ML A# 4 | MSDS25030 | Atique Ahmad

ICA and PCA Implementation Assignment Report

Git Repository: https://github.com/atique-ahmad-01/natural-image-pca-ica.git

Introduction

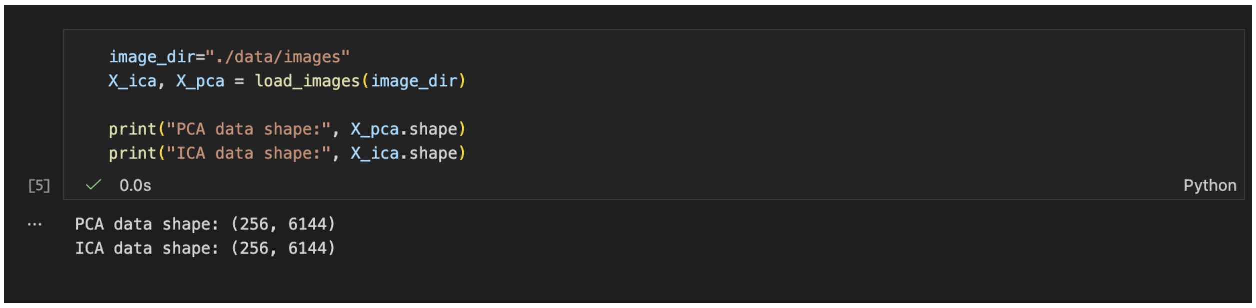
This assignment implements two fundamental unsupervised learning algorithms - Independent Component Analysis (ICA) and Principal Component Analysis (PCA) - to discover visual features from natural images. The goal is to extract meaningful filters that represent how images can be decomposed into basic building blocks.

Dataset Description

For this assignment, I worked with natural images to extract image patches:

- **Source**: 4 natural images stored in the data/images directory
- **Patch Extraction**: From each image, I extracted non-overlapping 16×16 pixel patches
- **Total Patches**: 6,144 image patches were collected
- **Image Preprocessing**: RGB color images were converted to grayscale by averaging the three color channels
- **Data Representation**: Each 16×16 patch was flattened into a 256-dimensional vector

The final dataset consists of 256 features (representing pixel positions) across 6,144 samples (individual patches).



```
image_dir="./data/images"
X_ica, X_pca = load_images(image_dir)

print("PCA data shape:", X_pca.shape)
print("ICA data shape:", X_ica.shape)
```

```
PCA data shape: (256, 6144)
ICA data shape: (256, 6144)
```

Methodology

Part 1: Data Preprocessing

Before applying ICA and PCA, the data needed different preprocessing steps:

Preprocessing for ICA:

The ICA algorithm requires whitened data as input. Whitening is a transformation that decorrelates the data and gives all dimensions equal variance. The steps I implemented were:

1. Computed the covariance matrix of all image patches
2. Performed eigendecomposition on the covariance matrix
3. Created a whitening matrix using the eigenvectors and eigenvalues
4. Mean-centered the data by subtracting the average patch
5. Applied the whitening transformation
6. Scaled the result by a factor of 2 for numerical stability

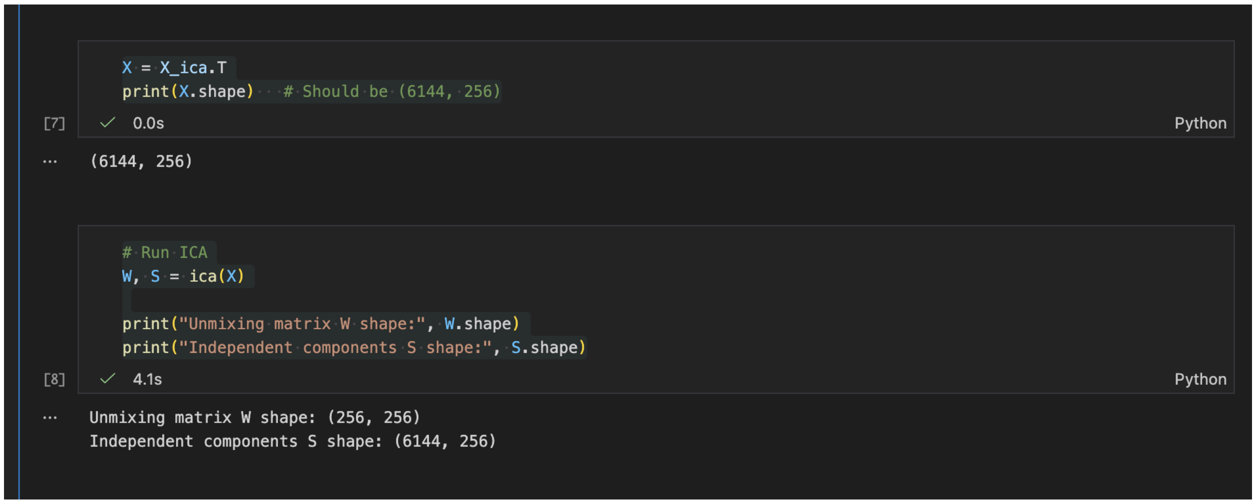
Preprocessing for PCA:

PCA works best with standardized data. The preprocessing steps were:

1. Mean-centered the data by subtracting the mean of each feature
2. Standardized each feature by dividing by its standard deviation
3. Handled edge cases where standard deviation might be zero

Part 2: ICA Implementation

I implemented the FastICA algorithm, which is an efficient method for performing Independent Component Analysis:



```
X = X_ica.T
print(X.shape)   # Should be (6144, 256)
```

```
[7] ✓ 0.0s                                                    Python
```

```
(6144, 256)
```

```
# Run ICA
W, S = ica(X)

print("Unmixing matrix W shape:", W.shape)
print("Independent components S shape:", S.shape)
```

```
[8] ✓ 4.1s                                                    Python
```

```
Unmixing matrix W shape: (256, 256)
Independent components S shape: (6144, 256)
```

Algorithm Steps:

1. **Initialization**: Created a random 256×256 weight matrix to start the iterative process

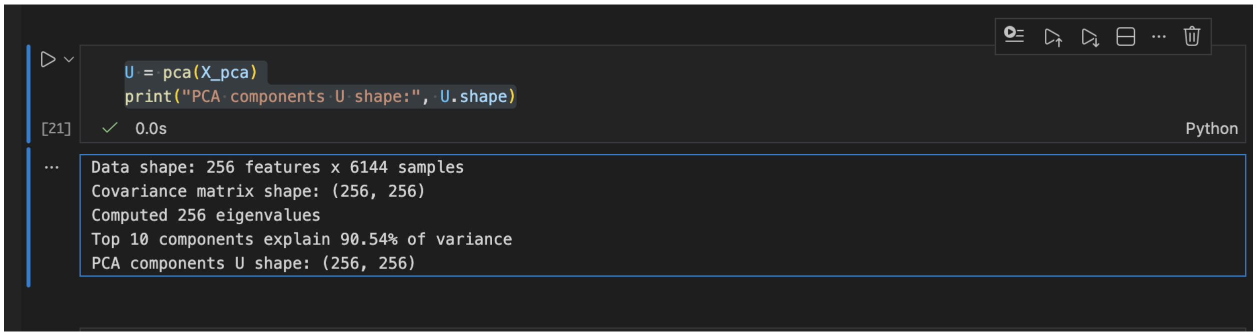
2. **Iterative Updates**: For each iteration:
    - Computed the projection of data through current weights
    - Applied the tanh nonlinear function to find non-Gaussian directions
    - Updated the weight matrix based on the gradient
    - Applied symmetric decorrelation to keep components independent
3. **Decorrelation**: After each update, I used symmetric decorrelation to ensure the weight vectors remain orthogonal to each other

4. **Convergence Check**: The algorithm stops when the change between iterations is less than 0.00001 or after 200 iterations maximum

5. **Output**: The final unmixing matrix W (256×256) represents 256 independent component filters

Part 3: PCA Implementation

Principal Component Analysis finds orthogonal directions of maximum variance in the data:



```
U = pca(X_pca)
print("PCA components U shape:", U.shape)
```

```
Data shape: 256 features x 6144 samples
Covariance matrix shape: (256, 256)
Computed 256 eigenvalues
Top 10 components explain 90.54% of variance
PCA components U shape: (256, 256)
```

Algorithm Steps:

1. **Covariance Computation**: Calculated the 256×256 covariance matrix showing how pixel positions vary together

2. **Eigendecomposition**: Found the eigenvalues and eigenvectors of the covariance matrix using NumPy's efficient symmetric matrix solver

3. **Sorting**: Arranged the eigenvectors in descending order of their eigenvalues, so the first component captures the most variance

4. **Variance Analysis**: Computed that the top 10 principal components explain 90.54% of the total variance in the data

5. **Output**: The matrix U (256×256) where each column is a principal component filter

Part 4: Visualization

To visualize the learned filters, I created grid displays:

For ICA Filters:

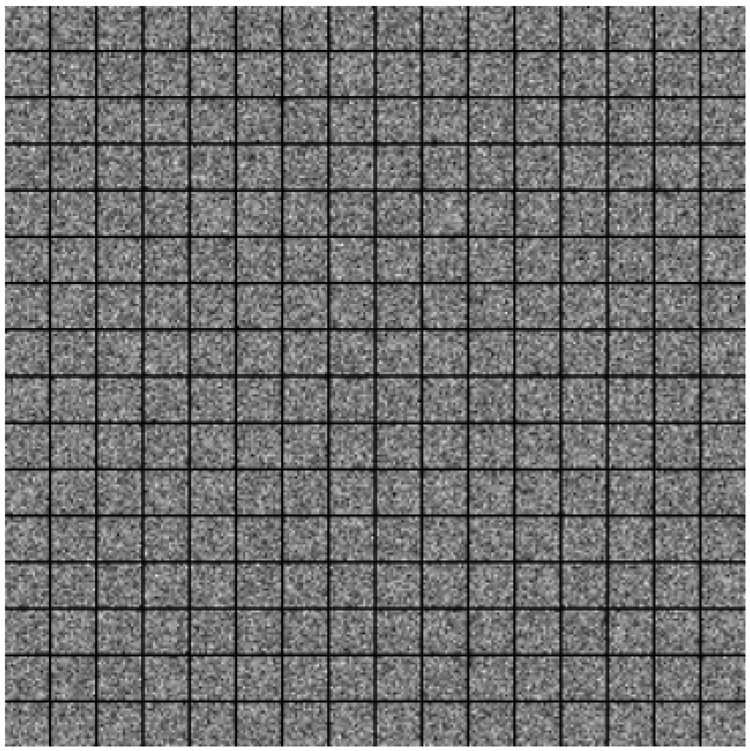
- Sorted all 256 filters by their magnitude (2-norm)
- Arranged them in a 16×16 grid with spacing between patches
- Each small square shows one learned filter reshaped back to 16×16 pixels

For PCA Filters:

- Displayed all 256 principal components in order of importance
- Arranged them in a 16×16 grid with spacing
- Each position shows one component reshaped to 16×16 pixels

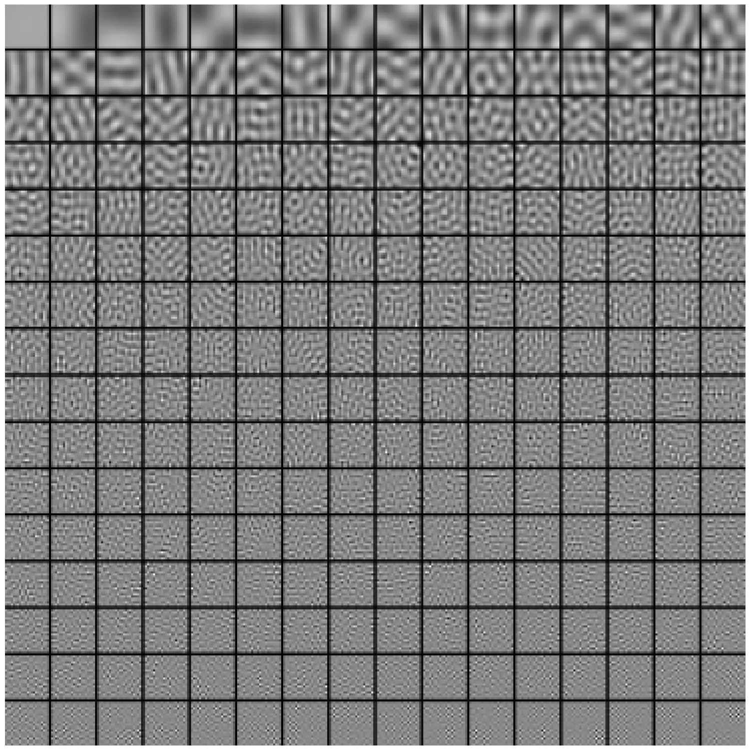
Results

Figure 1: ICA Filters



The ICA filters show localized, edge-like features at different orientations and positions. These filters resemble what neuroscientists have discovered in the primary visual cortex of the human brain. Each filter responds to specific edge orientations, making them useful for detecting lines and boundaries in images.

Figure 2: PCA Filters



The PCA filters display a different pattern. The early components (top-left) show smooth, low-frequency variations representing overall brightness patterns. As we move through the components, they capture increasingly fine details and high-frequency textures. Unlike ICA, PCA components are globally distributed across the entire patch.

Analysis and Observations

Key Differences Between ICA and PCA:

1. **Locality**: ICA filters are localized (active in small regions), while PCA filters are global (spread across the entire patch)

2. **Biological Plausibility**: ICA filters resemble actual receptive fields in mammalian visual systems, suggesting this is how the brain might process visual information

3. **Independence vs. Orthogonality**: ICA seeks statistically independent components, while PCA finds orthogonal components with maximum variance

4. **Interpretability**: ICA produces edge detectors that are easy to interpret as "feature detectors," while PCA components represent variance modes

Performance Metrics:

- The top 10 PCA components capture 90.54% of the data's variance, showing that natural images have strong low-dimensional structure
- ICA successfully converged within the 200-iteration limit
- Both algorithms processed 6,144 samples with 256 features efficiently

Conclusion

This assignment successfully implemented both ICA and PCA algorithms from scratch using only NumPy. The results demonstrate that:

1. Natural images can be decomposed into meaningful visual features
2. ICA discovers localized edge-like filters similar to biological vision systems
3. PCA finds global patterns ordered by variance
4. Both methods reveal different aspects of the underlying structure in natural images

The implementation shows how unsupervised learning can discover interpretable representations without any labeled training data, simply by analyzing the statistical properties of natural images.

Code Structure

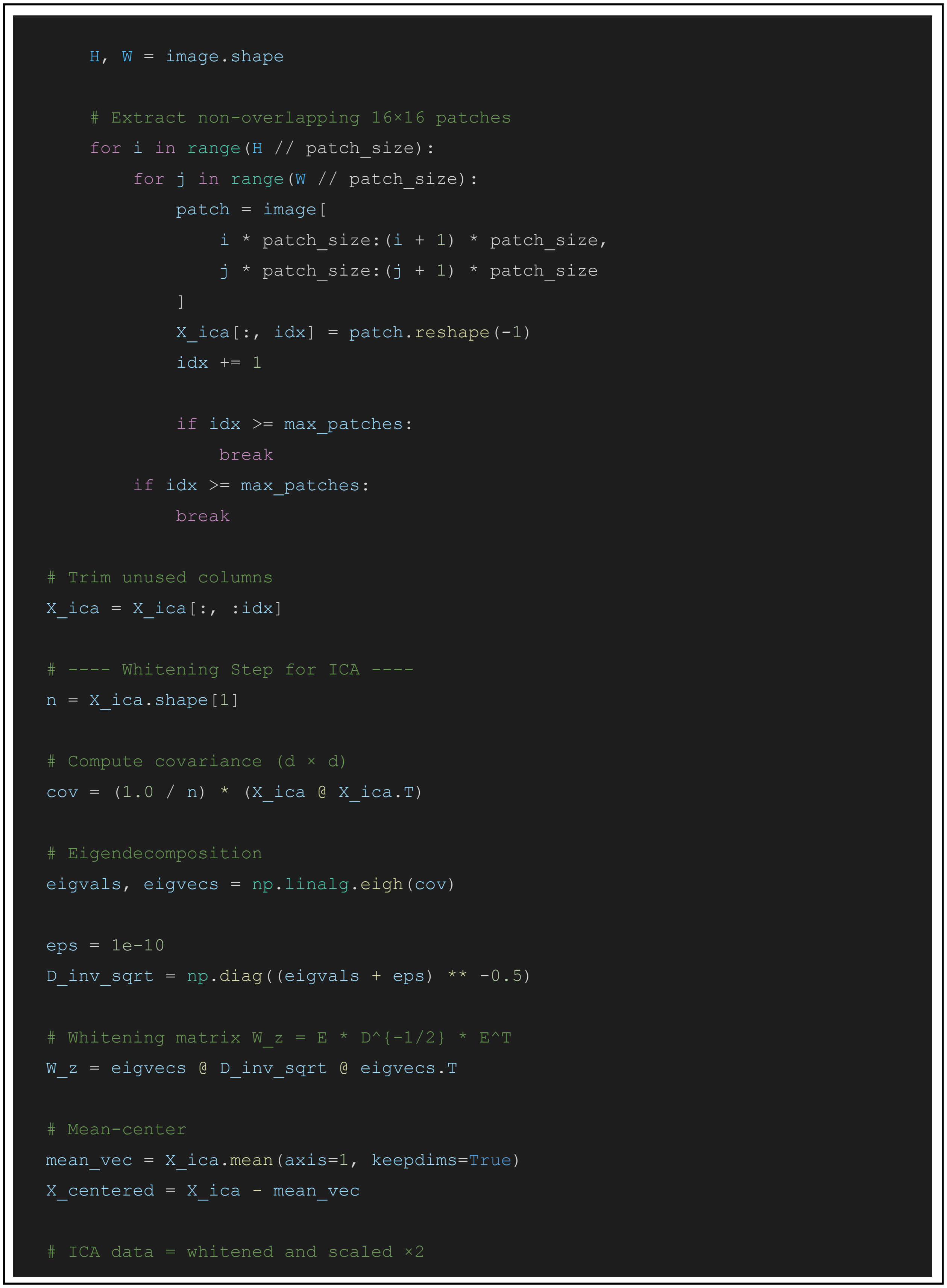
The implementation consists of:

- `load_images()`: Loads images and prepares data for both algorithms
- `ica()`: FastICA algorithm implementation
- `pca()`: Principal Component Analysis implementation
- `plot_ica_filters()`: Visualization for ICA results
- `plot_pca_filters()`: Visualization for PCA results

```
# Cell 1: Imports and globals
```

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import imread
# from imageio.v2 import imread  # or from matplotlib.pyplot import imread

patch_size = 16
W_z = None
```

```
# Cell 2: load_images

def load_images(image_dir="images"):
   """
   Load image patches and compute whitened data for ICA and standardized data for
PCA.

   Parameters
   ----------
   image_dir : str
       Directory containing images named 1.jpg, 2.jpg, 3.jpg, 4.jpg

   Returns
   -------
   X_ica : np.ndarray
       Whitened and mean-centered data for ICA (shape: [patch_size^2, num_patches])
   X_pca : np.ndarray
       Standardized data for PCA (shape: [patch_size^2, num_patches])
   """
   global patch_size, W_z

   max_patches = 40000
   X_ica = np.zeros((patch_size * patch_size, max_patches), dtype=float)

   idx = 0
   for img_idx in range(1, 5):
       img_path = f"{image_dir}/{img_idx}.jpg"
       image = imread(img_path).astype(float)

       # Convert RGB → grayscale by averaging channels
       if image.ndim == 3:
           image = image.mean(axis=2)
```



```python
    H, W = image.shape

    # Extract non-overlapping 16×16 patches
    for i in range(H // patch_size):
        for j in range(W // patch_size):
            patch = image[
                i * patch_size:(i + 1) * patch_size,
                j * patch_size:(j + 1) * patch_size
            ]
            X_ica[:, idx] = patch.reshape(-1)
            idx += 1

            if idx >= max_patches:
                break
        if idx >= max_patches:
            break

# Trim unused columns
X_ica = X_ica[:, :idx]

# ---- Whitening Step for ICA ----
n = X_ica.shape[1]

# Compute covariance (d × d)
cov = (1.0 / n) * (X_ica @ X_ica.T)

# Eigendecomposition
eigvals, eigvecs = np.linalg.eigh(cov)

eps = 1e-10
D_inv_sqrt = np.diag((eigvals + eps) ** -0.5)

# Whitening matrix W_z = E * D^{-1/2} * E^T
W_z = eigvecs @ D_inv_sqrt @ eigvecs.T

# Mean-center
mean_vec = X_ica.mean(axis=1, keepdims=True)
X_centered = X_ica - mean_vec

# ICA data = whitened and scaled ×2
```

```
    X_ica_whitened = 2.0 * (W_z @ X_centered)

    # PCA data = standardized per feature
    std_vec = X_centered.std(axis=1, keepdims=True)
    std_vec[std_vec == 0] = 1.0  # avoid divide-by-zero

    X_pca_standardized = X_centered / std_vec

    return X_ica_whitened, X_pca_standardized
```

```
image_dir="./data/images"
X_ica, X_pca = load_images(image_dir)

print("PCA data shape:", X_pca.shape)
print("ICA data shape:", X_ica.shape)
```

```
# Cell 3: ICA implementation (corrected)
def ica(X, max_iter=200, tol=1e-5):
    """
    Perform ICA using FastICA algorithm (pure NumPy version).

    Parameters
    ----------
    X : np.ndarray
        Whitened data matrix of shape (n_samples, n_features)
    max_iter : int
        Maximum number of iterations
    tol : float
        Convergence tolerance

    Returns
    -------
    W : np.ndarray
        Unmixing matrix of shape (n_features, n_features)
    S : np.ndarray
        Estimated independent components, shape (n_samples, n_features)
    """
    n_samples, n_features = X.shape

    # Initialize random weights
    W = np.random.randn(n_features, n_features)

    # Decorrelate function
    def decorrelate(W):
        # Symmetric decorrelation
        s, u = np.linalg.eigh(W @ W.T)
        return (u @ np.diag(1.0 / np.sqrt(s)) @ u.T) @ W

    W = decorrelate(W)

    for i in range(max_iter):
        W_old = W.copy()

        # FastICA update
        WX = X @ W.T
        g = np.tanh(WX)
        g_prime = 1 - g ** 2
        W = (X.T @ g) / n_samples - np.mean(g_prime, axis=0) * W

        # Decorrelate
        W = decorrelate(W)

        # Check convergence
        lim = np.max(np.abs(np.abs(np.diag(W @ W_old.T)) - 1))
        if lim < tol:
            break

    # Estimated independent components
    S = X @ W.T
    return W, S
```

```
X = X_ica.T
print(X.shape)   # Should be (6144, 256)
```

```
# Run ICA
W, S = ica(X)

print("Unmixing matrix W shape:", W.shape)
print("Independent components S shape:", S.shape)
```

```
# Cell 4: PCA function stub

def pca(X):
    """
    Principal Component Analysis

    Parameters
    ----------
    X : np.ndarray
        Data matrix of shape (d, n) where:
        - d = number of features (256 for 16x16 patches)
        - n = number of samples (~20,000 patches)

    Returns
    -------
    U : np.ndarray
        Matrix of principal components (eigenvectors), shape (d, d)
        Each COLUMN is a principal component (basis vector)
    """

    # Get dimensions
    d, n = X.shape
    print(f"Data shape: {d} features x {n} samples")

    # Compute covariance matrix
    # Cov = (1/n) * X * X^T
    # This is a d×d matrix showing variance/covariance between features
    cov = (1.0 / n) * (X @ X.T)
    print(f"Covariance matrix shape: {cov.shape}")

    # Eigendecomposition
    # For symmetric matrices, eigh is more efficient than eig
    # eigvals: variance along each principal component
    # eigvecs: the principal components (directions)
    eigvals, eigvecs = np.linalg.eigh(cov)
    print(f"Computed {len(eigvals)} eigenvalues")

    # Sort by eigenvalues (descending order)
    # Largest eigenvalue = direction of maximum variance
    idx = np.argsort(eigvals)[::-1]  # [::-1] reverses to descending
    # Reorder eigenvectors by importance
    U = eigvecs[:, idx]

    #Print variance explained by top components
    total_var = np.sum(eigvals)
    top_10_var = np.sum(eigvals[idx[:10]]) / total_var * 100
    print(f"Top 10 components explain {top_10_var:.2f}% of variance")

    return U
```

```
U = pca(X_pca)
print("PCA components U shape:", U.shape)
```

```
# Cell 5: plot_ica_filters

def plot_ica_filters(W):
    """
    Plot ICA filters in a big grid.

    Parameters
    ----------
    W : np.ndarray
        Unmixing/filters matrix of shape (num_filters, patch_size^2)
    """
    global patch_size, W_z
    if W_z is None:
        raise ValueError("W_z is not set. Call load_images() first to compute W_z.")

    F = W @ W_z

    # Sort filters by 2-norm of rows of F.
    norms = np.linalg.norm(F, axis=1)
    idxs = np.argsort(norms)  # ascending order

    # Big image size: ((patch_size+1)*patch_size - 1)
    big_side = (patch_size + 1) * patch_size - 1
    big_filters = np.min(W) * np.ones((big_side, big_side))

    # Fill in each patch
    for i in range(patch_size):
        for j in range(patch_size):
            filter_idx = idxs[i * patch_size + j]
            filt = W[filter_idx, :].reshape(patch_size, patch_size)

            row_start = i * (patch_size + 1)
            row_end = row_start + patch_size
            col_start = j * (patch_size + 1)
            col_end = col_start + patch_size

            big_filters[row_start:row_end, col_start:col_end] = filt

    plt.figure()
    plt.imshow(big_filters, cmap="gray", aspect="equal")
    plt.axis("off")
    plt.show()
```

```
plot_ica_filters(W)
```

```
# Cell 6: plot_pca_filters

def plot_pca_filters(U):
    """
    Plot PCA filters in a big grid.

    Parameters
    ----------
    U : np.ndarray
        Matrix of principal components of shape (patch_size^2, patch_size^2),
        where each COLUMN is a filter.
    """
    global patch_size

    big_side = (patch_size + 1) * patch_size - 1
    big_filters = np.min(U) * np.ones((big_side, big_side))

    for i in range(patch_size):
        for j in range(patch_size):
            col_idx = i * patch_size + j
            filt = U[:, col_idx].reshape(patch_size, patch_size)

            row_start = i * (patch_size + 1)
            row_end = row_start + patch_size
            col_start = j * (patch_size + 1)
            col_end = col_start + patch_size

            big_filters[row_start:row_end, col_start:col_end] = filt

    plt.figure()
    plt.imshow(big_filters, cmap="gray", aspect="equal")
    plt.axis("off")
    plt.show()
```

```
plot_pca_filters(U)
```