Introduction to PyTorch

In this notebook you will learn the fundamentals of PyTorch — the most widely used deep learning framework in research and industry.

By the end of this notebook you will be able to:

- Create and manipulate **tensors** (PyTorch's core data structure)
- Reshape tensors confidently
- Build a **custom Dataset** and load data with a **DataLoader**
- Apply **data augmentation** and preprocessing transforms

📌 **Prerequisites:** basic Python and NumPy familiarity.

1. What is PyTorch?

PyTorch is an open-source deep learning framework developed by Meta AI. It is built around two key ideas:

1. **Tensor computation** — like NumPy, but with GPU acceleration.
2. **Automatic differentiation** — PyTorch tracks operations on tensors so it can compute gradients automatically (you will use this heavily when training models in the next notebook).

To install PyTorch, visit https://pytorch.org/get-started/locally/ and follow the instructions for your system.

```
# Typical CPU-only install
pip install torch torchvision
```

2. Working with Tensors

A **tensor** is the fundamental data structure in PyTorch — think of it as a generalisation of a __________ array:

Concept	NumPy	PyTorch
1-D array	`np.array([1,2,3])`	`torch.tensor([1,2,3])`
2-D array	`np.zeros((3,4))`	`torch.zeros((3,4))`
N-D array	`ndarray`	`Tensor`

The key advantage over NumPy: tensors can live on a **GPU** and support automatic differentiation.

2.1 Creating Tensors

Fill in the missing function names to create each type of tensor:

In [1]:
```
import torch

tensor_random = torch.rand((2, 3))
tensor_zeros = torch.zeros((4, 4))
tensor_ones = torch.ones((3, 3))
tensor_list = torch.tensor([[1, 2, 3], [4, 5, 6]])

print("Random:\n", tensor_random)
print("Zeros:\n", tensor_zeros)
print("Ones:\n", tensor_ones)
print("From list:\n", tensor_list)
```

```
Random:
 tensor([[0.6635, 0.7451, 0.3933],
        [0.7099, 0.2604, 0.7642]])
Zeros:
 tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
Ones:
 tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
From list:
 tensor([[1, 2, 3],
        [4, 5, 6]])
```

2.2 Basic Tensor Operations

Fill in the correct operators and function name:

- Element-wise addition: `c = a ______ b`
- Element-wise multiplication: `d = a ______ b`
- Matrix multiplication: `e = torch.________(a, b)`

In [2]:
```
a = torch.tensor([[1.0, 2.0], [3.0, 4.0]])
b = torch.tensor([[5.0, 6.0], [7.0, 8.0]])

c = a + b
d = a * b
e = torch.matmul(a, b)

print("a + b:\n", c)
```

```
print("a * b (element-wise):\n", d)
print("a @ b (matmul):\n", e)
```

```
a + b:
 tensor([[ 6.,  8.],
        [10., 12.]])
a * b (element-wise):
 tensor([[ 5., 12.],
        [21., 32.]])
a @ b (matmul):
 tensor([[19., 22.],
        [43., 50.]])
```

2.3 Inspecting Tensor Properties

Three attributes you will check constantly when debugging:

- **`.shape`** — dimensions of the tensor (e.g. `torch.Size([2, 3])`)
- **`.dtype`** — element data type (e.g. `torch.float32`)
- **`.device`** — where the tensor lives (`cpu` or `cuda:0`)

Fill in the attribute names:

In [3]:
```
tensor = torch.rand(2, 3)

print("Shape:", tensor.shape)
print("Data Type:", tensor.dtype)
print("Device:", tensor.device)
```

```
Shape: torch.Size([2, 3])
Data Type: torch.float32
Device: cpu
```

2.4 Moving Tensors Between CPU and GPU

GPU training can be 10–100× faster than CPU for large models. PyTorch makes it easy to move data between devices. The standard pattern is to detect the available device once at the top of your script and then move everything there.

Fill in the missing method name and variable:

In [4]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

tensor_gpu = tensor.to(device)
print("Tensor device:", tensor_gpu.device)
```

```
Using device: cpu
Tensor device: cpu
```

3. Tensor Reshaping

Reshaping is one of the most common operations you will perform — for example, flattening an image before passing it to a linear layer, or adding a batch dimension.

Method	What it does
`.view(shape)`	Reinterpret the data with a new shape — **zero-copy**, requires contiguous memory
`.reshape(shape)`	Same as view but works even if memory is non-contiguous (safer default)
`.squeeze()`	Remove all dimensions of size 1
`.unsqueeze(dim)`	Insert a new dimension of size 1 at position `dim`
`.permute(dims)`	Reorder dimensions (useful when converting between channel-first and channel-last)

💡 Use `-1` in a shape to let PyTorch infer that dimension automatically.

Fill in the blanks to perform the described reshape operations:

In [5]:
```
t = torch.arange(24)
print("Original shape:", t.shape)

t_2d = t.reshape(4, 6)
print("Reshaped to (4, 6):\n", t_2d)

t_flat = t_2d.reshape(-1)
print("Flattened:", t_flat.shape)

t_3d = t.reshape(2, 3, 4)
print("Reshaped to (2, 3, 4):\n", t_3d)
```

```
Original shape: torch.Size([24])
Reshaped to (4, 6):
 tensor([[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11],
        [12, 13, 14, 15, 16, 17],
        [18, 19, 20, 21, 22, 23]])
Flattened: torch.Size([24])
Reshaped to (2, 3, 4):
 tensor([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]])
```

Fill in the blanks for `squeeze` and `unsqueeze`:

In [6]:
```
t = torch.rand(3, 1, 4)
print("Original shape:", t.shape)

t_sq = t.squeeze()
```

```
print("After squeeze:", t_sq.shape)

t_unsq = t_sq.unsqueeze(0)
print("After unsqueeze(0):", t_unsq.shape)
```

```
Original shape: torch.Size([3, 1, 4])
After squeeze: torch.Size([3, 4])
After unsqueeze(0): torch.Size([1, 3, 4])
```

Fill in the `permute` call to convert from channel-first `(B, C, H, W)` to channel-last `(B, H, W, C)`:

In [7]:
```
img = torch.rand(8, 3, 32, 32)
img_hwc = img.permute(0, 2, 3, 1)
print("Channel-first:", img.shape)
print("Channel-last: ", img_hwc.shape)
```

```
Channel-first: torch.Size([8, 3, 32, 32])
Channel-last:  torch.Size([8, 32, 32, 3])
```

4. Datasets and DataLoaders

PyTorch separates **what your data is** from **how it is served to the model**:

- A **`Dataset`** defines *what* your data looks like — it knows how many samples there are and how to retrieve the i-th one.
- A **`DataLoader`** wraps a Dataset and handles *how* data is delivered — batching, shuffling, parallel loading.

```
Dataset  →  DataLoader  →  Training loop
 (storage)    (batching)
```

A Dataset represents the __________ data, defining how each sample is stored and accessed.

A DataLoader helps in __________ the data into manageable batches for training.

Why not just load everything into a list?

Real datasets (ImageNet: 1.2 M images) do not fit in RAM. The Dataset + DataLoader pattern lets you load only what you need, when you need it.

5. Creating a Custom Dataset

To create a custom Dataset, subclass `torch.utils.data.Dataset` and implement exactly **two methods**:

- `__len__()` — returns the total number of samples
- `__getitem__(index)` — returns the sample at position `index`

The DataLoader will call these methods internally. Fill in the missing return value:

In [8]:
```
from torch.utils.data import Dataset, DataLoader
from PIL import Image
import os

class CustomImageDataset(Dataset):
    def __init__(self, image_dir, transform=None):
        self.image_dir = image_dir
        self.image_files = os.listdir(image_dir)
        self.transform = transform

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, index):
        img_path = os.path.join(self.image_dir, self.image_files[index])
        image = Image.open(img_path).convert("RGB")

        if self.transform:
            image = self.transform(image)

        return image
```

6. Using DataLoader

Key `DataLoader` parameters:

Parameter	Description
`batch_size`	How many samples per batch
`shuffle`	Randomise order each epoch (use `True` for training, `False` for validation)
`num_workers`	Number of parallel processes for loading (0 = main process only)

Fill in sensible values for a training DataLoader:

In [9]:
```
# dataset = CustomImageDataset(image_dir="path/to/images")

# dataloader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=

# for batch in dataloader:
#     print(batch.shape)
#     break

print("Uncomment the code above once you have a real image directory.")
```

```
Uncomment the code above once you have a real image directory.
```

7. Loading a Built-in Dataset (MNIST)

`torchvision.datasets` ships with many standard datasets (MNIST, CIFAR-10, ImageNet, ...). This is the quickest way to get started without managing files yourself.

Before loading, we define a **transform pipeline** — a sequence of preprocessing steps applied to every sample on the fly.

Fill in the correct transform names and the expected dataset length:

In [10]:
```
import torchvision.transforms as transforms
from torchvision import datasets

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(0.5, 0.5)
])

train_dataset = datasets.MNIST(root='data', train=True, transform=transform,

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

print("Training samples:", len(train_dataset))  # Expected: 60000
print("Number of batches:", len(train_loader))

images, labels = next(iter(train_loader))
print("Batch shape:", images.shape)
print("Labels shape:", labels.shape)
```

```
  0%|          | 0.00/9.91M [00:00<?, ?B/s]
  1%|          | 65.5k/9.91M [00:00<00:31, 310kB/s]
  2%|▏         | 197k/9.91M [00:00<00:14, 686kB/s]
  3%|▎         | 328k/9.91M [00:00<00:11, 870kB/s]
  7%|▋         | 721k/9.91M [00:00<00:05, 1.79MB/s]
 14%|█▎        | 1.34M/9.91M [00:00<00:02, 2.98MB/s]
 17%|█▋        | 1.67M/9.91M [00:00<00:02, 2.92MB/s]
 26%|██▋       | 2.62M/9.91M [00:00<00:01, 4.73MB/s]
 41%|████▏     | 4.10M/9.91M [00:01<00:00, 7.24MB/s]
 49%|████▉     | 4.85M/9.91M [00:01<00:00, 6.71MB/s]
 56%|█████▌    | 5.54M/9.91M [00:01<00:00, 6.54MB/s]
 63%|██████▎   | 6.23M/9.91M [00:01<00:00, 6.29MB/s]
 69%|██████▉   | 6.88M/9.91M [00:01<00:00, 6.08MB/s]
 76%|███████▌  | 7.50M/9.91M [00:01<00:00, 5.82MB/s]
 82%|████████▏ | 8.09M/9.91M [00:01<00:00, 4.32MB/s]
 87%|████████▋ | 8.65M/9.91M [00:01<00:00, 4.49MB/s]
 92%|█████████▏| 9.14M/9.91M [00:02<00:00, 4.42MB/s]
 97%|█████████▋| 9.63M/9.91M [00:02<00:00, 4.33MB/s]
100%|██████████| 9.91M/9.91M [00:02<00:00, 4.31MB/s]

  0%|          | 0.00/28.9k [00:00<?, ?B/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 253kB/s]
```

```
100%|██████████| 28.9k/28.9k [00:00<00:00, 242kB/s]

  0%|          | 0.00/1.65M [00:00<?, ?B/s]
  2%|▏         | 32.8k/1.65M [00:00<00:05, 294kB/s]
  6%|▌         | 98.3k/1.65M [00:00<00:03, 461kB/s]
 12%|█▏        | 197k/1.65M [00:00<00:02, 652kB/s]
 24%|██▍       | 393k/1.65M [00:00<00:01, 1.08MB/s]
 48%|████▊     | 786k/1.65M [00:00<00:00, 1.95MB/s]
 74%|███████▍  | 1.21M/1.65M [00:00<00:00, 2.59MB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 2.41MB/s]

  0%|          | 0.00/4.54k [00:00<?, ?B/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 12.4MB/s]
Training samples: 60000
Number of batches: 1875
Batch shape: torch.Size([32, 1, 28, 28])
Labels shape: torch.Size([32])
```

8. Data Augmentation & Preprocessing

Data augmentation artificially increases the diversity of your training set by applying random transformations — without collecting new data. Common benefits:

- Reduces overfitting
- Makes the model more robust to real-world variation (different lighting, orientations, etc.)

⚠️ Important: augmentations should only be applied to the **training set**, not to validation/test sets.

Fill in the missing transform names:

In [11]:
```
import torchvision.transforms as transforms

train_transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(degrees=15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])

val_transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])
```

Applying Transforms and Exploring the Dataset

Let's make this concrete. Below we apply three different transform pipelines to MNIST and compare what comes out:

1. **No transform** — raw PIL image
2. **Minimal** — `ToTensor()` only
3. **Full augmentation** — resize, flip, rotation, normalise

We then inspect dataset size, batch shapes, pixel value ranges, and visualise a sample batch.

In [12]:
```
import torch
import torchvision.transforms as transforms
from torchvision import datasets
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np

raw_dataset = datasets.MNIST(root='data', train=True, transform=None, downlo

minimal_transform = transforms.Compose([
    transforms.ToTensor()
])
minimal_dataset = datasets.MNIST(root='data', train=True, transform=minimal_

augment_transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(degrees=15),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])
augmented_dataset = datasets.MNIST(root='data', train=True, transform=augmer

test_dataset = datasets.MNIST(root='data', train=False, transform=minimal_tr

print("=== Dataset sizes ===")
print(f"Training samples : {len(minimal_dataset):,}")
print(f"Test samples     : {len(test_dataset):,}")
print(f"Total            : {len(minimal_dataset) + len(test_dataset):,}")
print(f"Classes          : {minimal_dataset.classes}")
print(f"Number of classes: {len(minimal_dataset.classes)}")
```

```
=== Dataset sizes ===
Training samples : 60,000
Test samples     : 10,000
Total            : 70,000
Classes          : ['0 - zero', '1 - one', '2 - two', '3 - three', '4 - fou
r', '5 - five', '6 - six', '7 - seven', '8 - eight', '9 - nine']
Number of classes: 10
```

In [13]:
```
print("=== Single sample inspection ===")

raw_img, label = raw_dataset[0]
```

```
print(f"Raw PIL image  : type={type(raw_img).__name__}, size={raw_img.size},

min_img, label = minimal_dataset[0]
print(f"Minimal tensor : shape={min_img.shape}, dtype={min_img.dtype}")
print(f"                 min={min_img.min():.3f}, max={min_img.max():.3f}")

aug_img, label = augmented_dataset[0]
print(f"Augmented tensor: shape={aug_img.shape}, dtype={aug_img.dtype}")
print(f"                  min={aug_img.min():.3f}, max={aug_img.max():.3f}")
print()
# Q: Why is the min of the augmented tensor negative but the minimal tensor'
# A: The augmented pipeline applies Normalize(mean=0.5, std=0.5) which shift
```

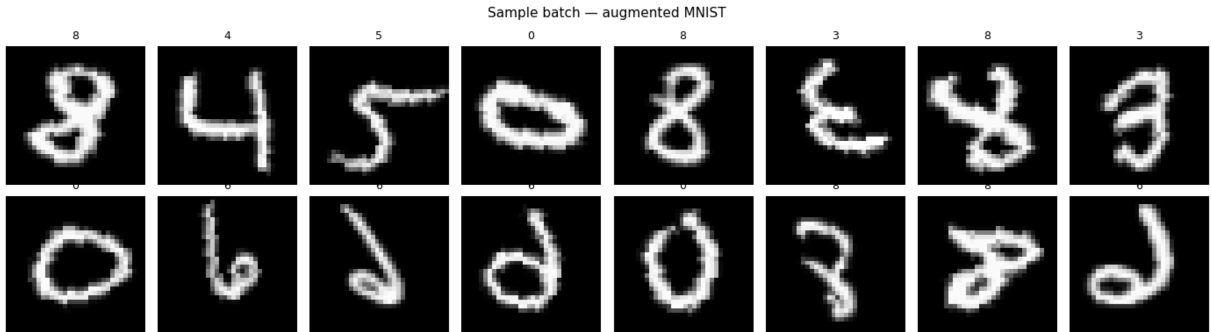
```
=== Single sample inspection ===
Raw PIL image  : type=Image, size=(28, 28), label=5
Minimal tensor : shape=torch.Size([1, 28, 28]), dtype=torch.float32
                 min=0.000, max=1.000
Augmented tensor: shape=torch.Size([1, 32, 32]), dtype=torch.float32
                  min=-1.000, max=0.984
```

In [14]:
```
loader_minimal   = DataLoader(minimal_dataset,   batch_size=32, shuffle=True
loader_augmented = DataLoader(augmented_dataset, batch_size=32, shuffle=True

imgs_min, labels = next(iter(loader_minimal))
print(f"Minimal   batch — images: {imgs_min.shape}, labels: {labels.shape}")
# Expected shape: (32, 1, 28, 28)

imgs_aug, labels = next(iter(loader_augmented))
print(f"Augmented batch — images: {imgs_aug.shape}, labels: {labels.shape}")
# Expected shape: (32, 1, 32, 32)

print(f"\nBatches per epoch (batch_size=32): {len(loader_minimal)}")

unique, counts = torch.unique(labels, return_counts=True)
for cls, cnt in zip(unique.tolist(), counts.tolist()):
    print(f"  Class {cls}: {cnt} sample(s)")
```

```
Minimal   batch — images: torch.Size([32, 1, 28, 28]), labels: torch.Size([3
2])
Augmented batch — images: torch.Size([32, 1, 32, 32]), labels: torch.Size([3
2])

Batches per epoch (batch_size=32): 1875
  Class 0: 5 sample(s)
  Class 1: 2 sample(s)
  Class 2: 4 sample(s)
  Class 3: 1 sample(s)
  Class 4: 3 sample(s)
  Class 5: 4 sample(s)
  Class 6: 1 sample(s)
  Class 7: 2 sample(s)
  Class 8: 6 sample(s)
  Class 9: 4 sample(s)
```

In [15]:
```
imgs, labels = next(iter(loader_augmented))
```

```python
fig, axes = plt.subplots(2, 8, figsize=(14, 4))
fig.suptitle("Sample batch — augmented MNIST", fontsize=11)

for i, ax in enumerate(axes.flat):
    img = imgs[i].squeeze().numpy()
    img = (img * 0.5) + 0.5
    ax.imshow(img, cmap='gray', vmin=0, vmax=1)
    ax.set_title(str(labels[i].item()), fontsize=9)
    ax.axis('off')

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



Summary

Concept	Key class / function
Create tensor	`torch.tensor()`, `torch.rand()`, `torch.zeros()`, `torch.ones()`
Reshape tensor	`.reshape()`, `.view()`, `.squeeze()`, `.unsqueeze()`, `.permute()`
Move to GPU	`.to(device)`
Custom dataset	Subclass `Dataset`, implement `__len__` and `__getitem__`
Load in batches	`DataLoader(dataset, batch_size=…, shuffle=…)`
Preprocessing	`transforms.Compose([…])`

Next notebook: building neural networks with `nn.Module` and writing a training loop.