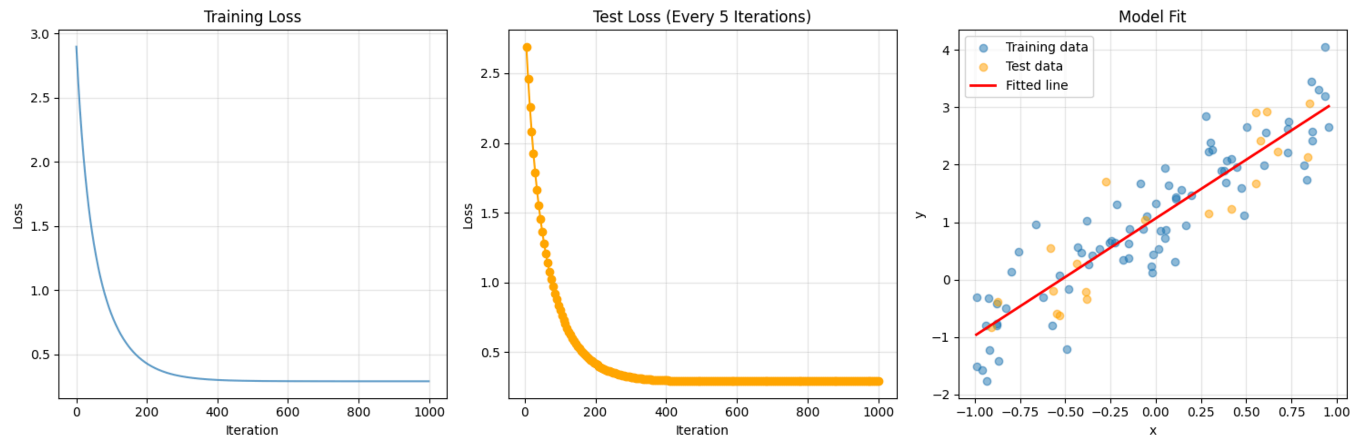
```
import torch
```

Linear Regression Using a Neural Network

```
import torch
import torch.nn as nn

# Generate 100 samples
num_samples = 100

random_samples = torch.rand(num_samples)
noise = torch.randn(num_samples) * 0.5

x = 2 * random_samples - 1
y = x * 2 + 1 + noise

# Split into train and test sets (80/20 split)
split_idx = int(0.8 * num_samples)

x_train = x[:split_idx].view(-1, 1)
y_train = y[:split_idx].view(-1, 1)

x_test = x[split_idx:].view(-1, 1)
y_test = y[split_idx:].view(-1, 1)

print(f"Training samples: {len(x_train)}, Test samples: {len(x_test)}")

# Construct Linear Model
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)

# Instantiate model
model = Model()

# Define loss and optimizer
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training parameters
num_epochs = 1000

# Storage for losses
train_losses = []
test_losses = []
test_loss_iterations = []

iteration = 0

# Training loop
for epoch in range(num_epochs):
    # Training mode
    model.train()

    # Zero the gradients
    optimizer.zero_grad()

    # Forward pass
    outputs = model(x_train)

    # Calculate loss
    loss = criterion(outputs, y_train)

    # Backward pass
    loss.backward()

    # Store training loss (FIXED: added parentheses)
    train_losses.append(loss.item())

    # Update parameters
```

```
        optimizer.step()

        iteration += 1

        # Evaluate on test set every 5 iterations
        if iteration % 5 == 0:
            model.eval()

            with torch.no_grad():
                test_pred = model(x_test)
                test_loss = criterion(test_pred, y_test)

                test_losses.append(test_loss.item())
                test_loss_iterations.append(iteration)

                if iteration % 50 == 0:  # Print less frequently
                    print(f"Iteration {iteration}: Train Loss = {loss.item():.4f}, Test Loss = {test_loss.item():.4f}")

            model.train()

    # Display learned parameters
    print(f"\nLearned weight: {model.linear.weight.item():.4f} (True: 2.0)")
    print(f"Learned bias: {model.linear.bias.item():.4f} (True: 1.0)")

    # Plot the results
    import matplotlib.pyplot as plt

    plt.figure(figsize=(15, 5))

    # Plot 1: Training loss over all iterations
    plt.subplot(1, 3, 1)
    plt.plot(train_losses, alpha=0.7)
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title('Training Loss')
    plt.grid(True, alpha=0.3)

    # Plot 2: Test loss every 5 iterations
    plt.subplot(1, 3, 2)
    plt.plot(test_loss_iterations, test_losses, 'o-', color='orange')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title('Test Loss (Every 5 Iterations)')
    plt.grid(True, alpha=0.3)

    # Plot 3: Fitted line vs actual data
    plt.subplot(1, 3, 3)
    plt.scatter(x_train.numpy(), y_train.numpy(), alpha=0.5, label='Training data')
    plt.scatter(x_test.numpy(), y_test.numpy(), alpha=0.5, label='Test data', color='orange')

    # Plot the learned line
    x_plot = torch.linspace(x.min(), x.max(), 100).view(-1, 1)
    with torch.no_grad():
        y_plot = model(x_plot)
    plt.plot(x_plot.numpy(), y_plot.numpy(), 'r-', linewidth=2, label='Fitted line')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Model Fit')
    plt.legend()
    plt.grid(True, alpha=0.3)

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

```
Training samples: 80, Test samples: 20
Iteration 50: Train Loss = 1.3737, Test Loss = 1.3620
Iteration 100: Train Loss = 0.8157, Test Loss = 0.7989
Iteration 150: Train Loss = 0.5592, Test Loss = 0.5432
Iteration 200: Train Loss = 0.4299, Test Loss = 0.4165
Iteration 250: Train Loss = 0.3629, Test Loss = 0.3524
Iteration 300: Train Loss = 0.3280, Test Loss = 0.3201
Iteration 350: Train Loss = 0.3097, Test Loss = 0.3040
Iteration 400: Train Loss = 0.3002, Test Loss = 0.2962
Iteration 450: Train Loss = 0.2952, Test Loss = 0.2926
Iteration 500: Train Loss = 0.2926, Test Loss = 0.2910
Iteration 550: Train Loss = 0.2913, Test Loss = 0.2904
Iteration 600: Train Loss = 0.2905, Test Loss = 0.2902
Iteration 650: Train Loss = 0.2902, Test Loss = 0.2902
Iteration 700: Train Loss = 0.2900, Test Loss = 0.2904
Iteration 750: Train Loss = 0.2899, Test Loss = 0.2905
Iteration 800: Train Loss = 0.2898, Test Loss = 0.2906
Iteration 850: Train Loss = 0.2898, Test Loss = 0.2907
Iteration 900: Train Loss = 0.2898, Test Loss = 0.2907
Iteration 950: Train Loss = 0.2898, Test Loss = 0.2908
Iteration 1000: Train Loss = 0.2898, Test Loss = 0.2908

Learned weight: 2.0437 (True: 2.0)
Learned bias: 1.0660 (True: 1.0)
```



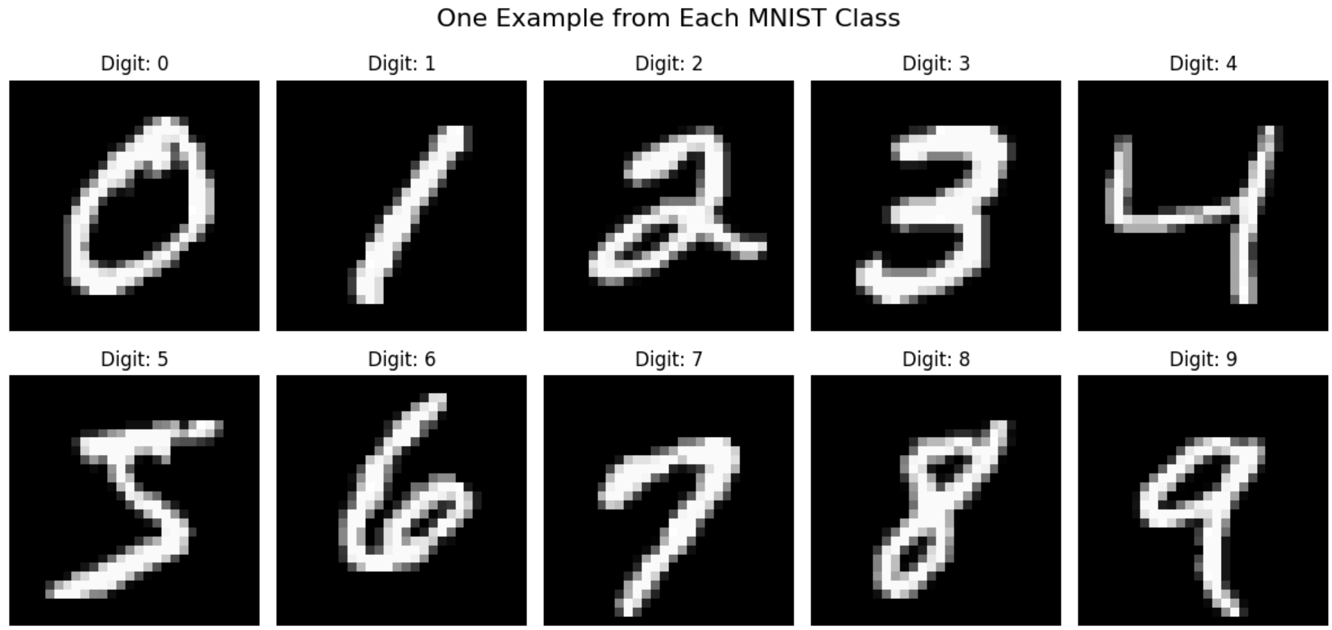
Classifying MIST dataset

```
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np

# 1. Create transform to convert data to tensors
transform = transforms.ToTensor()

# 2. Download and instantiate MNIST datasets
# Training dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)

# Testing dataset
test_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=False,             # Testing split
    download=True,
    transform=transform
)

print(f"Training samples: {len(train_dataset)}")
print(f"Test samples: {len(test_dataset)}")
```

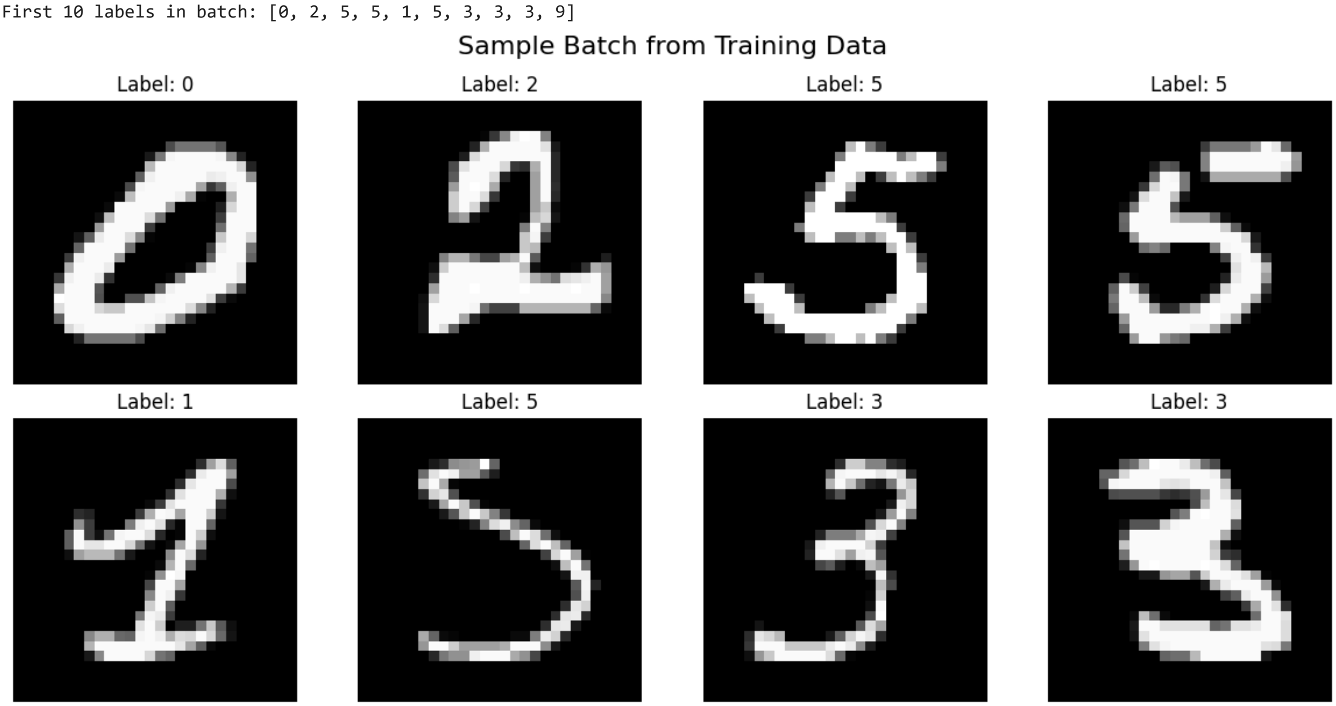
```
# 3. Visualize one image from each of the 10 classes (0-9)
def visualize_mnist_classes(dataset):
    """Find and display one example from each digit class (0-9)"""

    # Dictionary to store first occurrence of each class
    class_examples = {}

    # Find one example of each digit
    for img, label in dataset:
        if label not in class_examples:
            class_examples[label] = img

        # Stop once we have all 10 digits
        if len(class_examples) == 10:
            break

    # Plot the images
    fig, axes = plt.subplots(2, 5, figsize=(12, 6))
    fig.suptitle('One Example from Each MNIST Class', fontsize=16)

    for digit in range(10):
        row = digit // 5
        col = digit % 5

        # Get the image tensor and convert to numpy
        img = class_examples[digit].squeeze()  # Remove channel dimension

        axes[row, col].imshow(img, cmap='gray')
        axes[row, col].set_title(f'Digit: {digit}')
        axes[row, col].axis('off')

    plt.tight_layout()
    plt.show()

# Visualize the classes
visualize_mnist_classes(train_dataset)

# 4. Create DataLoaders
batch_size = 64  # Adjust if you run out of GPU memory (try 32, 16, etc.)

# Training DataLoader (with shuffling)
train_loader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=True,        # Shuffle training data
    num_workers=2,       # Number of subprocesses for data loading
    pin_memory=True      # Faster data transfer to GPU
)

# Testing DataLoader (no shuffling needed)
test_loader = DataLoader(
    test_dataset,
    batch_size=batch_size,
    shuffle=False,       # Don't shuffle test data
    num_workers=2,
    pin_memory=True
)

print(f"\nDataLoader Info:")
print(f"Batch size: {batch_size}")
print(f"Number of training batches: {len(train_loader)}")
print(f"Number of test batches: {len(test_loader)}")

# 5. Verify the data loader works
# Get one batch
images, labels = next(iter(train_loader))
print(f"\nBatch shape: {images.shape}")  # Should be [batch_size, 1, 28, 28]
print(f"Labels shape: {labels.shape}")   # Should be [batch_size]
print(f"First 10 labels in batch: {labels[:10].tolist()}")

# 6. Visualize a batch of images
def show_batch(images, labels, n=8):
    """Display n images from a batch"""
    fig, axes = plt.subplots(2, 4, figsize=(12, 6))
    fig.suptitle('Sample Batch from Training Data', fontsize=16)

    for idx in range(min(n, len(images))):
```

```
        row = idx // 4
        col = idx % 4

        img = images[idx].squeeze()  # Remove channel dimension

        axes[row, col].imshow(img, cmap='gray')
        axes[row, col].set_title(f'Label: {labels[idx].item()}')
        axes[row, col].axis('off')

    plt.tight_layout()
    plt.show()

# Show a batch
show_batch(images, labels)

# Check if GPU is available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"\nUsing device: {device}")
```

```
100%|██████████| 9.91M/9.91M [00:01<00:00, 4.96MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 130kB/s]
100%|██████████| 1.65M/1.65M [00:01<00:00, 1.24MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 14.0MB/s]
Training samples: 60000
Test samples: 10000
```




```
DataLoader Info:
Batch size: 64
Number of training batches: 938
Number of test batches: 157

Batch shape: torch.Size([64, 1, 28, 28])
Labels shape: torch.Size([64])
First 10 labels in batch: [0, 2, 5, 5, 1, 5, 3, 3, 3, 9]
```




```
Using device: cuda
```

```
class MLP(nn.Module):
```

```
    def __init__(self):
        super().__init__()

        # Input: 28x28 = 784 pixels (flattened)
        # Hidden layer 1: 784 → 50
        self.linear1 = nn.Linear(784, 50)

        # Hidden layer 2: 50 → 50
        self.linear2 = nn.Linear(50, 50)

        # Output layer: 50 → 10 (10 classes for digits 0-9)
        self.linear3 = nn.Linear(50, 10)

    def forward(self, x):
        # Flatten the images: [batch_size, 1, 28, 28] → [batch_size, 784]
        x = x.view(x.size(0), -1)

        # First hidden layer with ReLU activation
        x = self.linear1(x)
        x = torch.relu(x)

        # Second hidden layer with ReLU activation
        x = self.linear2(x)
        x = torch.relu(x)

        # Output layer (no activation - will use CrossEntropyLoss)
        x = self.linear3(x)

        return x

# Instantiate the model
model = MLP()

# Move model to GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
print(f"Model is on device: {device}")
print(f"Model parameter device: {next(model.parameters()).device}\n")

# Define loss function (FIXED: renamed from 'loss' to 'criterion')
criterion = nn.CrossEntropyLoss()

# Define optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training parameters
num_epochs = 20

# Initialize storage lists
train_losses = []
test_accuracies = []

# Training loop
for epoch in range(num_epochs):
    # ========== TRAINING PHASE ==========
    model.train()  # Set model to training mode

    epoch_train_loss = 0.0
    num_train_batches = 0

    # First inner loop: iterate over training data
    for batch_idx, (images, labels) in enumerate(train_loader):
        # Move data to GPU
        images = images.to(device)
        labels = labels.to(device)

        # Zero the gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(images)

        # Compute loss (FIXED: now using 'criterion' consistently)
        loss = criterion(outputs, labels)

        # Backward pass (compute gradients)
        loss.backward()
```

```
        # Update parameters
        optimizer.step()

        # Track training loss
        epoch_train_loss += loss.item()
        num_train_batches += 1

    # Average training loss for this epoch
    avg_train_loss = epoch_train_loss / num_train_batches
    train_losses.append(avg_train_loss)

    # ========== TESTING PHASE ==========
    model.eval()  # Set model to evaluation mode

    correct = 0
    total = 0

    # Second inner loop: iterate over test data
    with torch.no_grad():  # No gradient computation during testing
        for images, labels in test_loader:
            # Move data to GPU
            images = images.to(device)
            labels = labels.to(device)

            # Forward pass
            outputs = model(images)

            # Get predictions (class with highest score)
            _, predicted = torch.max(outputs.data, 1)

            # Count correct predictions
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    # Compute accuracy
    accuracy = 100 * correct / total
    test_accuracies.append(accuracy)

    # Print progress
    print(f'Epoch [{epoch+1}/{num_epochs}], '
          f'Train Loss: {avg_train_loss:.4f}, '
          f'Test Accuracy: {accuracy:.2f}%')

print('\nTraining complete!')

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# Plot training loss
ax1.plot(range(1, num_epochs + 1), train_losses, 'b-o', linewidth=2, markersize=6)
ax1.set_xlabel('Epoch', fontsize=12)
ax1.set_ylabel('Training Loss', fontsize=12)
ax1.set_title('Training Loss over Epochs', fontsize=14)
ax1.grid(True, alpha=0.3)

# Plot test accuracy
ax2.plot(range(1, num_epochs + 1), test_accuracies, 'r-o', linewidth=2, markersize=6)
ax2.set_xlabel('Epoch', fontsize=12)
ax2.set_ylabel('Test Accuracy (%)', fontsize=12)
ax2.set_title('Test Accuracy over Epochs', fontsize=14)
ax2.grid(True, alpha=0.3)

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