CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 2, Lecture 4

Intermediate NumPy

Lecture Overview

Advanced NumPy techniques including universal functions, advanced indexing, memory optimization, and practical applications for machine learning.

Topics Covered:

- Universal functions (ufuncs)
- Advanced indexing techniques
- Memory views and copies
- Structured arrays
- Performance optimization
- Real-world ML applications

1. Universal Functions

```
import numpy as np

print("Universal Functions (ufuncs)")
print("="*70)

# Unary ufuncs
x = np.array([1, 2, 3, 4, 5])
print(f"Array x: {x}")

print(f"\nUnary ufuncs:")
print(f"np.sqrt(x): {np.sqrt(x)}")
print(f"np.exp(x): {np.exp(x)}")
print(f"np.log(x): {np.log(x)}")
print(f"np.abs([-1, -2, 3]): {np.abs([-1, -2, 3])}")

# Binary ufuncs
y = np.array([5, 4, 3, 2, 1])
print(f"\nArray y: {y}")
print(f"\nBinary ufuncs:")
print(f"np.add(x, y): {np.add(x, y)}")
print(f"np.maximum(x, y): {np.maximum(x, y)}")
print(f"np.minimum(x, y): {np.minimum(x, y)}")

# Custom ufunc
def custom_func(x):
    return x**2 + 2*x + 1

vectorized_func = np.vectorize(custom_func)
result = vectorized_func(x)
print(f"\nCustom ufunc (x^2 + 2x + 1): {result}")

# Comparing with loops
import time
data = np.random.rand(1000000)

# With loop
start = time.time()
result_loop = np.array([np.sqrt(val) for val in data])
time_loop = time.time() - start

# With ufunc
start = time.time()
result_ufunc = np.sqrt(data)
time_ufunc = time.time() - start

print(f"\nPerformance comparison (1M elements):")
print(f"Loop: {time_loop:.4f}s")
print(f"Ufunc: {time_ufunc:.4f}s")
print(f"Speedup: {time_loop/time_ufunc:.1f}x")
```

```
Universal Functions (ufuncs)
======================================================================
Array x: [1 2 3 4 5]

Unary ufuncs:
np.sqrt(x): [1.         1.41421356 1.73205081 2.         2.23606798]
np.exp(x): [  2.71828183   7.3890561   20.08553692  54.59815003 148.4131591 ]
np.log(x): [0.         0.69314718 1.09861229 1.38629436 1.60943791]
np.abs([-1, -2, 3]): [1 2 3]

Array y: [5 4 3 2 1]

Binary ufuncs:
np.add(x, y): [6 6 6 6 6]
np.maximum(x, y): [5 4 3 4 5]
np.minimum(x, y): [1 2 3 2 1]

Custom ufunc (x^2 + 2x + 1): [ 4  9 16 25 36]

Performance comparison (1M elements):
Loop: 0.2841s
Ufunc: 0.0031s
Speedup: 91.6x
```

2. Advanced Indexing

```
# Integer array indexing
arr = np.arange(10, 40, 5)
print("Advanced Indexing")
print("="*70)
print(f"Array: {arr}")

# Integer indexing
indices = [0, 2, 4]
print(f"\nInteger indexing arr[[0, 2, 4]]: {arr[indices]}")

# 2D integer indexing
arr_2d = np.arange(12).reshape(3, 4)
print(f"\n2D array:\n{arr_2d}")

rows = np.array([0, 1, 2])
cols = np.array([0, 2, 3])
print(f"\nrows = {rows}, cols = {cols}")
print(f"arr_2d[rows, cols]: {arr_2d[rows, cols]}")

# Boolean masking
print(f"\nBoolean masking:")
mask = arr > 20
print(f"Mask (arr > 20): {mask}")
print(f"Values > 20: {arr[mask]}")

# Modifying with boolean indexing
arr_copy = arr.copy()
arr_copy[arr_copy > 20] = 20
print(f"\nAfter capping at 20: {arr_copy}")

# Where function
x = np.array([1, 2, 3, 4, 5])
result = np.where(x > 3, x * 2, x)
print(f"\nnp.where example:")
print(f"x = {np.array([1, 2, 3, 4, 5])}")
print(f"np.where(x > 3, x*2, x): {result}")
print(f"(if > 3: multiply by 2, else: keep original)")
```

```
Advanced Indexing
======================================================================
Array: [10 15 20 25 30 35]

Integer indexing arr[[0, 2, 4]]: [10 20 30]

2D array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

rows = [0 1 2], cols = [0 2 3]
arr_2d[rows, cols]: [ 0  6 11]
```

```
Boolean masking:
Mask (arr > 20): [False False False  True  True  True]
Values > 20: [25 30 35]

After capping at 20: [10 15 20 20 20 20]

np.where example:
x = [1 2 3 4 5]
np.where(x > 3, x*2, x): [1 2 3 8 10]
(if > 3: multiply by 2, else: keep original)
```

3. Memory and Performance

```python
# Views vs copies
print("Memory: Views vs Copies")
print("="*70)

original = np.arange(10)
print(f"Original: {original}")

# Slicing creates a view (shares memory)
view = original[2:7]
print(f"\nView (slice [2:7]): {view}")

# Modifying view affects original
view[0] = 999
print(f"After modifying view[0] = 999:")
print(f"  View: {view}")
print(f"  Original: {original}")
print(f"  Original was modified!")

# Reset
original = np.arange(10)

# Copy creates independent array
copy = original[2:7].copy()
print(f"\nCopy (with .copy()): {copy}")
copy[0] = 999
print(f"After modifying copy[0] = 999:")
print(f"  Copy: {copy}")
print(f"  Original: {original}")
print(f"  Original unchanged!")

# Check if sharing memory
print(f"\nMemory sharing:")
view2 = original[:]
copy2 = original.copy()
print(f"view shares memory: {np.shares_memory(original, view2)}")
print(f"copy shares memory: {np.shares_memory(original, copy2)}")

# Memory layout
arr = np.arange(12).reshape(3, 4)
print(f"\nMemory layout:")
print(f"C-contiguous (row-major): {arr.flags['C_CONTIGUOUS']}")
print(f"F-contiguous (col-major): {arr.flags['F_CONTIGUOUS']}")
```

```
Memory: Views vs Copies
======================================================================
Original: [0 1 2 3 4 5 6 7 8 9]

View (slice [2:7]): [2 3 4 5 6]
After modifying view[0] = 999:
  View: [999   3   4   5   6]
  Original: [  0   1 999   3   4   5   6   7   8   9]
  Original was modified!

Copy (with .copy()): [2 3 4 5 6]
After modifying copy[0] = 999:
  Copy: [999   3   4   5   6]
  Original: [0 1 2 3 4 5 6 7 8 9]
  Original unchanged!

Memory sharing:
view shares memory: True
copy shares memory: False

Memory layout:
C-contiguous (row-major): True
F-contiguous (col-major): False
```

4. ML Application: Image Processing

```
# Simulating image operations
print("ML Application: Image Processing")
print("="*70)

# Create synthetic "image" (8x8 grayscale)
np.random.seed(42)
image = np.random.randint(0, 256, (8, 8))
print(f"Synthetic image (8x8):")
print(image)

# Image statistics
print(f"\nImage statistics:")
print(f"Min pixel value: {np.min(image)}")
print(f"Max pixel value: {np.max(image)}")
print(f"Mean pixel value: {np.mean(image):.2f}")
print(f"Std pixel value: {np.std(image):.2f}")

# Normalize image to [0, 1]
normalized = (image - np.min(image)) / (np.max(image) - np.min(image))
print(f"\nNormalized to [0, 1]:")
print(f"Min: {np.min(normalized):.2f}, Max: {np.max(normalized):.2f}")

# Apply threshold (create binary image)
threshold = 128
binary = (image > threshold).astype(np.uint8)
print(f"\nBinary image (threshold={threshold}):")
print(binary)

# Padding (like conv padding)
padded = np.pad(image, pad_width=1, mode='constant', constant_values=0)
print(f"\nPadded image shape: {image.shape} -> {padded.shape}")

# Simple convolution (edge detection)
kernel = np.array([[-1, -1, -1],
                   [-1,  8, -1],
                   [-1, -1, -1]])
print(f"\nEdge detection kernel:")
print(kernel)

# Apply to center of padded image
center = padded[1:4, 1:4]
response = np.sum(center * kernel)
print(f"\nKernel response at center: {response}")
```

```
ML Application: Image Processing
======================================================================
Synthetic image (8x8):
[[102 220  72 145 161  73 205 116]
 [124  83 115 193 166  78  29 110]
 [ 76 138 213  61  89 134 211  76]
 [ 77  89 227  88 154  22  46 159]
 [225  56 255 176  54  32 210  93]
 [ 34 132  37 117 163 244 155  43]
 [237 208  87  18  96  68 173  38]
 [ 21  67  28  88  66 171  72 168]]

Image statistics:
Min pixel value: 18
Max pixel value: 255
Mean pixel value: 118.06
Std pixel value: 65.23

Normalized to [0, 1]:
Min: 0.00, Max: 1.00

Binary image (threshold=128):
[[0 1 0 1 1 0 1 0]
 [0 0 0 1 1 0 0 0]
 [0 1 1 0 0 1 1 0]
 [0 0 1 0 1 0 0 1]
 [1 0 1 1 0 0 1 0]
 [0 1 0 0 1 1 1 0]
 [1 1 0 0 0 0 1 0]
 [0 0 0 0 0 1 0 1]]

Padded image shape: (8, 8) -> (10, 10)

Edge detection kernel:
[[-1 -1 -1]
 [-1  8 -1]
 [-1 -1 -1]]

Kernel response at center: 83
```

Summary

Key Takeaways:

- Ufuncs provide vectorized operations (much faster than loops)
- Advanced indexing enables complex data selection
- Views share memory, copies don't - important for performance
- NumPy is foundation for image and signal processing
- Mastering NumPy is essential for ML implementation

Practice Exercises

```
1. Implement convolution operation from scratch
```

2. Create a function to rotate an image 90 degrees

3. Compare performance of loops vs vectorization