Week 2, Lecture 3

NumPy Fundamentals

Lecture Overview

This lecture deepens your understanding of NumPy with advanced array manipulation, reshaping, stacking, and linear algebra operations essential for machine learning implementations.

Topics Covered:

- Array reshaping and transposing
- Concatenation and splitting
- Linear algebra operations
- Matrix decompositions
- Solving linear systems
- Eigenvalues and eigenvectors

1. Array Manipulation

1.1 Reshaping Arrays

```
import numpy as np

print("Array Reshaping")
print("="*70)

# Create array
arr = np.arange(12)
print(f"Original array: {arr}")
print(f"Shape: {arr.shape}")

# Reshape to 2D
reshaped_3x4 = arr.reshape(3, 4)
print(f"\nReshaped to 3x4:\n{reshaped_3x4}")

reshaped_2x6 = arr.reshape(2, 6)
print(f"\nReshaped to 2x6:\n{reshaped_2x6}")

# Automatic dimension
reshaped_auto = arr.reshape(3, -1)  # -1 means "figure it out"
print(f"\nReshaped to 3x-1 (automatic):\n{reshaped_auto}")

# Transpose
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(f"\nOriginal (2x3):\n{matrix}")
print(f"\nTransposed (3x2):\n{matrix.T}")

# Flatten
flattened = reshaped_3x4.flatten()
print(f"\nFlattened: {flattened}")

# Ravel (similar but returns view when possible)
raveled = reshaped_3x4.ravel()
print(f"Raveled: {raveled}")

# Reshape vs resize
print(f"\nreshape: Returns new array, doesn't modify original")
print(f"resize: Modifies array in-place")
```

```
Array Reshaping
======================================================================
Original array: [ 0  1  2  3  4  5  6  7  8  9 10 11]
Shape: (12,)

Reshaped to 3x4:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

Reshaped to 2x6:
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]

Reshaped to 3x-1 (automatic):
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

Original (2x3):
```

```
[[1 2 3]
 [4 5 6]]

Transposed (3x2):
[[1 4]
 [2 5]
 [3 6]]

Flattened: [ 0  1  2  3  4  5  6  7  8  9 10 11]
Raveled: [ 0  1  2  3  4  5  6  7  8  9 10 11]

reshape: Returns new array, doesn't modify original
resize: Modifies array in-place
```

1.2 Stacking and Splitting

```python
# Stacking arrays
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

print("Stacking Arrays")
print("="*70)
print(f"Array a (2x2):\n{a}")
print(f"\nArray b (2x2):\n{b}")

# Vertical stack (row-wise)
vstack_result = np.vstack([a, b])
print(f"\nvstack (vertical):\n{vstack_result}")
print(f"Shape: {vstack_result.shape}")

# Horizontal stack (column-wise)
hstack_result = np.hstack([a, b])
print(f"\nhstack (horizontal):\n{hstack_result}")
print(f"Shape: {hstack_result.shape}")

# Depth stack (3D)
dstack_result = np.dstack([a, b])
print(f"\ndstack (depth):")
print(f"Shape: {dstack_result.shape}")

# Concatenate with axis
concat_0 = np.concatenate([a, b], axis=0)
concat_1 = np.concatenate([a, b], axis=1)
print(f"\nconcatenate axis=0 (like vstack): {concat_0.shape}")
print(f"concatenate axis=1 (like hstack): {concat_1.shape}")

# Splitting
arr = np.arange(16).reshape(4, 4)
print(f"\nOriginal array (4x4):\n{arr}")

# Vertical split
v_split = np.vsplit(arr, 2)
print(f"\nvsplit into 2 arrays:")
for i, sub in enumerate(v_split):
    print(f"  Subarray {i}:\n{sub}")

# Horizontal split
h_split = np.hsplit(arr, 2)
print(f"\nhsplit into 2 arrays:")
for i, sub in enumerate(h_split):
    print(f"  Subarray {i}:\n{sub}")
```

```
Stacking Arrays
======================================================================
Array a (2x2):
[[1 2]
 [3 4]]

Array b (2x2):
[[5 6]
 [7 8]]

vstack (vertical):
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
Shape: (4, 2)

hstack (horizontal):
[[1 2 5 6]
 [3 4 7 8]]
Shape: (2, 4)

dstack (depth):
Shape: (2, 2, 2)

concatenate axis=0 (like vstack): (4, 2)
concatenate axis=1 (like hstack): (2, 4)

Original array (4x4):
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]

vsplit into 2 arrays:
  Subarray 0:
[[0 1 2 3]
 [4 5 6 7]]
  Subarray 1:
[[ 8  9 10 11]
 [12 13 14 15]]

hsplit into 2 arrays:
  Subarray 0:
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
  Subarray 1:
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

2. Linear Algebra

2.1 Matrix Operations

```
# Matrix multiplication
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print("Matrix Operations")
print("="*70)
print(f"Matrix A:\n{A}")
print(f"\nMatrix B:\n{B}")

# Dot product (matrix multiplication)
C = np.dot(A, B)
print(f"\nDot product (A @ B):\n{C}")

# Also can use @ operator
C_operator = A @ B
print(f"\nUsing @ operator:\n{C_operator}")

# Element-wise vs matrix multiplication
element_wise = A * B
print(f"\nElement-wise (A * B):\n{element_wise}")
print(f"\nMatrix mult (A @ B):\n{C}")

# Matrix properties
print(f"\nMatrix properties:")
print(f"Trace of A: {np.trace(A)}")
print(f"Determinant of A: {np.linalg.det(A)}")
print(f"Rank of A: {np.linalg.matrix_rank(A)}")

# Matrix inverse
A_inv = np.linalg.inv(A)
print(f"\nInverse of A:\n{A_inv}")

# Verify: A @ A_inv = I
identity = A @ A_inv
print(f"\nA @ A_inv (should be identity):\n{np.round(identity, decimals=10)}")
```

```
Matrix Operations
======================================================================
Matrix A:
[[1 2]
 [3 4]]

Matrix B:
[[5 6]
 [7 8]]

Dot product (A @ B):
[[19 22]
 [43 50]]

Using @ operator:
[[19 22]
 [43 50]]

Element-wise (A * B):
[[ 5 12]
 [21 32]]
```

```
Matrix mult (A @ B):
[[19 22]
 [43 50]]

Matrix properties:
Trace of A: 5
Determinant of A: -2.0000000000000004
Rank of A: 2

Inverse of A:
[[-2.   1. ]
 [ 1.5 -0.5]]

A @ A_inv (should be identity):
[[1. 0.]
 [0. 1.]]
```

2.2 Solving Linear Systems

```
# Solve Ax = b
print("Solving Linear Systems")
print("="*70)

# System: 2x + 3y = 8
#         4x + 5y = 14
A = np.array([[2, 3], [4, 5]])
b = np.array([8, 14])

print(f"System: Ax = b")
print(f"A =\n{A}")
print(f"b = {b}")

# Solve
x = np.linalg.solve(A, b)
print(f"\nSolution: x = {x}")

# Verify
result = A @ x
print(f"\nVerification: A @ x = {result}")
print(f"Expected b = {b}")
print(f"Match: {np.allclose(result, b)}")

# Least squares (overdetermined system)
print(f"\nLeast Squares Example:")
# More equations than unknowns
A_over = np.array([[1, 1], [1, 2], [1, 3]])
b_over = np.array([2, 3, 5])

print(f"A (3x2):\n{A_over}")
print(f"b (3,): {b_over}")

# Find best fit
x_ls, residuals, rank, s = np.linalg.lstsq(A_over, b_over, rcond=None)
print(f"\nLeast squares solution: {x_ls}")
print(f"Residuals: {residuals}")
```

```
Solving Linear Systems
======================================================================
System: Ax = b
A =
[[2 3]
 [4 5]]
b = [ 8 14]

Solution: x = [1. 2.]

Verification: A @ x = [ 8. 14.]
Expected b = [ 8 14]
Match: True

Least Squares Example:
A (3x2):
[[1 1]
 [1 2]
 [1 3]]
b (3,): [2 3 5]

Least squares solution: [0.33333333 1.5       ]
Residuals: [0.16666667]
```

2.3 Eigenvalues and Eigenvectors

```
# Eigenvalue decomposition
print("Eigenvalues and Eigenvectors")
print("="*70)

A = np.array([[4, 2], [1, 3]])
print(f"Matrix A:\n{A}")

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

print(f"\nEigenvalues: {eigenvalues}")
print(f"\nEigenvectors:\n{eigenvectors}")

# Verify: A @ v = λ @ v
print(f"\nVerification:")
for i in range(len(eigenvalues)):
    lambda_i = eigenvalues[i]
    v_i = eigenvectors[:, i]

    left = A @ v_i
    right = lambda_i * v_i

    print(f"\nEigenvalue {i+1}: λ = {lambda_i:.4f}")
    print(f"  A @ v = {left}")
    print(f"  λ * v = {right}")
    print(f"  Match: {np.allclose(left, right)}")

# Diagonalization: A = V @ D @ V^(-1)
D = np.diag(eigenvalues)
V = eigenvectors
V_inv = np.linalg.inv(V)

A_reconstructed = V @ D @ V_inv
print(f"\nDiagonalization:")
print(f"D (diagonal):\n{D}")
print(f"\nReconstructed A:\n{np.real(A_reconstructed)}")
print(f"Match original: {np.allclose(A, A_reconstructed)}")
```

```
Eigenvalues and Eigenvectors
======================================================================
Matrix A:
[[4 2]
 [1 3]]

Eigenvalues: [5. 2.]

Eigenvectors:
[[0.89442719 0.70710678]
 [0.4472136  0.70710678]]

Verification:

Eigenvalue 1: λ = 5.0000
  A @ v = [4.47213595 2.23606798]
  λ * v = [4.47213595 2.23606798]
  Match: True

Eigenvalue 2: λ = 2.0000
  A @ v = [1.41421356 1.41421356]
  λ * v = [1.41421356 1.41421356]
  Match: True
```

```
Diagonalization:
D (diagonal):
[[5. 0.]
 [0. 2.]]

Reconstructed A:
[[4. 2.]
 [1. 3.]]
Match original: True
```

Summary

Key Takeaways:

- reshape() and transpose() modify array structure
- vstack/hstack combine arrays along different axes
- Matrix operations use @ operator or np.dot()
- `np.linalg provides comprehensive linear algebra tools`
- Eigenvalues are critical for PCA and other ML algorithms

Practice Exercises

1. Solve a 3x3 system of linear equations

2. Compute eigenvalues of a covariance matrix

3. Implement matrix multiplication without using @ or dot