Lab 2: NumPy for Audio Processing

Duration: 3-5 minutes **Presenter Guide:** This lab teaches NumPy fundamentals for audio manipulation

---

Slide 1: Lab Overview (30 seconds)

Title: "Lab 2: NumPy for Audio Processing"

Subtitle: "The Numerical Foundation"

What you'll learn: - NumPy array operations on audio - Array slicing and manipulation - Why NumPy is essential for audio ML

Team Member Connections: - **Cameron's POC:** Performs ~765,000 measurements using NumPy arrays - **cervanj2's Architecture:** Both preprocessing and post-processing use NumPy - **Ryan & Yovannoa:** Both showed NumPy ↔︎ PyTorch tensor conversion

---

Slide 2: Why NumPy? (45 seconds)

Title: "NumPy Powers Audio Processing"

Key Reasons: 1. **Speed:** Vectorized operations are 10-100x faster than Python loops 2. **Memory:** Efficient array storage 3. **Interoperability:** Works with librosa, PyTorch, matplotlib 4. **Broadcasting:** Apply operations to entire arrays at once

Cameron's Use Case: - 18-slice multi-scale analysis - ~765,000 spectrogram measurements - All using NumPy array operations

The NumPy Advantage:

```python
# Slow Python way (don't do this)
for i in range(len(audio)):
    audio[i] = audio[i] * 0.5  # Adjust volume

# Fast NumPy way
audio = audio * 0.5  # Same operation, 100x faster!
```

Slide 3: Audio Arrays in Action (1 minute 30 seconds)

Title: "Hello World: NumPy Audio Operations"

Code Example:

```python
import numpy as np
import matplotlib.pyplot as plt

# Create a simple audio signal (440 Hz sine wave - A note)
sample_rate = 22050
duration = 1.0  # seconds
frequency = 440  # Hz (A4 note)

# Generate time points
t = np.linspace(0, duration, int(sample_rate * duration))

# Generate sine wave using NumPy
audio_signal = np.sin(2 * np.pi * frequency * t)

print("=== Audio Signal Info ===")
print(f"Signal shape: {audio_signal.shape}")
print(f"Signal dtype: {audio_signal.dtype}")
print(f"Signal range: [{audio_signal.min():.3f}, {audio_signal.max():.3f}]
print(f"Number of samples: {len(audio_signal)}")

# Simple operations
quiet_signal = audio_signal * 0.5  # Reduce volume
loud_signal = audio_signal * 2.0   # Increase volume
inverted = -audio_signal           # Invert phase

print(f"\nQuiet range: [{quiet_signal.min():.3f}, {quiet_signal.max():.3f}
print(f"Loud range: [{loud_signal.min():.3f}, {loud_signal.max():.3f}]")

# Visualize
plt.figure(figsize=(12, 6))
plt.subplot(3, 1, 1)
plt.plot(audio_signal[:200])
plt.title("Original Signal")
plt.ylabel("Amplitude")

plt.subplot(3, 1, 2)
plt.plot(quiet_signal[:200])
plt.title("Quiet Signal (0.5x)")
plt.ylabel("Amplitude")

plt.subplot(3, 1, 3)
plt.plot(loud_signal[:200])
plt.title("Loud Signal (2.0x)")
plt.xlabel("Sample")
plt.ylabel("Amplitude")

plt.tight_layout()
plt.show()
```

Demo Points: - “We create 22,050 samples in one NumPy operation” - “Volume adjustments are just multiplication” - “All operations happen simultaneously on the entire array”

Slide 4: Array Slicing - Cameron’s Approach (1 minute 30 seconds)

Title: “18-Slice Analysis Like Cameron’s POC”

Code Example:

```python
import numpy as np
import librosa

# Load audio (from Lab 1)
audio, sr = librosa.load('sample_audio.wav', sr=22050)

# Convert to spectrogram
stft = librosa.stft(audio)
magnitude = np.abs(stft)

print(f"Full spectrogram shape: {magnitude.shape}")
# Example output: (1025, 646) - 1025 frequencies, 646 time frames

# Cameron's approach: Divide into slices for multi-scale analysis
num_slices = 18
time_frames = magnitude.shape[1]
slice_size = time_frames // num_slices

print(f"\n=== Cameron's 18-Slice Analysis ===")
print(f"Total time frames: {time_frames}")
print(f"Slice size: {slice_size} frames")
print(f"Number of slices: {num_slices}")

# Extract slices using NumPy array slicing
slices = []
for i in range(num_slices):
    start = i * slice_size
    end = start + slice_size
    slice_data = magnitude[:, start:end]  # All frequencies, specific time
    slices.append(slice_data)

    # Analyze this slice
    slice_mean = slice_data.mean()
    slice_std = slice_data.std()
    print(f"Slice {i+1}: mean={slice_mean:.3f}, std={slice_std:.3f}")

# This is the foundation of Cameron's ~765,000 measurements!
total_measurements = magnitude.shape[0] * magnitude.shape[1]
print(f"\nTotal measurements analyzed: {total_measurements:,}")
```

Expected Output:

```
Full spectrogram shape: (1025, 646)

=== Cameron's 18-Slice Analysis ===
Total time frames: 646
Slice size: 35 frames
Number of slices: 18
Slice 1: mean=0.023, std=0.041
Slice 2: mean=0.019, std=0.035
...
Slice 18: mean=0.025, std=0.039

Total measurements analyzed: 662,150
```

Key Concepts: - Array slicing: `magnitude[:, start:end]` - `:` means “all rows (all frequencies)” - `start:end` selects specific columns (time frames) - This is how Cameron divides audio into segments

---

Slide 5: NumPy → PyTorch Bridge (45 seconds)

Title: “From NumPy to PyTorch (Preview Lab 4)”

Code Example:

```python
import numpy as np
import torch

# NumPy array (from librosa)
numpy_array = np.random.randn(1025, 646)  # Simulating spectrogram
print(f"NumPy array shape: {numpy_array.shape}")
print(f"NumPy array type: {type(numpy_array)}")

# Convert to PyTorch tensor (for U-Net processing)
pytorch_tensor = torch.from_numpy(numpy_array).float()
print(f"\nPyTorch tensor shape: {pytorch_tensor.shape}")
print(f"PyTorch tensor type: {type(pytorch_tensor)}")

# They share memory! Efficient conversion
print(f"\nMemory shared: {np.shares_memory(numpy_array, pytorch_tensor.num

# Back to NumPy (after U-Net processing)
back_to_numpy = pytorch_tensor.numpy()
print(f"Back to NumPy shape: {back_to_numpy.shape}")
```

Why This Matters: - librosa returns NumPy arrays - PyTorch models (U-Net) use PyTorch tensors - NumPy ↔ PyTorch conversion is seamless - Both Ryan and Yovannoa learned this in their tutorials

---

Slide 6: Connection to Pipeline (30 seconds)

Title: "NumPy in the Full Pipeline"

The Complete Pipeline:

```
Load Audio → [NumPy Processing] → STFT → PyTorch Tensor → U-Net
     ↓              ↓                 ↓          ↓            ↓
  librosa        NumPy            librosa    NumPy→Torch   PyTorch
   arrays        ops              arrays      bridge       tensors
```

cervanj2's Architecture: - **Preprocessing Layer:** NumPy + librosa for STFT - **Post-processing Layer:** NumPy + librosa for ISTFT

Cameron's POC: - All 765,000 measurements are NumPy operations - Array slicing divides spectrogram into 18 sections - Each measurement is a NumPy calculation

---

Demo Script (Detailed Timing) - Provided by Claude

0:00 - 0:30: Introduction

"Welcome to Lab 2: NumPy for Audio Processing! In Lab 1, we loaded audio into Python. Now we'll learn how to manipulate it efficiently using NumPy - the foundation of the POC and all audio processing."

0:30 - 1:15: Why NumPy?

"Why is NumPy so important? [Show Slide 2]. First, speed - NumPy operations are vectorized, meaning they're 10 to 100 times faster than Python loops. The POC performs about 765,000 measurements on spectrograms. Without NumPy, that would take hours. With NumPy, it's seconds.

Second, NumPy arrays work seamlessly with librosa for audio processing and PyTorch for deep learning. It's the glue that holds everything together."

1:15 - 2:45: Basic Operations

"Let's see NumPy in action. [Show Slide 3]. We'll create a simple audio signal - a 440 Hz sine wave, which is the musical note A.

[Run the code]

Look at this - we created 22,050 samples in a single line! That's one second of audio. Now watch how easy it is to manipulate…

[Show the volume adjustments]

We can make it quieter or louder with simple multiplication. The whole array is processed at once - that's the NumPy advantage. And you can see the results in the plots."

2:45 - 4:00: Cameron's Slicing Approach

"Now here's the really cool part - Cameron's 18-slice analysis. [Show Slide 4]. In his POC, Cameron divides the spectrogram into 18 slices for multi-scale analysis. Let's see how NumPy makes this possible.

[Run the code]

Our spectrogram has 1,025 frequency bins and 646 time frames - that's 662,150 measurements total! Cameron divides this into 18 slices using NumPy array slicing.

See this syntax? `magnitude[:, start:end]` - the colon means 'all frequencies', and `start:end` selects specific time frames. This is how we carve up the audio for analysis.

Each slice is analyzed independently - that's the multi-scale approach. All powered by NumPy array operations."

4:00 - 4:45: PyTorch Bridge

"One more thing - [Show Slide 5] - librosa gives us NumPy arrays, but our U-Net model uses PyTorch tensors. Watch how easy the conversion is…

[Show the code]

It's nearly instant, and they even share memory! This seamless integration is why both Ryan and Yovannoa learned about NumPy in their PyTorch tutorials. It's the bridge between audio processing and deep learning."

4:45 - 5:00: Wrap-up

"That's Lab 2! NumPy is the numerical engine that powers everything - from Cameron's 765,000 measurements to the PyTorch tensors we'll use in Lab 5. Next up: Lab 3, where we use librosa to transform audio into spectrograms.

Questions?"

Code Files to Provide

`lab2_numpy_basics.py`

```python
"""
Lab 2: NumPy for Audio Processing
Array operations and slicing for audio data
"""
import numpy as np
import matplotlib.pyplot as plt
import librosa

def create_sine_wave(frequency=440, duration=1.0, sample_rate=22050):
```

```python
def create_sine_wave(frequency=440, duration=1.0, sample_rate=22050):
    """Create a sine wave audio signal"""
    t = np.linspace(0, duration, int(sample_rate * duration))
    audio = np.sin(2 * np.pi * frequency * t)
    return audio, sample_rate

def volume_operations(audio):
    """Demonstrate volume adjustment with NumPy"""
    quiet = audio * 0.5
    loud = audio * 2.0
    inverted = -audio

    print("=== Volume Operations ===")
    print(f"Original range: [{audio.min():.3f}, {audio.max():.3f}]")
    print(f"Quiet range: [{quiet.min():.3f}, {quiet.max():.3f}]")
    print(f"Loud range: [{loud.min():.3f}, {loud.max():.3f}]")

    return quiet, loud, inverted

def slice_analysis(audio_path, num_slices=18):
    """Perform Cameron-style 18-slice analysis"""
    # Load audio
    audio, sr = librosa.load(audio_path, sr=22050)

    # Create spectrogram
    stft = librosa.stft(audio)
    magnitude = np.abs(stft)

    print(f"Spectrogram shape: {magnitude.shape}")

    # Divide into slices
    time_frames = magnitude.shape[1]
    slice_size = time_frames // num_slices

    print(f"\n=== {num_slices}-Slice Analysis ===")
    print(f"Slice size: {slice_size} frames")

    slices = []
    for i in range(num_slices):
        start = i * slice_size
        end = start + slice_size
        slice_data = magnitude[:, start:end]
        slices.append(slice_data)

        # Statistics for each slice
        print(f"Slice {i+1:2d}: "
              f"mean={slice_data.mean():.4f}, "
              f"std={slice_data.std():.4f}, "
              f"max={slice_data.max():.4f}")

    total_measurements = magnitude.shape[0] * magnitude.shape[1]
    print(f"\nTotal measurements: {total_measurements:,}")

    return slices

def numpy_pytorch_bridge():
    """Demonstrate NumPy to PyTorch conversion"""
    try:
        import torch

        # Create NumPy array
        numpy_array = np.random.randn(1025, 646)

        # Convert to PyTorch
        tensor = torch.from_numpy(numpy_array).float()

        # Back to NumPy
        back_to_numpy = tensor.numpy()

        print("=== NumPy ↔ PyTorch Bridge ===")
        print(f"NumPy shape: {numpy_array.shape}")
        print(f"PyTorch shape: {tensor.shape}")
        print(f"Back to NumPy shape: {back_to_numpy.shape}")
        print(f"Memory shared: {np.shares_memory(numpy_array, back_to_nump

    except ImportError:
        print("PyTorch not installed - skipping bridge demo")

if __name__ == "__main__":
    # Create sine wave
    audio, sr = create_sine_wave()

    # Volume operations
    quiet, loud, inverted = volume_operations(audio)

    # Slice analysis (requires audio file)
    # slices = slice_analysis('sample_audio.wav')

    # NumPy-PyTorch bridge
    numpy_pytorch_bridge()
```

Visual Aids Needed

1. **Speed Comparison:** Chart showing Python loop vs NumPy vectorization
2. **Array Slicing Diagram:** Visual showing `magnitude[:, start:end]` operation
3. **18-Slice Visualization:** Spectrogram divided into 18 vertical sections
4. **NumPy ↔️ PyTorch Flow:** Arrows showing data flow between libraries

---

Backup Slides

Extra: Broadcasting

```
# Broadcasting example
audio = np.array([1.0, 2.0, 3.0, 4.0])
# Add a constant to all elements
audio_shifted = audio + 0.5  # [1.5, 2.5, 3.5, 4.5]
```

Extra: Common Audio Operations

- Normalization: `audio / np.abs(audio).max()`
- Fade in: `audio * np.linspace(0, 1, len(audio))`
- Fade out: `audio * np.linspace(1, 0, len(audio))`

Q&A Prep - Provided by Claude

Q: "Why array slicing instead of loops?" A: "Speed and readability. `magnitude[:, 10:20]` is faster and clearer than a loop. This matters when processing 765,000 data points."

Q: "What's the difference between lists and NumPy arrays?" A: "Lists store individual Python objects. Arrays store uniform data types in contiguous memory - much faster for math operations."

Q: "Can we slice audio in different ways?" A: "Yes! `audio[::2]` takes every other sample. `audio[:1000]` takes first 1000. `audio[-1000:]` takes last 1000. Very flexible."

Success Criteria

Students should be able to: - [ ] Create NumPy arrays for audio signals - [ ] Perform element-wise operations (volume, phase) - [ ] Use array slicing to extract segments - [ ] Understand why Cameron uses 18 slices - [ ] Convert between NumPy and PyTorch (preview)

Connection Summary

Cameron's POC: 765,000 measurements via NumPy operations **cervanj2's Architecture:** NumPy in preprocessing & post-processing **Ryan & Yovannoa:** NumPy↔︎PyTorch conversion in tutorials **Next Lab:** librosa STFT using NumPy arrays