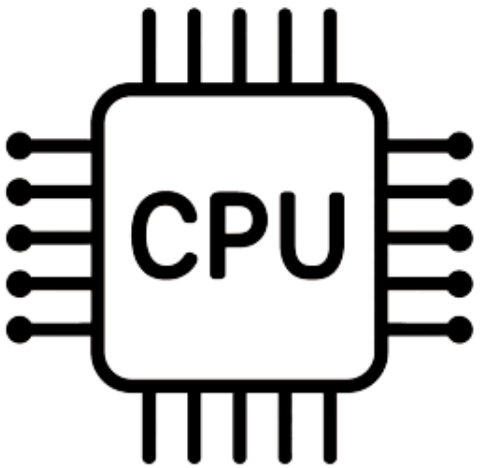
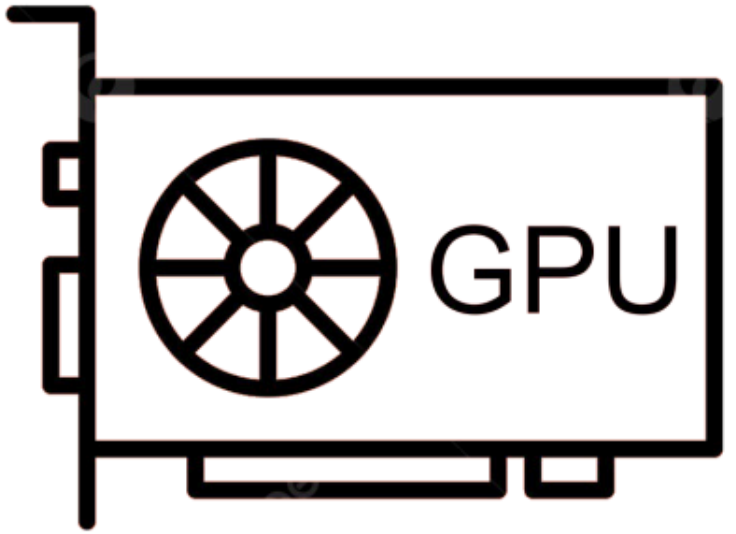
Module 2 - Session 3: Device Management and Image Classification Setup

Session 3

Device Management





Every tensor and model lives on a device

- CPU (default)
- GPU (accelerator)

Key rule: Model and data must be on the same device!

CPU vs GPU

CPU:

- Default device
- General purpose
- Sequential operations

GPU:

- Accelerator
- Parallel operations
- **10-15x faster for training**

Checking for GPU

```
torch.cuda.is_available()  # Returns True if GPU available
```

Common pattern:

```
device = torch.device('cuda' if torch.cuda.is_available()
                      else 'cpu')
```

Moving Model to Device

```
model = MyModel()
model = model.to(device)  # Move model to device
```

Puts model’s parameters on the selected device

Moving Data to Device

```
for batch in dataloader:
    inputs, targets = batch
    inputs = inputs.to(device)
    targets = targets.to(device)
    # ... rest of training
```

Move each batch within the training loop

Checking Device Location

```
# For tensors
tensor.device

# For models (check a parameter)
next(model.parameters()).device
```

Common Mistake with `.to()`

`.to()` doesn't change tensor in place

It creates a new one!

```
# Wrong
tensor.to(device)  # Result is discarded!

# Right
tensor = tensor.to(device)  # Reassign
```

Complete Training Loop with Device Management

```
device = torch.device('cuda' if torch.cuda.is_available()
                      else 'cpu')
model = MyModel().to(device)

for batch in dataloader:
    inputs, targets = batch[0].to(device), batch[1].to(device)
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_fn(outputs, targets)
    loss.backward()
    optimizer.step()
```

Key steps: 1. Choose device up front 2. Move model once 3. Move data in every batch

GPU Memory Limits

GPU memory is limited

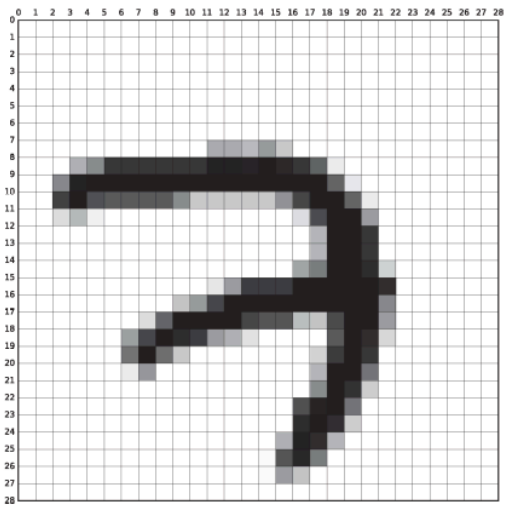
Error if batch size too large:

```
RuntimeError: CUDA out of memory
```

Solution: Lower batch size (32-64 is good starting point)

Building Your First Image Classifier

28x28 Image



(a) MNIST sample belonging to the digit '7'.

Dataset: 60,000 images

(b) 100 samples from the MNIST training set.

MNIST Dataset:

- 60,000 training images
- 10,000 test images
- 28×28 pixels, grayscale
- 10 classes (digits 0-9)

Setting Up the Data Pipeline

```python
import torchvision
from torchvision import transforms

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

ToTensor: converts to tensors, scales 0-255 → 0-1

Normalize: centers around 0 using dataset mean/std

Loading the Dataset

```
train_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)

test_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=False,
    transform=transform
)
```

TorchVision handles downloading and organizing

Creating DataLoaders

```
train_loader = DataLoader(
    train_dataset,
    batch_size=64,
    shuffle=True
)

test_loader = DataLoader(
    test_dataset,
    batch_size=1000,
    shuffle=False
)
```

Training: shuffle=True (mix up order each epoch)

Testing: shuffle=False (order doesn’t matter)

Building the Model Architecture

```python
class MNISTClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(784, 128)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        return x
```

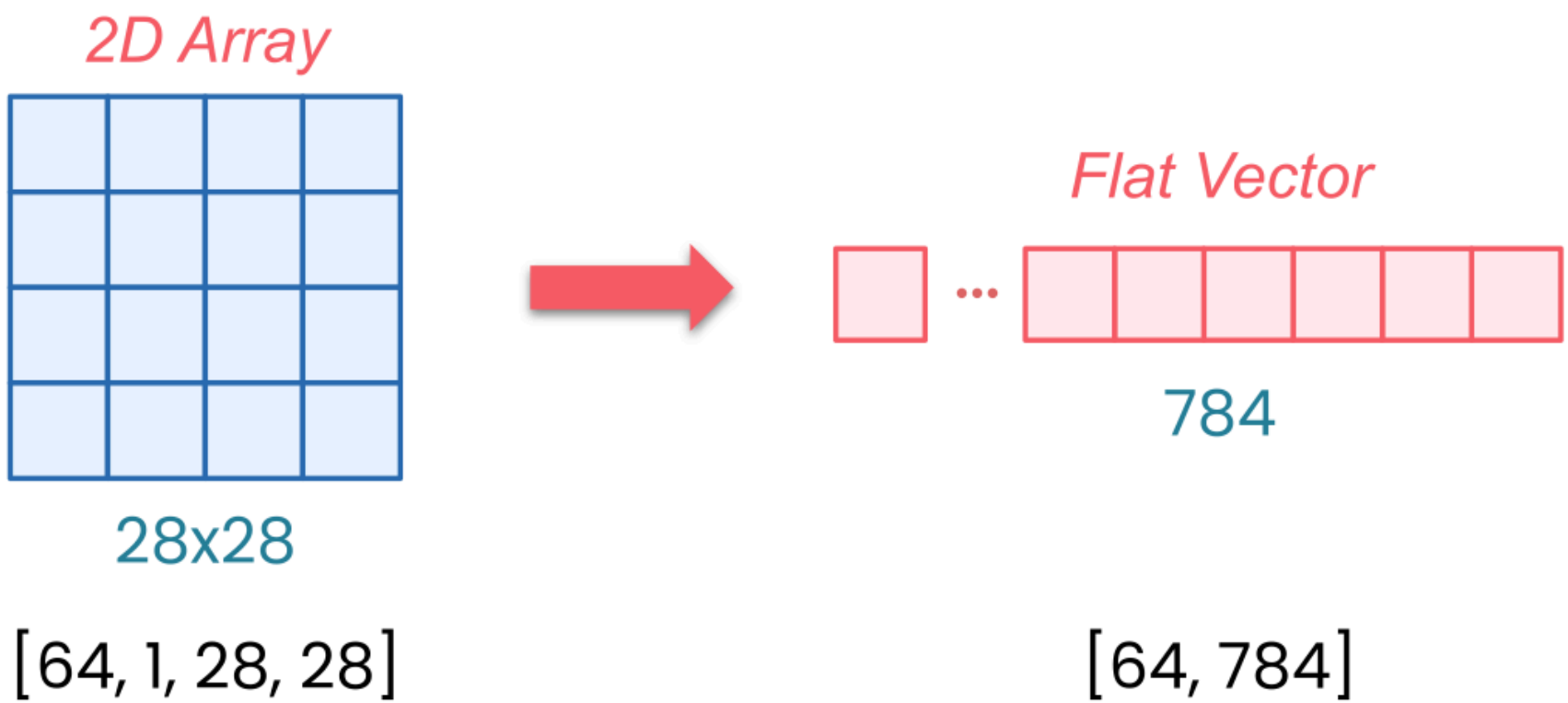
Why Flatten?

MNIST images: shape `[1, 28, 28]` (channels, height, width)

With batch: shape `[64, 1, 28, 28]`

Linear layers expect: flat vectors `[batch, features]`

Flatten: `[64, 1, 28, 28]` → `[64, 784]`



Model Architecture Breakdown

Linear(784, 128):

- 784 pixel values → 128 hidden features

ReLU:

- Activation function (non-linearity)

Linear(128, 10):

- 128 features → 10 outputs (one per digit class)

What’s Next?

In **Session 4: Training and Evaluating Your Classifier** we learn:

- Setting up loss function and optimizer
- Writing the training loop
- Evaluating on test set
- Watching your model learn!