Session // 01
INTRODUCTION TO PYTORCH

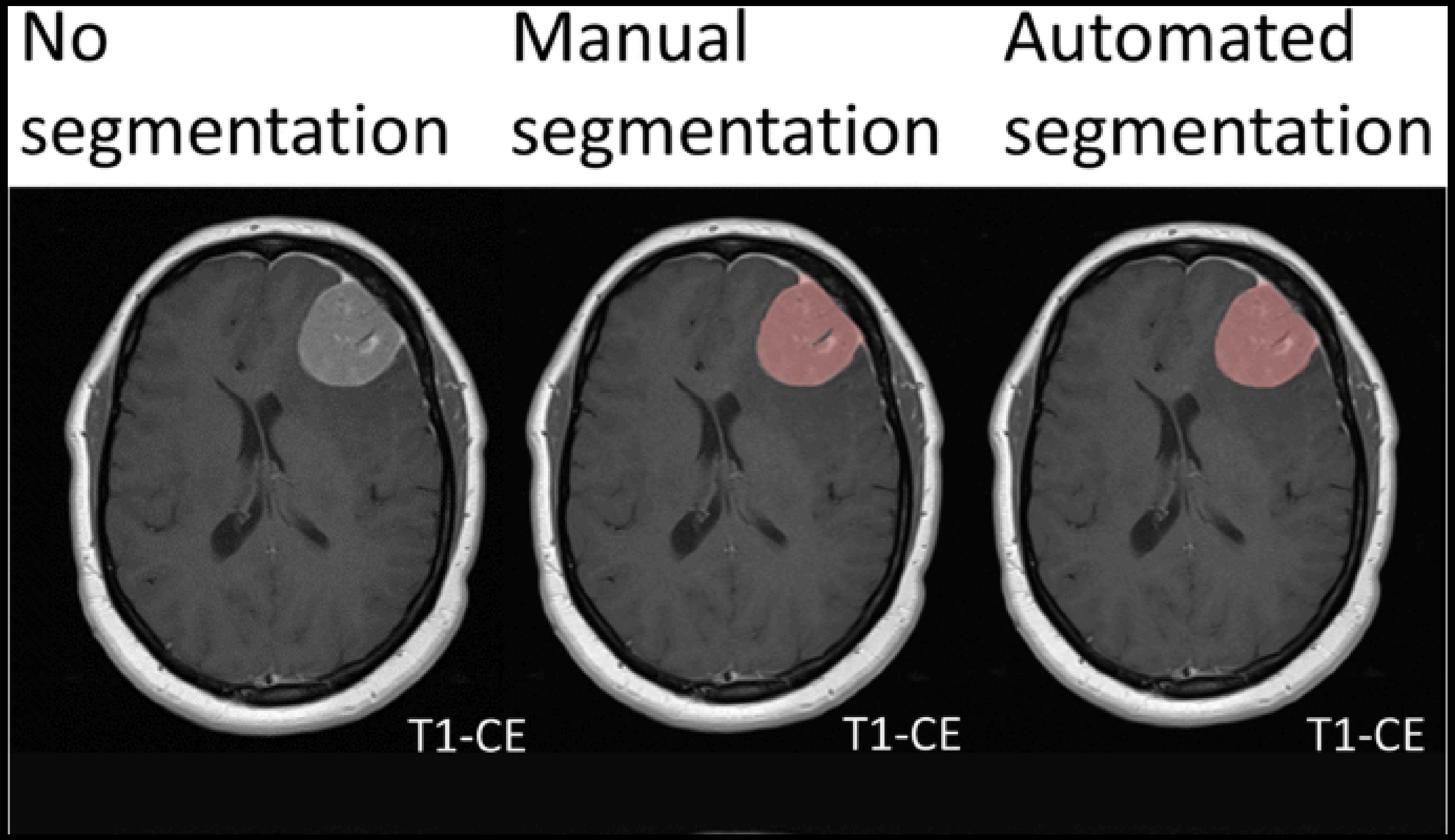
FACULTY OF
SCIENCE AND ENGINEERING
+++

Diego Corona Lopez – AI Technical Specialist

MANCHESTER 1824
The University of Manchester

Agenda

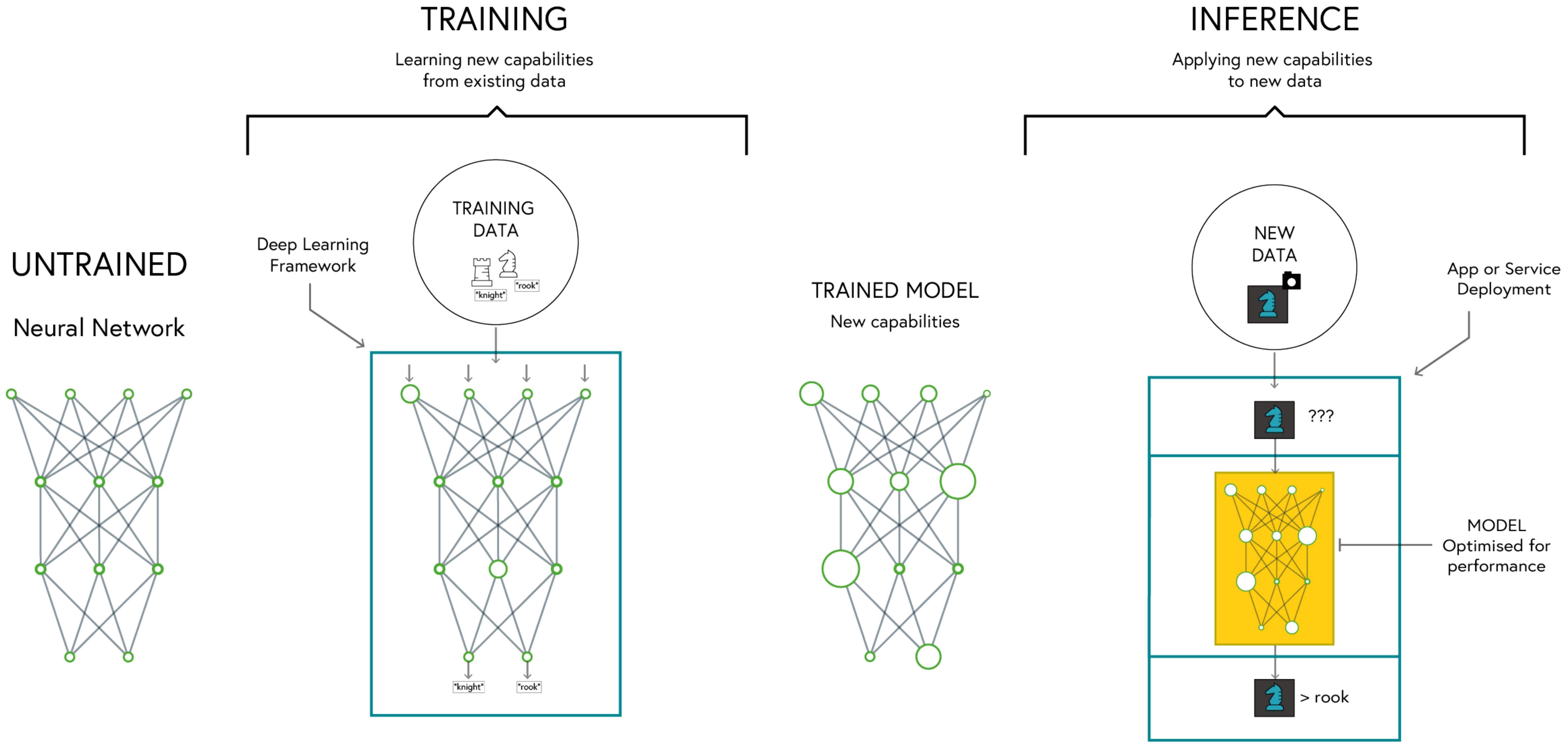
- PyTorch fundamentals and advantages
- Working with tensors
- Tensor operations and manipulation
- Automatic differentiation (Autograd)
- Moving from data to tensors
- GPU acceleration



Introduction

Deep Learning is a subset of machine learning where models — typically neural networks — learn directly from data.Inspired by the structure and function of the human brain. Just like humans

Deep learning



PyTorch

- **Dynamic Computation Graph:** Easier debugging and flexible model building
- **Pythonic and Intuitive API:** Seamless integration with Python libraries
- **Strong Research and Industry Adoption:** Used by major companies and researchers
- **Excellent GPU Acceleration:** Optimised for performance on GPUs and TPUs

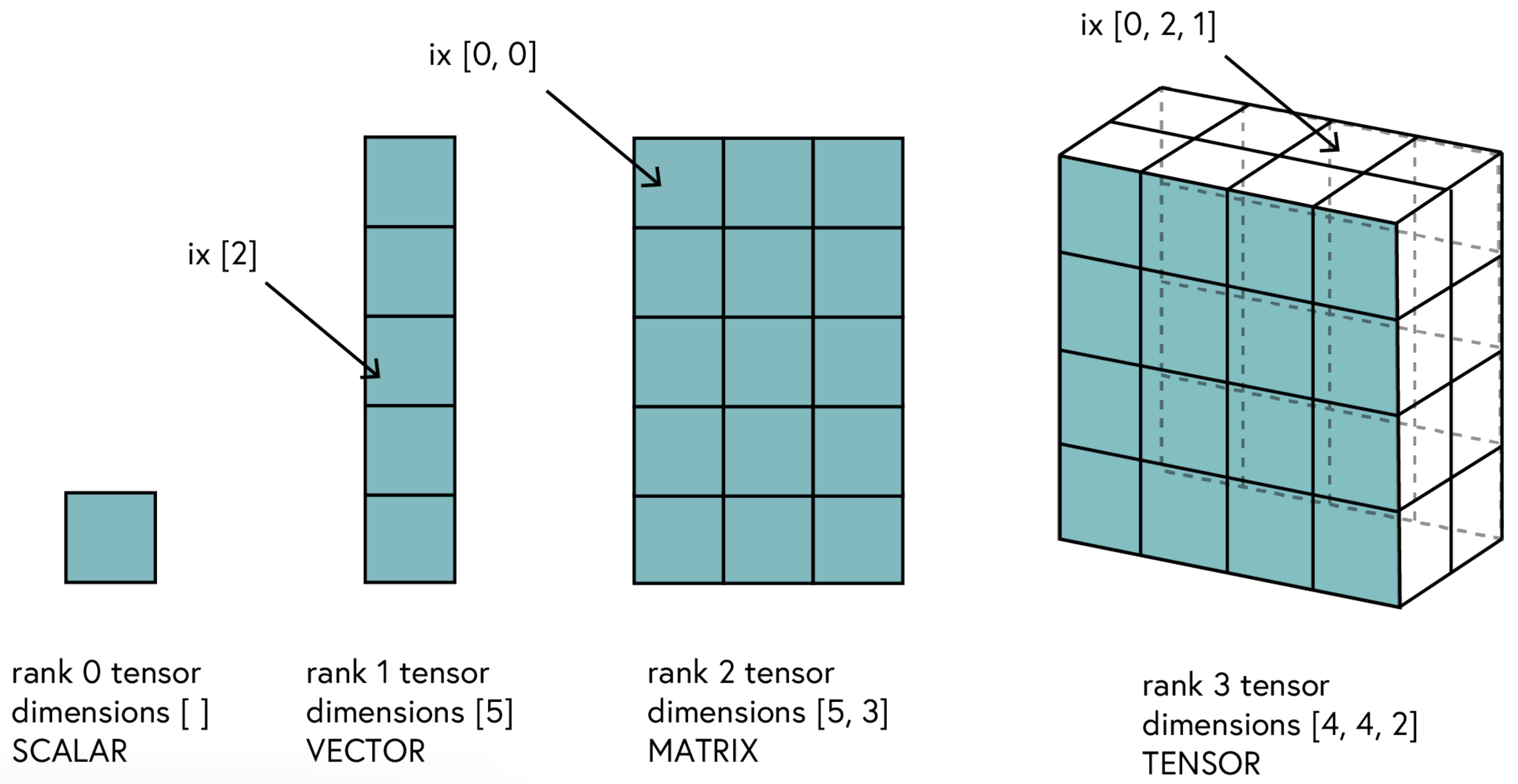
Why PYTORCH

Feature	PyTorch	TensorFlow	Keras
Ease of Use	High (Pythonic, dynamic computation graph)	Moderate (Static graph by default, more setup)	Very High (High-level API)
Flexibility	High	Moderate	Low (abstracted API)
Performance	High	Very High (Optimized for deployment)	Moderate
Debugging	Easy (Eager execution)	Harder (Graph-based execution)	Easy
GPU Support	Excellent	Excellent	Good
Industry Use	Research, Prototyping	Production, Deployment	Rapid Prototyping

Tensors

Definition: A generalization of vectors and matrices to higher dimensions

Why Tensors?




```python
import torch
# Different tensor ranks
scalar = torch.tensor(42)            # Rank 0
vector = torch.tensor([1, 2, 3])     # Rank 1
matrix = torch.tensor([[1, 2], [3, 4]])  # Rank 2
tensor_3d = torch.tensor([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])  # Rank 3t-wise addition
```

Creating Tensors

Basic Tensor Creation Methods

- torch.tensor() - from existing data
- torch.zeros(), torch.ones() - filled tensors
- torch.rand(), torch.randn() - random tensors
- torch.arange(), torch.linspace() - sequences
- torch.eye() - identity matrices

Data types can be specified with dtype parameter

```python
# Creating different tensors
data_tensor = torch.tensor([1, 2, 3, 4])
zeros = torch.zeros(2, 3)
ones = torch.ones(2, 3)
random_uniform = torch.rand(2, 3)      # Values from U(0,1)
random_normal = torch.randn(2, 3)      # Values from N(0,1)
sequence = torch.arange(0, 10, step=2) # [0, 2, 4, 6, 8]
linspace = torch.linspace(0, 1, steps=5)  # 5 evenly spaced points
identity = torch.eye(3)                # 3x3 identity matrix
```

TENSOR PROPERTIES

- **Working with Tensor Attributes**

- **Shape:** tensor.shape
- **Data type:** tensor.dtype
- **Device:** tensor.device
- **Accessing values:** tensor.item() for scalars
- **Converting types:** tensor.float(), tensor.int()

```python
# Exploring tensor properties
x = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
print(f"Shape: {x.shape}")        # Shape: torch.Size([2, 2])
print(f"Data type: {x.dtype}")    # Data type: torch.float32
print(f"Device: {x.device}")      # Device: cpu

# Converting types
x_int = x.int()
x_double = x.double()  # or x.to(torch.float64)
```

TENSOR indexing

Accessing Tensor Data

- **Basic indexing:** tensor[i, j]
- **Slicing:** tensor[1:3]

Advanced indexing techniques:

- **Boolean masks:** tensor[tensor > 0]
- **Negative indexing:** tensor[-1] (last element)
- **Using ellipsis:** tensor[..., 0]

```python
# Various indexing techniques
matrix = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Basic indexing and slicing
element = matrix[1, 2]      # Value at row 1, column 2: 6
row = matrix[1]             # Second row: [4, 5, 6]
column = matrix[:, 1]       # Second column: [2, 5, 8]
submatrix = matrix[0:2, 1:] # Top-right 2x2: [[2, 3], [5, 6]]

# Advanced indexing
mask = matrix > 5           # Boolean mask
values = matrix[mask]       # Values > 5: [6, 7, 8, 9]
corners = matrix[[0, -1], [0, -1]]  # Diagonal corners: [1, 9]
```

BASIC TENSOR OPERATIONS

Common Operations

- **Arithmetic:** +, -, *, /
- **Element-wise operations:** torch.sqrt(), torch.pow()
- **Reduction:** torch.sum(), torch.mean()
- **Comparisons:** >, <, ==
- **In-place operations:** tensor.add_(1) (note the underscore)

```python
# Basic operations
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

c = a + b                    # [5, 7, 9]
d = a * b                    # [4, 10, 18] (element-wise)
e = torch.sqrt(b)            # [2.0, 2.236, 2.449]

# Reduction operations
total = torch.sum(a)         # 6
mean_value = torch.mean(a.float())  # 2.0

# In-place operations
a.add_(10)                   # a becomes [11, 12, 13]
```

MATRIX OPERATIONS

Linear Algebra with PyTorch

- **Matrix multiplication:** @ or torch.matmul()
- **Transposition:** .T or torch.transpose()
- **Inverse:** torch.inverse()
- **Determinant:** torch.det()
- **Eigenvalues:** torch.eig()
- **SVD:** torch.svd()

```python
# Linear algebra operations
a = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
b = torch.tensor([[5, 6], [7, 8]], dtype=torch.float32)

# Matrix multiplication
c = a @ b                    # or torch.matmul(a, b)
# Result: [[19, 22], [43, 50]]

# Other operations
a_transpose = a.T            # [[1, 3], [2, 4]]
a_inv = torch.inverse(a)     # [[-2.0, 1.0], [1.5, -0.5]]
det_a = torch.det(a)         # -2.0

# SVD decomposition
U, S, V = torch.svd(a)
```

Broadcasting

Working with Different Shapes

- Automatic expansion of smaller tensors
- Rules follow NumPy broadcasting
- Eliminates need for explicit reshaping

Examples:

- Add scalar to matrix
- Multiply matrix by row/column vector
- Scale batches of data
- Powerful but requires understanding

```python
# Broadcasting examples
matrix = torch.tensor([[1, 2], [3, 4]])
scalar = torch.tensor(10)
row = torch.tensor([10, 20])
column = torch.tensor([[10], [20]])

# Broadcasting in action
matrix + scalar            # Add 10 to each element
# Result: [[11, 12], [13, 14]]

matrix * row               # Multiply each row by [10, 20]
# Result: [[10, 40], [30, 80]]

matrix + column            # Add column to each column
# Result: [[11, 12], [23, 24]]

# 3D example
batch = torch.randn(32, 3, 224, 224)  # Batch of images
scale = torch.tensor([0.5, 1.0, 0.8]) # Per-channel scale
scale = scale.view(1, 3, 1, 1)        # Reshape for broadcasting
normalized = batch * scale            # Scale each channel separately
```

RESHAPING

Changing Tensor Dimensions

- **reshape()** - new shape, possibly new memory
- **view()** - new shape, same memory (must be contiguous)
- **squeeze()** - remove dimensions of size 1
- **unsqueeze()** - add dimension of size 1
- **expand()** - broadcast dimensions without copying

```python
# Reshaping examples
x = torch.tensor([1, 2, 3, 4, 5, 6])

# Different reshape methods
reshaped = x.reshape(2, 3)          # [[1, 2, 3], [4, 5, 6]]
viewed = x.view(3, 2)               # [[1, 2], [3, 4], [5, 6]]

# Adding/removing dimensions
x_unsqueezed = x.unsqueeze(0)       # Add dimension: [1, 2, 3, 4, 5, 6] -> [[1, 2, 3, 4, 5, 6]]
single_dim = torch.tensor([7])      # Shape: [1]
squeezed = single_dim.squeeze()     # Shape: [] (scalar)

# Expand example
a = torch.tensor([1, 2, 3])         # Shape: [3]
b = a.unsqueeze(0)                  # Shape: [1, 3]
expanded = b.expand(4, 3)           # Shape: [4, 3], repeated rows without copying data
```

Using autograd

Computing Gradients

- Enable tracking with **requires_grad=True**
- Build computation graph through operations
- Call **backward()** to compute gradients
- Access gradients via **tensor.grad**

```python
# Complex function with multiple inputs
x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(3.0, requires_grad=True)

# f(x, y) = x²y + y³
z = x*x*y + y*y*y

# Compute gradients
z.backward()

# ∂z/∂x = 2xy = 2*2*3 = 12
# ∂z/∂y = x² + 3y² = 4 + 3*9 = 31
print(f"∂z/∂x: {x.grad}")  # 12.0
print(f"∂z/∂y: {y.grad}")  # 31.0

# Gradient accumulation
x = torch.tensor(1.0, requires_grad=True)
y = x * 2
y.backward()
print(f"First gradient: {x.grad}")  # 2.0

# Gradient accumulation (need to zero first)
x.grad.zero_()
z = x * 3
z.backward()
print(f"Second gradient: {x.grad}")  # 3.0
```

LOADING DATA

From Raw Data to Tensors

- Common data sources: CSV, images, text
- Using pandas to load structured data
- Converting to tensors:
  - **df = pd.read_csv('data.csv')**
  - **tensor = torch.tensor(df.values)**
- DataFrames as an intermediate representation

```python
import pandas as pd

# Load CSV data
df = pd.read_csv('data.csv')
print(f"DataFrame shape: {df.shape}")

# Convert specific columns to tensors
features = torch.tensor(df[['feature1', 'feature2', 'feature3']].values, dtype=torch.float32)
labels = torch.tensor(df['target'].values, dtype=torch.float32)

print(f"Features shape: {features.shape}")
print(f"Labels shape: {labels.shape}")
```

Using The gpu

Leveraging Hardware

- **Check availability:** torch.cuda.is_available()
- **Select device:** device = torch.device('cuda')
- **Move tensors to device:** tensor = tensor.to(device)

When to use GPU:

- Large tensors/datasets
- Computationally expensive operations
- Deep learning model training
- Keep all tensors on same device for efficiency

```python
# GPU handling
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Create tensor and move to appropriate device
x = torch.randn(1000, 1000)
x = x.to(device)

# Create model and move to device
model = MyNeuralNetwork().to(device)


# Check device of tensor
print(f"Tensor is on: {x.device}")

# Multiple GPUs
if torch.cuda.device_count() > 1:
    print(f"Using {torch.cuda.device_count()} GPUs!")
```

BEST PRACTICES

- Match tensor types before operations
- Understand broadcasting rules
- Keep track of your tensor devices
- Leverage PyTorch's documentation