Eigenvalues and Eigenvectors for Image Processing — Documentation

Exported 9/11/2025 · Presented by Mejbah Ahammad

Mathematical Definition

Mathematical Definition

For a square matrix A, an eigenvector $\mathbf{v}$ and eigenvalue λ satisfy:

$$A\mathbf{v} = \lambda\mathbf{v}$$

This can also be written as:

$$(A - \lambda I)\mathbf{v} = \mathbf{0}$$

Where:

- A is a square matrix of size $n \times n$
- $\mathbf{v} \neq \mathbf{0}$ is the eigenvector
- λ is the corresponding eigenvalue
- I is the identity matrix

Finding Eigenvalues

Finding Eigenvalues

To find eigenvalues, we solve the characteristic equation:

$$\det(A - \lambda I) = 0$$

For a 2×2 matrix $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$:

$$\det\begin{bmatrix} a-\lambda & b \\ c & d-\lambda \end{bmatrix} = (a-\lambda)(d-\lambda) - bc = 0$$

This expands to the characteristic polynomial:

$$\lambda^2 - (a+d)\lambda + (ad - bc) = 0$$

The roots of this polynomial are the eigenvalues λ_1 and λ_2.

PCA in Image Processing

PCA in Image Processing

PCA uses eigenvalues and eigenvectors to:

- Find the directions of maximum variance in data
- Reduce dimensionality while preserving important information
- Compress images efficiently

The covariance matrix C is computed, and its eigenvectors represent the principal components:

$$C = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T$$

Where $\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i$ is the mean vector.

The principal components are found by solving:

$$C\mathbf{v}_i = \lambda_i \mathbf{v}_i$$

Where $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$ are the eigenvalues in descending order.

Key Applications in Image Processing

Key Applications in Image Processing

- **Principal Component Analysis (PCA) -** Dimensionality reduction
- **Image Compression -** Efficient storage and transmission
- **Feature Extraction -** Pattern recognition
- **Image Denoising -** Noise reduction techniques
- **Edge Detection -** Boundary identification
- **Texture Analysis -** Surface characterization

Step 1: Raw Image

Step 1: Raw Image

This is the original input image that we will process with PCA.

Step 2: Convert to Grayscale

Step 2: Convert to Grayscale

We convert RGB to intensity using the luminance formula:

$$I = 0.299, R + 0.587, G + 0.114, B$$

RGB

Grayscale (visualized)

Step 3: Vectorize and Center

Step 3: Vectorize and Center

Flatten image (or patches) into vectors and subtract the mean.

$$\mathbf{x}_i \in \mathbb{R}^d, \quad \boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i, \quad \tilde{\mathbf{x}}_i = \mathbf{x}_i - \boldsymbol{\mu}$$

Form the data matrix of centered vectors:

$$\tilde{X} = \begin{bmatrix} \tilde{\mathbf{x}}_1 & \tilde{\mathbf{x}}_2 & \cdots & \tilde{\mathbf{x}}_n \end{bmatrix}^T \in \mathbb{R}^{n\times d}$$

Raw Calculations (2D example)

Raw Calculations (2D example)

Use five 2D samples (e.g., two channels/features per pixel):

$$X = \begin{bmatrix} 52 & 55\ 61 & 59\ 63 & 62\ 59 & 80\ 55 & 52 \end{bmatrix}$$

Mean vector:

$$\boldsymbol{\mu} = \frac{1}{5}\sum_{i=1}^{5}\mathbf{x}_i = \begin{bmatrix} 58 \\ 61.6 \end{bmatrix}$$

Centered data (each row minus $\boldsymbol{\mu}$):

$$\tilde{X} = \begin{bmatrix} -6 & -6.6\ 3 & -2.6\ 5 & 0.4\ 1 & 18.4\ -3 & -9.6 \end{bmatrix}$$

Covariance

Covariance

Compute $C = \frac{1}{n-1}\tilde{X}^T\tilde{X}$ with $n = 5$:

$$C = \frac{1}{4}\begin{bmatrix} 80 & 81 \\ 81 & 481.2 \end{bmatrix} = \begin{bmatrix} 20 & 20.25 \\ 20.25 & 120.3 \end{bmatrix}$$

Eigenvalues of a symmetric 2×2 matrix $\begin{bmatrix} a & b \\ b & d \end{bmatrix}$:

$$\lambda_{1,2} = \frac{(a+d)\pm\sqrt{(a-d)^2+4b^2}}{2}$$

With $a = 20,\ b = 20.25,\ d = 120.3$:

$$\lambda_1 \approx 124.235, \qquad \lambda_2 \approx 16.065$$

Eigenvectors and Projection

Eigenvectors and Projection

Unit eigenvectors (approx.):

$$\mathbf{v}_1 \approx \begin{bmatrix} 0.191 \\ 0.981 \end{bmatrix}, \qquad \mathbf{v}_2 \approx \begin{bmatrix} 0.982 \\ -0.191 \end{bmatrix}$$

Project first centered sample $\tilde{\mathbf{x}}_1 = [-6, -6.6]^T$ onto $\mathbf{v}_1$:

$$z_1 = \mathbf{v}_1^T\tilde{\mathbf{x}}_1 \approx 0.191(-6) + 0.981(-6.6) \approx -7.621$$

Reconstruction from Top-1

Reconstruction from Top-1

Using $\hat{\mathbf{x}} \approx \mathbf{v}_1 z + \boldsymbol{\mu}$:

$$\hat{\mathbf{x}}_1 \approx \begin{bmatrix} 0.191 \\ 0.981 \end{bmatrix} (-7.621) + \begin{bmatrix} 58 \\ 61.6 \end{bmatrix} = \begin{bmatrix} -1.456 \\ -7.477 \end{bmatrix} + \begin{bmatrix} 58 \\ 61.6 \end{bmatrix} = \begin{bmatrix} 56.544 \\ 54.123 \end{bmatrix}$$

This shows how a single principal component approximates the original sample.

Step 4: Covariance and Eigenpairs

Step 4: Covariance and Eigenpairs

$$C = \frac{1}{n-1} \tilde{X}^T \tilde{X} \in \mathbb{R}^{d \times d}$$

$$C\mathbf{v}_k = \lambda_k \mathbf{v}_k, \quad \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$$

Step 5: Project

Step 5: Project

Select matrix of top-k eigenvectors $V_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k]$.

$$\mathbf{z}_i = V_k^T \tilde{\mathbf{x}}_i \in \mathbb{R}^k$$

Low-dimensional representation $mathbf{z}_i$ captures most variance for small k.

Step 6: Reconstruct from Top-k

Step 6: Reconstruct from Top-k

$$\hat{\mathbf{x}}_i = V_k \mathbf{z}_i + \boldsymbol{\mu} = V_k V_k^T \tilde{\mathbf{x}}_i + \boldsymbol{\mu}$$

Below we visualize the idea of reconstruction. (For demo, we reuse the image.)

Original

Reconstruction (k components)

Step-by-Step Recap

Step-by-Step Recap

1. Load raw image
2. Convert to grayscale (luminance)
3. Vectorize pixels (or patches) and mean-center
4. Compute covariance matrix
5. Find eigenvalues/eigenvectors and sort
6. Project onto top-k components
7. Reconstruct approximate image

These steps demonstrate how eigenvalues and eigenvectors power PCA-based image compression.

Basic PCA Implementation

Basic PCA Implementation

Step 1: Data Preparation Copy

```
# Reshape image into matrix
def prepare_image_data(image):
    height, width, channels = image.shape
    # Flatten image into 2D matrix
    matrix = image.reshape(height * width, channels)
    return matrix, height, width, channels
```

Step 2: Center the Data Copy

```
# Calculate mean and center the data
def center_data(matrix):
    mean = np.mean(matrix, axis=0)
    centered = matrix - mean
    return centered, mean
```

Step 3: Compute Covariance Matrix Copy

```
# Compute covariance matrix
def compute_covariance(centered_data):
    n = centered_data.shape[0]
    covariance = np.dot(centered_data.T, centered_data) / (n - 1)
    return covariance
```

Step 4: Find Eigenvalues and Eigenvectors Copy

```
# Find eigenvalues and eigenvectors
def find_eigencomponents(covariance_matrix):
    eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
    # Sort by eigenvalues in descending order
    idx = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]
    return eigenvalues, eigenvectors
```

Step 5: PCA Compression Copy

```
# Complete PCA compression function
def pca_compress_image(image, num_components):
    # Prepare data
    matrix, height, width, channels = prepare_image_data(image)

    # Center the data
    centered, mean = center_data(matrix)
```

```
    # Compute covariance
    covariance = compute_covariance(centered)

    # Find eigencomponents
    eigenvalues, eigenvectors = find_eigencomponents(covariance)

    # Select top components
    top_components = eigenvectors[:, :num_components]

    # Project and reconstruct
    projected = np.dot(centered, top_components)
    reconstructed = np.dot(projected, top_components.T) + mean

    # Reshape back to image
    compressed_image = reconstructed.reshape(height, width, channels)

    return compressed_image, eigenvalues[:num_components]
```

Hessian Matrix for Edge Detection

Hessian Matrix for Edge Detection

For edge detection, we use the Hessian matrix of image intensity:

$$H = \begin{bmatrix} I_{xx} & I_{xy} \\ I_{yx} & I_{yy} \end{bmatrix}$$

Where $I_{xx}, I_{xy}, I_{yx}, I_{yy}$ are second-order partial derivatives.

The eigenvalues of H are:

$$\lambda_{1,2} = \frac{I_{xx} + I_{yy}}{2} \pm \frac{\sqrt{(I_{xx} - I_{yy})^2 + 4I_{xy}^2}}{2}$$

Eigenvalues indicate:

- Large eigenvalues: Strong edges and corners
- Small eigenvalues: Smooth regions
- One large, one small: Edge features
- Both large: Corner features

Eigenvalues in Texture Analysis

Eigenvalues in Texture Analysis

Texture analysis uses eigenvalues to characterize surface properties:

- **Co-occurrence Matrix** - Statistical texture measures
- **Local Binary Patterns** - Pattern recognition
- **Gabor Filters** - Frequency domain analysis

Eigenvalues provide rotation-invariant texture descriptors that are robust to image transformations.

Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD)

For any matrix A of size $m \times n$, SVD decomposes it as:

$$A = U\Sigma V^T$$

Where:

- U is $m \times m$ orthogonal matrix (left singular vectors)
- Σ is $m \times n$ diagonal matrix (singular values)
- V is $n \times n$ orthogonal matrix (right singular vectors)

The singular values σ_i are related to eigenvalues:

$$\sigma_i = \sqrt{\lambda_i(A^T A)}$$

Efficiency and Optimization

Efficiency and Optimization

- **Power Iteration** - For largest eigenvalue
- **QR Algorithm** - For all eigenvalues
- **Lanczos Method** - For sparse matrices
- **Parallel Processing** - GPU acceleration

Modern implementations use optimized libraries like LAPACK, BLAS, and CUDA for high-performance computing.

Summary

Summary

Eigenvalues and eigenvectors are fundamental tools in image processing, enabling:

- Efficient data compression and dimensionality reduction
- Robust feature extraction and pattern recognition
- Advanced image analysis techniques

Future Directions

- Deep learning integration
- Real-time processing optimization
- Quantum computing applications

Thank you for your attention!