Project: Handwritten Digit Recognition with MNIST

Goal: Train a neural network to classify handwritten digits (0–9) from the MNIST dataset.

Why this example?

* Real Dataset: MNIST is a benchmark dataset, freely downloadable, and mirrors real-world classification tasks.
* Scientific Tie-in: Think of it as recognizing patterns in experimental data (e.g., identifying peaks in a spectrum).
* Neural Network Learning: Introduces image data, convolutional layers (optional), and multi-class classification.

Step 1: Setup in PyCharm

1. Install required libraries (if not already installed):

bash

pip install numpy pandas matplotlib torch torchvision scikit-learn

* + torchvision provides easy access to MNIST and image-processing tools.

1. Create a new Python file (e.g., mnist\_digit\_recognition.py).

Step 2: Download MNIST

* PyTorch’s torchvision automatically downloads MNIST the first time you run the code. No manual download needed—it’ll save to your local machine (e.g., ~/.cache/torchvision).

Step 3: Code Example

Here’s the full code with explanations:

python

import numpy as np

import matplotlib.pyplot as plt

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

from torch.utils.data import DataLoader

# Step 1: Load and preprocess the MNIST dataset

transform = transforms.Compose([

transforms.ToTensor(), # Convert images to PyTorch tensors (0-1 range)

transforms.Normalize((0.1307,), (0.3081,)) # Normalize with MNIST mean and std

])

# Download and load training and test datasets

train\_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)

test\_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

# Create data loaders (like MATLAB's batch processing)

train\_loader = DataLoader(train\_dataset, batch\_size=64, shuffle=True)

test\_loader = DataLoader(test\_dataset, batch\_size=64, shuffle=False)

# Step 2: Visualize a few examples

def plot\_sample\_images(loader):

images, labels = next(iter(loader))

fig, axes = plt.subplots(1, 5, figsize=(10, 2))

for i in range(5):

axes[i].imshow(images[i][0], cmap='gray')

axes[i].set\_title(f'Label: {labels[i].item()}')

axes[i].axis('off')

plt.show()

plot\_sample\_images(train\_loader)

# Step 3: Define the neural network

class DigitClassifier(nn.Module):

def \_\_init\_\_(self):

super(DigitClassifier, self).\_\_init\_\_()

self.flatten = nn.Flatten() # Flatten 28x28 images to 784-element vector

self.layers = nn.Sequential(

nn.Linear(28\*28, 128), # Input: 784 pixels, Output: 128 neurons

nn.ReLU(),

nn.Linear(128, 64), # Hidden layer: 128 -> 64 neurons

nn.ReLU(),

nn.Linear(64, 10) # Output: 64 -> 10 (one per digit)

)

def forward(self, x):

x = self.flatten(x)

x = self.layers(x)

return x

# Instantiate the model

model = DigitClassifier()

# Step 4: Define loss function and optimizer

criterion = nn.CrossEntropyLoss() # For multi-class classification

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Step 5: Train the model

epochs = 5

train\_losses = []

for epoch in range(epochs):

model.train()

running\_loss = 0.0

for images, labels in train\_loader:

optimizer.zero\_grad() # Clear gradients

outputs = model(images) # Forward pass

loss = criterion(outputs, labels) # Compute loss

loss.backward() # Backward pass

optimizer.step() # Update weights

running\_loss += loss.item()

avg\_loss = running\_loss / len(train\_loader)

train\_losses.append(avg\_loss)

print(f'Epoch [{epoch+1}/{epochs}], Loss: {avg\_loss:.4f}')

# Step 6: Evaluate the model

model.eval()

correct = 0

total = 0

with torch.no\_grad():

for images, labels in test\_loader:

outputs = model(images)

\_, predicted = torch.max(outputs.data, 1) # Get index of max score

total += labels.size(0)

correct += (predicted == labels).sum().item()

accuracy = 100 \* correct / total

print(f'Test Accuracy: {accuracy:.2f}%')

# Step 7: Plot training loss

plt.plot(train\_losses, label='Training Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.title('Training Loss Over Time')

plt.legend()

plt.show()

# Step 8: Test on a few examples

def test\_samples(loader):

model.eval()

images, labels = next(iter(loader))

with torch.no\_grad():

outputs = model(images)

\_, predicted = torch.max(outputs, 1)

fig, axes = plt.subplots(1, 5, figsize=(10, 2))

for i in range(5):

axes[i].imshow(images[i][0], cmap='gray')

axes[i].set\_title(f'Pred: {predicted[i].item()}\nTrue: {labels[i].item()}')

axes[i].axis('off')

plt.show()

test\_samples(test\_loader)

Explanation of Key Steps

1. Data Loading:
   * torchvision.datasets.MNIST: Downloads and loads the dataset (60,000 train, 10,000 test images).
   * transforms: Normalizes pixel values (like MATLAB’s image preprocessing).
   * DataLoader: Batches data efficiently, similar to MATLAB’s minibatch approach.
2. Visualization:
   * plot\_sample\_images: Displays a few digits to confirm data loading (MATLAB’s imshow equivalent).
3. Neural Network:
   * Fully connected layers (nn.Linear) process flattened 28x28 images (784 inputs).
   * Output layer has 10 neurons (one per digit), no softmax since CrossEntropyLoss handles it internally.
4. Training:
   * nn.CrossEntropyLoss: Suits multi-class problems (unlike MSE for regression).
   * Loops over batches (train\_loader), a step up from the small dataset in the last example.
5. Evaluation:
   * Accuracy is computed by comparing predictions to true labels, a common metric in classification.
6. Results:
   * Plots training loss and shows predictions on test images for visual feedback.

Transition Tips

* MATLAB imread → torchvision: PyTorch handles image loading seamlessly.
* MATLAB Loops → PyTorch DataLoader: Batch processing is more Pythonic and efficient.
* MATLAB classperf → PyTorch Accuracy: Manual accuracy calculation replaces MATLAB’s built-in tools.

Next Steps

1. Run the Code: See how well it classifies digits (expect ~95% accuracy with this simple model).
2. Tweak It: Increase epochs, adjust batch\_size, or add layers (e.g., nn.Linear(128, 256)).
3. Upgrade: Replace fully connected layers with convolutional layers (nn.Conv2d) for better image handling—ask me if you want this version!
4. Science Twist: Adapt this to a chemistry dataset (e.g., classify molecular spectra)—I can help source one.

The MNIST dataset is downloaded automatically via the code, so you’re ready to go. Let me know how it runs or if you’d prefer a different dataset (e.g., chemical properties, physics simulations)!