Module 1 - Session 4: Working with Tensors

Session 4

What are Tensors?

Tensors are the core Data Structure in PyTorch

Scalar

$$7$$

Vector

$$\begin{bmatrix} 7 \\ 4 \end{bmatrix} \text{ or } \begin{bmatrix} 7 & 4 \end{bmatrix}$$

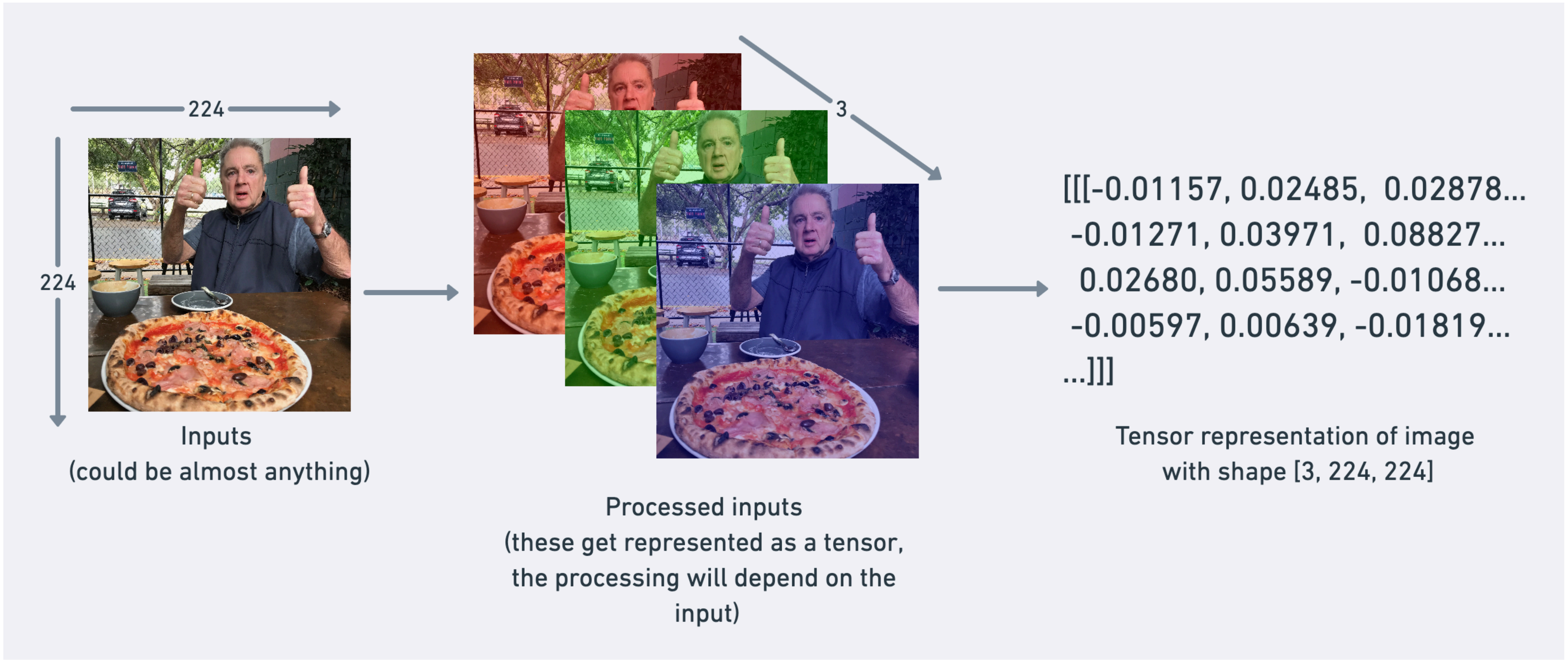
Matrix

$$\begin{bmatrix} 7 & 10 \\ 4 & 3 \\ 5 & 1 \end{bmatrix}$$

Tensor

$$\begin{bmatrix} \begin{bmatrix} 7 & 4 \end{bmatrix} & \begin{bmatrix} 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 9 \end{bmatrix} & \begin{bmatrix} 2 & 3 \end{bmatrix} \\ \begin{bmatrix} 5 & 6 \end{bmatrix} & \begin{bmatrix} 8 & 8 \end{bmatrix} \end{bmatrix}$$

Images as Tensors


224
224
Inputs
(could be almost anything)
3
Processed inputs
(these get represented as a tensor, the processing will depend on the input)
[[[-0.01157, 0.02485, 0.02878...
-0.01271, 0.03971, 0.08827...
0.02680, 0.05589, -0.01068...
-0.00597, 0.00639, -0.01819...
...]]]
Tensor representation of image with shape [3, 224, 224]

PyTorch convention ordering for images is: `(channels, height, width)`.

Why Learn About Tensors?

Many PyTorch errors come from tensor issues

- Shape mismatches
- Data type problems
- Device mismatches

Master tensors now, avoid frustration later

Understanding Shapes

```python
distances.shape   # torch.Size([6, 1])
```

`[6, 1]` means:

- 6 samples (batch size)
- 1 feature per sample

Shape mismatches = most common PyTorch errors

Why Batch Size Doesn’t Cause Problems

Model expects first dimension = batch size

Think of it like a stack of papers:

- Model reads each page the same way
- Whether 6 pages or 600 pages
- First dimension = how many
- Rest = what each sample looks like

Data Types

```python
# Defaults
torch.tensor([1, 2, 3])         # int64
torch.tensor([1.0, 2.0, 3.0])  # float32

# Explicit
torch.tensor([1, 2, 3], dtype=torch.float32)
tensor.float()  # convert to float32
```

For neural networks: `float32` is the sweet spot

Creating Tensors

From Python lists:

```
torch.tensor([[1, 2], [3, 4]])
```

From NumPy:

```
torch.from_numpy(np_array)
```

Built-in patterns:

```
torch.zeros(2, 3)
torch.ones(2, 3)
torch.randn(2, 3)  # random values
```

Reshaping Tensors

Common error: forgetting the batch dimension

```python
# Wrong: scalar
single_value = torch.tensor(25.0)  # shape: []

# Right: batch dimension
single_value = torch.tensor([[25.0]])  # shape: [1, 1]
```

Adding Dimensions

```python
# Add dimension
tensor.unsqueeze(0)  # add at position 0

# Remove dimensions
tensor.squeeze()  # removes all size-1 dimensions
```

Always check shape before using unsqueeze()

Indexing and Slicing

```
predictions[0]        # first prediction
predictions[0:3]      # first three
predictions[0].item() # convert to Python float
```

`.item()` only works on tensors with exactly one element

Multiple Features

```python
# Shape: [batch_size, num_features]
data = torch.tensor([[5.0, 14.0, 1.0],  # distance, hour, weather
                     [6.0, 15.0, 0.0]])

data[0, 1]  # first sample, second feature (hour = 14.0)
```

Element-Wise Operations

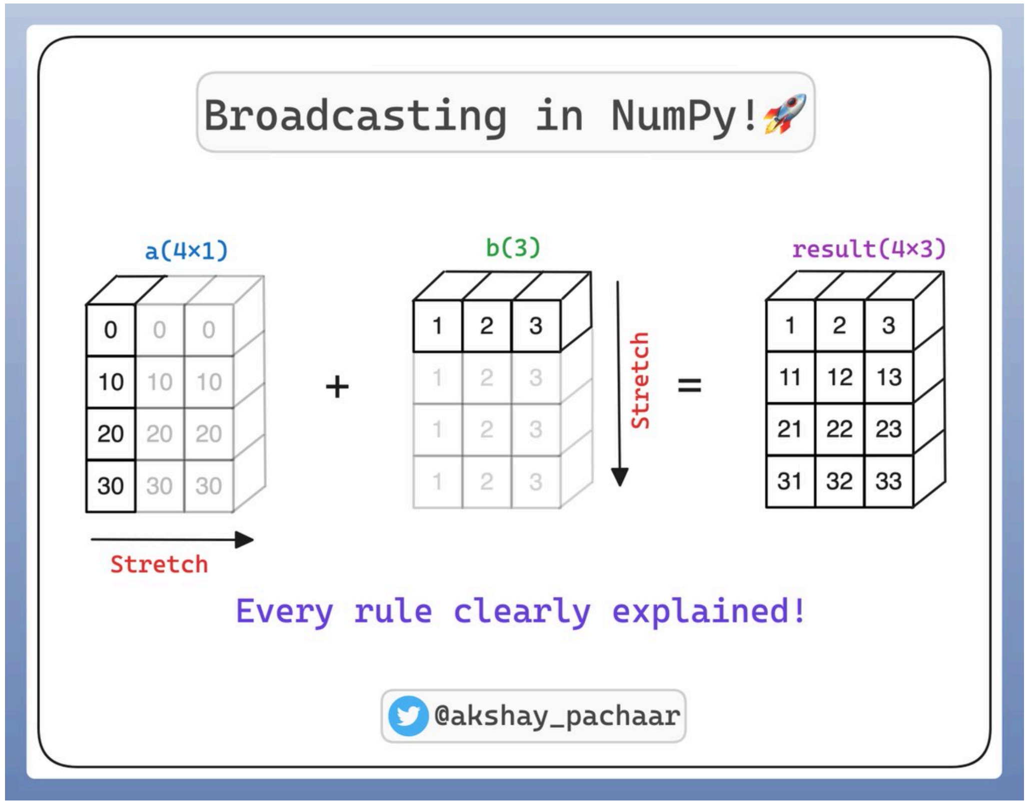
```python
weight = 3.4
bias = 5.0
distances = torch.tensor([[5.0], [6.0], [8.0]])

# Element-wise: applies to each element independently
predictions = weight * distances + bias
```

Same operation applied to all elements at once

Broadcasting

> **Broadcasting** is a powerful mechanism in PyTorch and NumPy that allows you to perform arithmetic operations on tensors of different shapes without manually duplicating data in memory. It makes your code faster and more memory-efficient by vectorizing operations in C rather than Python loops.



Broadcasting: Example

Problem: Want to apply different adjustments to different features

Solution: Broadcasting automatically expands dimensions

```
# Instead of repeating [1.1, 1.0, 5.0] three times:
adjustments = torch.tensor([[1.1, 1.0, 5.0]])  # shape: [1, 3]
data = torch.tensor([[5.0, 14.0, 1.0],
                     [6.0, 15.0, 0.0],
                     [8.0, 16.0, 1.0]])  # shape: [3, 3]

result = data * adjustments  # Broadcasting!
```

How Broadcasting Works

Rule: When one dimension is 1 and the other is larger, PyTorch expands the smaller dimension

Example:

- `[1, 3]` $\times$ `[3, 1]` $\rightarrow$ both become `[3, 3]`
- `[1, 1]` $\times$ `[1, 3]` $\rightarrow$ first becomes `[1, 3]`

Module 1 Summary

You’ve learned:

- Why PyTorch exists and what makes it special
- How neural networks learn from data
- The complete ML pipeline
- Building and training your first model
- Activation functions for non-linear patterns
- Working with tensors (shapes, types, operations)

Lab 3: Tensors: The Core of PyTorch

> “For the things we have to learn before we can do them, we learn by doing them.”

CUE: START THE LAB HERE

Assignment 1: Deeper Regression, Smarter Features

> “What I cannot create, I do not understand.”

CUE: START THE ASSIGNMENT HERE

What’s Next?

In **Module 2: Image Classification** we learn:

- Tackle classification problems
- Dive deeper into how neural networks learn
- Build your first image classifier