" of the set of images

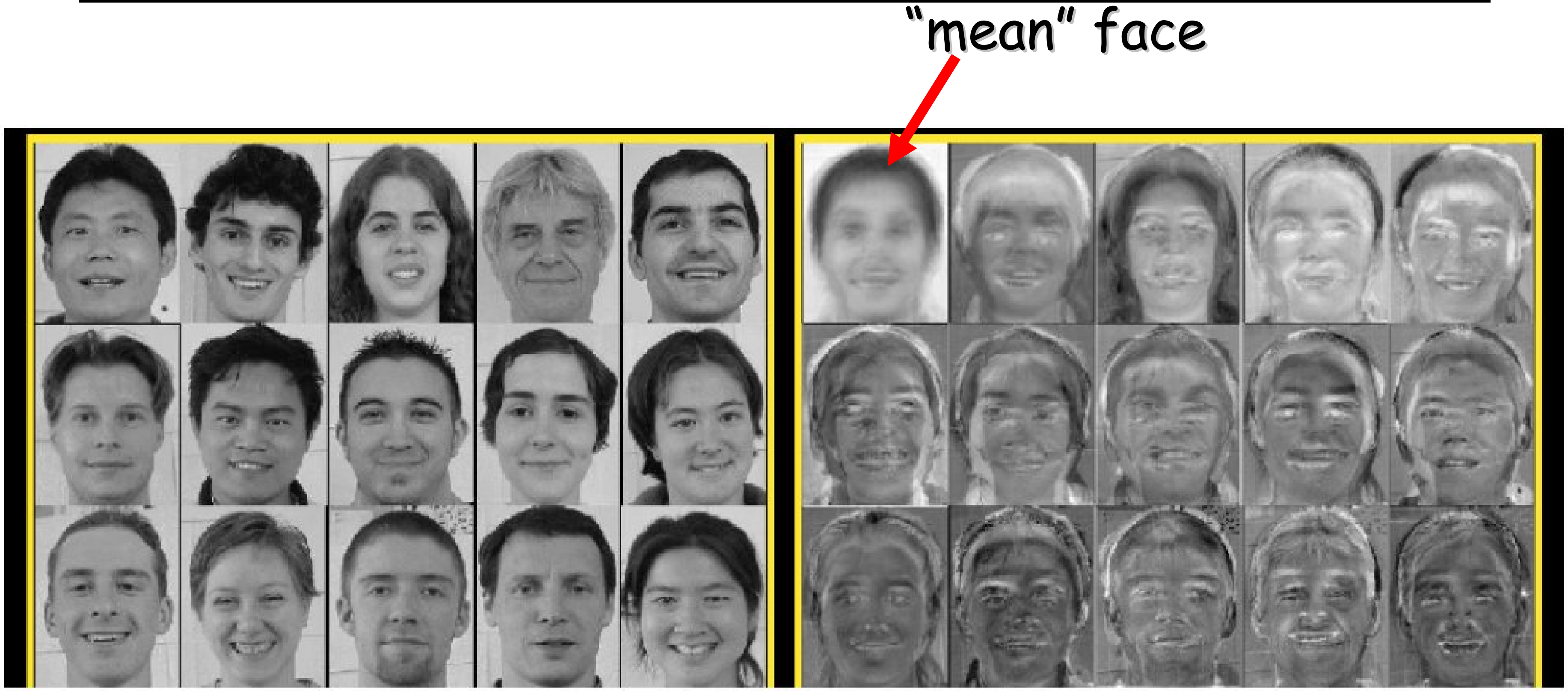
- Suppose it is K dimensional
- We can find the best subspace using PCA
- This is like fitting a "hyper-plane" to the set of faces
  - spanned by vectors $\mathbf{v_1}$, $\mathbf{v_2}$, ..., $\mathbf{v_K}$

Any face: $\mathbf{x} \approx \overline{\mathbf{x}} + a_1\mathbf{v_1} + a_2\mathbf{v_2} + \ldots + a_k\mathbf{v_k}$

Generating Eigenfaces – in words

1. Large set of images of human faces is taken.
2. The images are normalized to line up the eyes, mouths and other features.
3. The eigenvectors of the covariance matrix of the face image vectors are then extracted.
4. These eigenvectors are called eigenfaces.

Eigenfaces



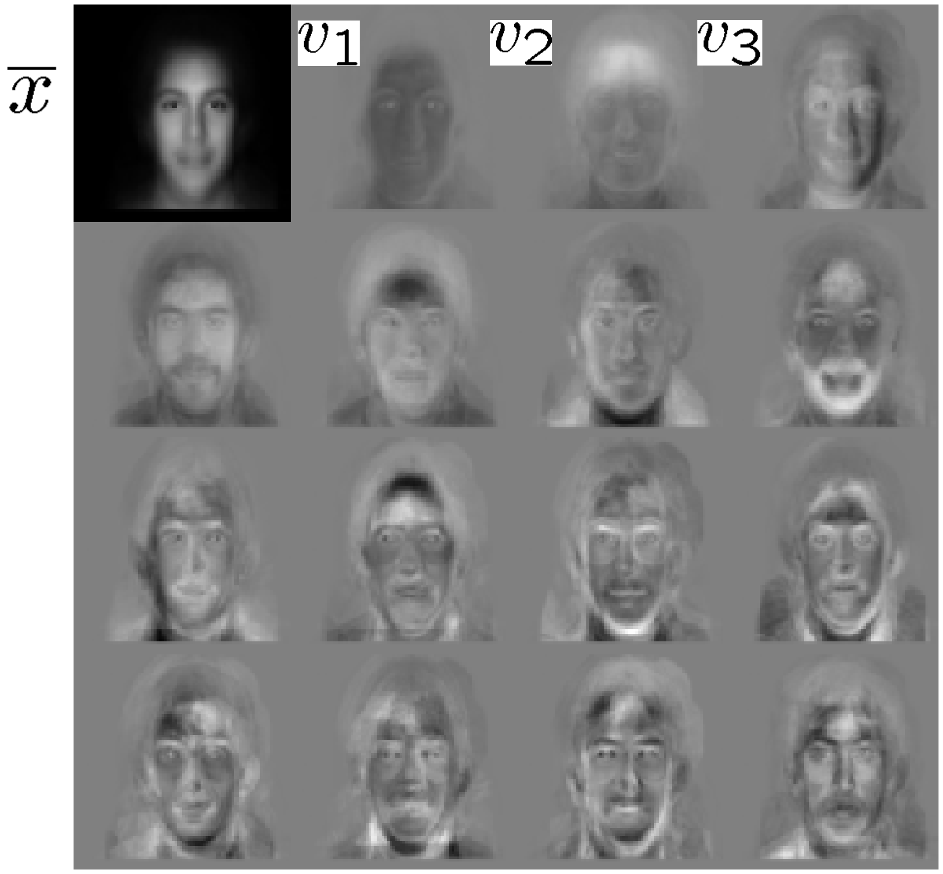
Eigenfaces look somewhat like generic faces.

Eigenfaces for Face Recognition

- When properly weighted, eigenfaces can be summed together to create an approximate gray-scale rendering of a human face.
- Remarkably few eigenvector terms are needed to give a fair likeness of most people's faces.
- Hence eigenfaces provide a means of applying data compression to faces for identification purposes.

Eigenfaces

- PCA extracts the eigenvectors of **A**
  - Gives a set of vectors $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \ldots$
  - Each one of these vectors is a direction in face space
    - what do these look like?

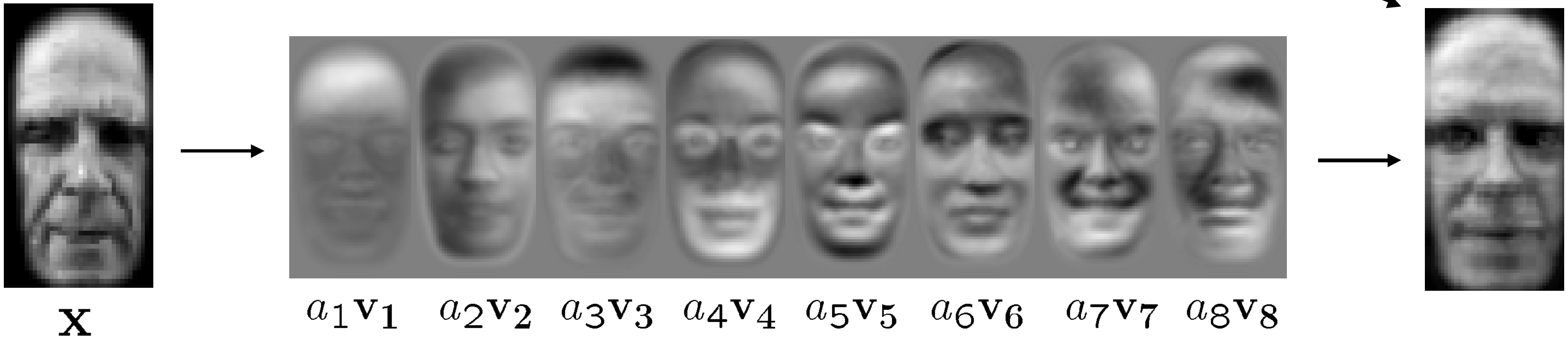


Projecting onto the Eigenfaces

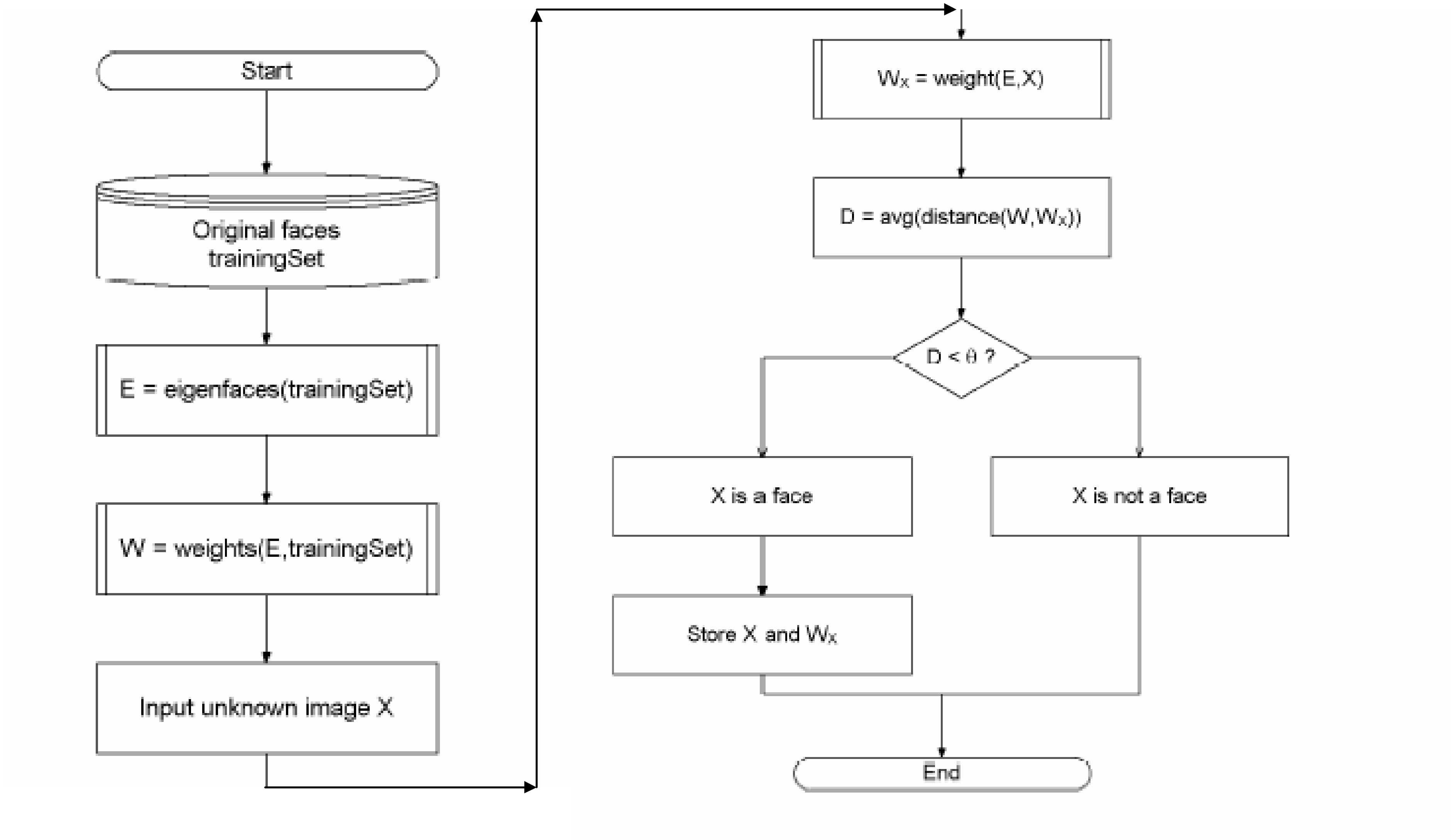
- The eigenfaces **v**$_1$, ..., **v**$_K$ span the space of faces
  - A face is converted to eigenface coordinates by

$$\mathbf{x} \rightarrow (\underbrace{(\mathbf{x}-\overline{\mathbf{x}})\cdot\mathbf{v}_1}_{a_1},\ \underbrace{(\mathbf{x}-\overline{\mathbf{x}})\cdot\mathbf{v}_2}_{a_2},\ldots,\ \underbrace{(\mathbf{x}-\overline{\mathbf{x}})\cdot\mathbf{v}_\mathrm{K}}_{a_K})$$

$$\mathbf{x} \approx \overline{\mathbf{x}} + a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \ldots + a_K\mathbf{v}_\mathrm{K}$$



Is this a face or not?



Start
Original faces trainingSet
E = eigenfaces(trainingSet)
W = weights(E,trainingSet)
Input unknown image X
W_X = weight(E,X)
D = avg(distance(W,W_X))
D < θ ?
X is a face
X is not a face
Store X and W_X
End

Figure 1: High-level functioning principle of the eigenface-based facial recognition algorithm

Recognition with Eigenfaces

- Algorithm

1. Process the image database (set of images with labels)
   - Run PCA—compute eigenfaces
   - Calculate the K coefficients for each image

2. Given a new image (to be recognized) **x**, calculate K coefficients

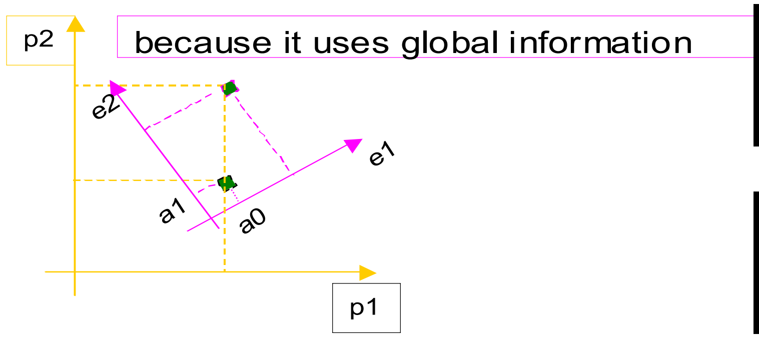
$$\mathbf{x} \rightarrow (a_1, a_2, \ldots, a_K)$$

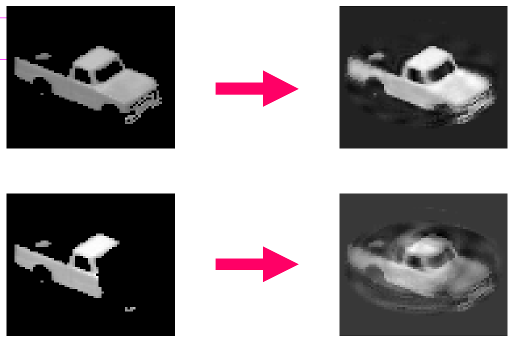
3. Detect if x is a face

$$\|\mathbf{x} - (\overline{\mathbf{x}} + a_1\mathbf{v_1} + a_2\mathbf{v_2} + \ldots + a_K\mathbf{v_K})\| < \text{threshold}$$

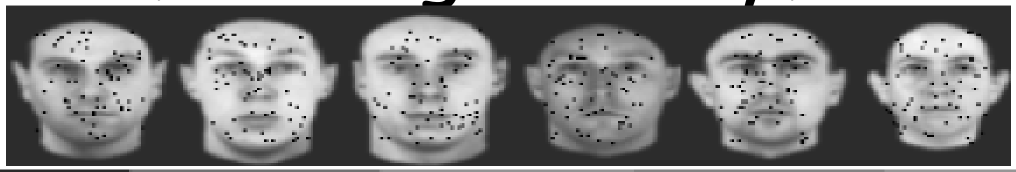
4. If it is a face, who is it?
   - Find closest labeled face in database
     - nearest-neighbor in K-dimensional space

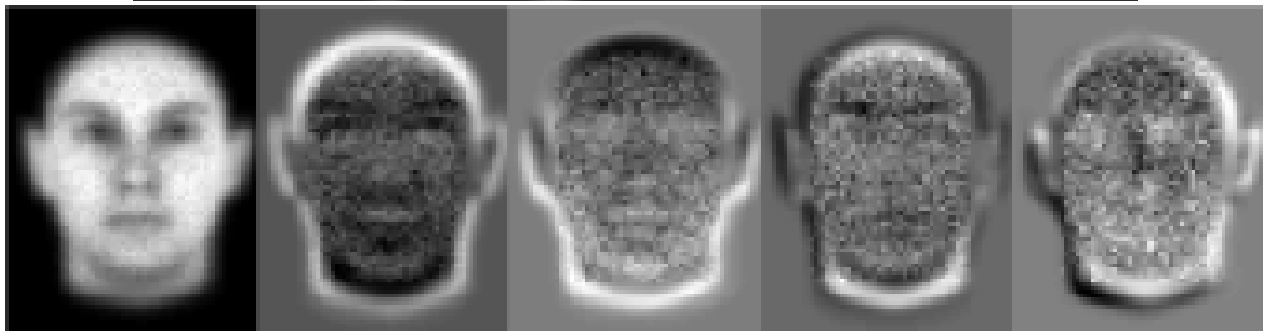
Cautionary Note:
PCA has problems with occlusion





and also, more generally, with outliers





PCA