```
import torch, torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
#torch: Main PyTorch library.
#torch.nn: Provides layers and neural network utilities.
#torchvision: Useful for image datasets (like MNIST), models, and
transforms.
#transforms.ToTensor(): Converts images to tensors (and normalizes them to
[0, 1]).
#DataLoader: Helps in loading data in batches, shuffling, and parallel
loading.


# 1. Load MNIST dataset
transform = transforms.ToTensor()
#Define a transformation to convert PIL images to PyTorch tensors.

train_data = torchvision.datasets.MNIST(root='./data', train=True,
transform=transform, download=True)
test_data = torchvision.datasets.MNIST(root='./data', train=False,
transform=transform, download=True)
#Download and load the MNIST training and test datasets with the
transformation applied.

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64)
#batch_size=64: Process 64 images at a time.
#shuffle=True: Shuffle training data for better learning.

# 2. Define CNN Model
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
#CNN is a subclass of PyTorch's nn.Module.
#super().__init__() initializes the base class


        self.net = nn.Sequential(
            nn.Conv2d(1, 16, 3, padding=1), nn.ReLU(),
#First convolution layer: input channel 1 (grayscale), output 16 filters,
3×3 kernel, padding=1 to keep size same.
#Followed by ReLU activation.

            nn.MaxPool2d(2),
```

```
#Max pooling with 2×2 window: Reduces spatial dimensions by half (from
28×28 to 14×14).


            nn.Conv2d(16, 32, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),
#Another conv layer: 16 input channels → 32 output channels. Followed by
ReLU and another pooling (14×14 → 7×7).

            nn.Flatten(),
#Flatten the output from 4D (batch, channels, height, width) to 2D (batch,
features).

            nn.Linear(32*7*7, 128), nn.ReLU(),
            nn.Linear(128, 10)
#Fully connected layer: From 1568 features → 128 → 10 (digits 0–9).
        )
    def forward(self, x): return self.net(x)
#forward defines how the input passes through the network


# 3. Setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#Compute Unified Device Architecture
#Automatically selects GPU if available, otherwise CPU.


model = CNN().to(device)
#Instantiate the model and move it to the chosen device.


loss_fn = nn.CrossEntropyLoss()
#Define the loss function for classification.


optimizer = torch.optim.Adam(model.parameters())
#Use Adam optimizer to update the model's parameters.


# 4. Train
for epoch in range(2):
    for x, y in train_loader:
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()
        loss = loss_fn(model(x), y)
        loss.backward()
```

```
        optimizer.step()
    print(f"Epoch {epoch+1} done")

# 5. Evaluate
correct = total = 0
with torch.no_grad():
    for x, y in test_loader:
        x, y = x.to(device), y.to(device)
        pred = model(x).argmax(1)
        correct += (pred == y).sum().item()
        total += y.size(0)

print(f"Test Accuracy: {correct / total:.2f}")
```