Deep Learning Writing Sample

In [1]:
```
import torch
print(torch.__version__)
```

```
2.7.1
```

In [2]:
```
import torch
print("PyTorch Version:", torch.__version__)
print("CUDA Available:", torch.cuda.is_available())
print("Device:", torch.device("cuda" if torch.cuda.is_available() else "cpu"
```

```
PyTorch Version: 2.7.1
CUDA Available: False
Device: cpu
```

In [3]:
```
!pip install numpy pandas matplotlib
```

```
Requirement already satisfied: numpy in /opt/anaconda3/lib/python3.12/site-p
ackages (1.26.4)
Requirement already satisfied: pandas in /opt/anaconda3/lib/python3.12/site-
packages (2.2.2)
Requirement already satisfied: matplotlib in /opt/anaconda3/lib/python3.12/s
ite-packages (3.9.2)
Requirement already satisfied: python-dateutil>=2.8.2 in /opt/anaconda3/lib/
python3.12/site-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /opt/anaconda3/lib/python3.1
2/site-packages (from pandas) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in /opt/anaconda3/lib/python3.
12/site-packages (from pandas) (2023.3)
Requirement already satisfied: contourpy>=1.0.1 in /opt/anaconda3/lib/python
3.12/site-packages (from matplotlib) (1.2.0)
Requirement already satisfied: cycler>=0.10 in /opt/anaconda3/lib/python3.1
2/site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in /opt/anaconda3/lib/pytho
n3.12/site-packages (from matplotlib) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in /opt/anaconda3/lib/pytho
n3.12/site-packages (from matplotlib) (1.4.4)
Requirement already satisfied: packaging>=20.0 in /opt/anaconda3/lib/python
3.12/site-packages (from matplotlib) (24.1)
Requirement already satisfied: pillow>=8 in /opt/anaconda3/lib/python3.12/si
te-packages (from matplotlib) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /opt/anaconda3/lib/python
3.12/site-packages (from matplotlib) (3.1.2)
Requirement already satisfied: six>=1.5 in /opt/anaconda3/lib/python3.12/sit
e-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
```

In [4]:
```
# !pip install torch
import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split, TensorDatase

import torchvision
import torchvision.transforms as transforms
```

```
from torchsummary import summary
from sklearn.metrics import f1_score, confusion_matrix
from sklearn.model_selection import train_test_split
```

In [5]:
```
#Assignment Test

num_samples = 1000
input_size = 10

x_data_np = np.random.randn(num_samples, input_size).astype(np.float32)  # F
y_data_np = np.random.randint(0, 2, num_samples).astype(np.int64)  # Binary

# Convert to PyTorch tensors
x_data = torch.tensor(x_data_np, dtype=torch.float32)
y_data = torch.tensor(y_data_np, dtype=torch.long)
```

In [8]:
```
print(x_data_np.shape, x_data.shape)
```

```
(1000, 10) torch.Size([1000, 10])
```

In [9]:
```
#Coding Assignment 1
print("Coding Assignment 1_Shuxin Tang_UNI:st3759")
```

```
Coding Assignment 1_Shuxin Tang_UNI:st3759
```

In [10]:
```
#1.Constructing tensors


import torch
import numpy as np

 #example 1
vector_tensor = torch.tensor([1, 2, 3, 4, 5])
print(f"Vector tensor: {vector_tensor}")
print(f"Shape: {vector_tensor.shape}")
print(f"Dimensions: {vector_tensor.dim()}")
print()
#example 2
float_tensor = torch.tensor([1.5, 2.0, 3.7], dtype=torch.float32)
print(f"Float tensor: {float_tensor}")
print(f"Data type: {float_tensor.dtype}")
print()
#example 3
int_tensor = torch.tensor([1, 2, 3,4,6,7,8] ,dtype=torch.int64)
print(f"Integer tensor: {int_tensor}")
print(f"Data type: {int_tensor.dtype}")
```

```python
print()

##
import torch

print("===== Example 1: Vector Tensor =====")
vector_tensor = torch.tensor([1, 2, 3, 4, 5])
print(f"Vector tensor: {vector_tensor}")
print(f"Shape: {vector_tensor.shape}")
print(f"Dimensions: {vector_tensor.dim()}")
print(f"Size: {vector_tensor.size()}")
print(f"Number of elements: {vector_tensor.numel()}")
print(f"Data type: {vector_tensor.dtype}")
print(f"Device: {vector_tensor.device}")
print(f"Stride: {vector_tensor.stride()}")
print(f"Is contiguous: {vector_tensor.is_contiguous()}")
print()

print("===== Example 2: Float Tensor =====")
float_tensor = torch.tensor([1.5, 2.0, 3.7], dtype=torch.float32)
print(f"Float tensor: {float_tensor}")
print(f"Shape: {float_tensor.shape}")
print(f"Dimensions: {float_tensor.dim()}")
print(f"Size: {float_tensor.size()}")
print(f"Number of elements: {float_tensor.numel()}")
print(f"Data type: {float_tensor.dtype}")
print(f"Device: {float_tensor.device}")
print(f"Min value: {float_tensor.min().item()}")
print(f"Max value: {float_tensor.max().item()}")
print(f"Mean value: {float_tensor.mean().item()}")
print()

print("===== Example 3: Integer Tensor =====")
int_tensor = torch.tensor([1, 2, 3, 4, 6, 7, 8], dtype=torch.int64)
print(f"Integer tensor: {int_tensor}")
print(f"Shape: {int_tensor.shape}")
print(f"Dimensions: {int_tensor.dim()}")
print(f"Size: {int_tensor.size()}")
print(f"Number of elements: {int_tensor.numel()}")
print(f"Data type: {int_tensor.dtype}")
print(f"Device: {int_tensor.device}")
print(f"Min value: {int_tensor.min().item()}")
print(f"Max value: {int_tensor.max().item()}")
print(f"Storage size in bytes: {int_tensor.storage().size() * int_tensor.ele
print()
```

```
Vector tensor: tensor([1, 2, 3, 4, 5])
Shape: torch.Size([5])
Dimensions: 1

Float tensor: tensor([1.5000, 2.0000, 3.7000])
Data type: torch.float32

Integer tensor: tensor([1, 2, 3, 4, 6, 7, 8])
Data type: torch.int64

===== Example 1: Vector Tensor =====
Vector tensor: tensor([1, 2, 3, 4, 5])
Shape: torch.Size([5])
Dimensions: 1
Size: torch.Size([5])
Number of elements: 5
Data type: torch.int64
Device: cpu
Stride: (1,)
Is contiguous: True

===== Example 2: Float Tensor =====
Float tensor: tensor([1.5000, 2.0000, 3.7000])
Shape: torch.Size([3])
Dimensions: 1
Size: torch.Size([3])
Number of elements: 3
Data type: torch.float32
Device: cpu
Min value: 1.5
Max value: 3.700000047683716
Mean value: 2.3999998569488525

===== Example 3: Integer Tensor =====
Integer tensor: tensor([1, 2, 3, 4, 6, 7, 8])
Shape: torch.Size([7])
Dimensions: 1
Size: torch.Size([7])
Number of elements: 7
Data type: torch.int64
Device: cpu
Min value: 1
Max value: 8
Storage size in bytes: 56
```

```python
print("===== Generating Tensors with torch.zeros =====")
zeros_tensor = torch.zeros(3, 4)#3*4 tensors
print(f"zeros(3, 4):\n{zeros_tensor}")
print(f"Shape: {zeros_tensor.shape}")
print(f"Data type: {zeros_tensor.dtype}")
print()

# Create a tensor of zeros with specific data type
zeros_int = torch.zeros(2, 3, dtype=torch.int64)
print(f"zeros(2, 3, dtype=torch.int64):\n{zeros_int}")
print(f"Data type: {zeros_int.dtype}")
print()

# Create a tensor of zeros like another tensor
template = torch.tensor([[1, 2, 3], [4, 5, 6]])
zeros_like = torch.zeros_like(template)
print(f"Template tensor:\n{template}")
print(f"zeros_like(template):\n{zeros_like}")
print()

print("===== Generating Tensors with torch.rand =====")
# Create a tensor with random values from uniform distribution [0, 1)
rand_tensor = torch.rand(3, 4)
print(f"rand(3, 4):\n{rand_tensor}")
print(f"Shape: {rand_tensor.shape}")
print(f"Min value: {rand_tensor.min().item()}")
print(f"Max value: {rand_tensor.max().item()}")
print()

# Create a 4D random tensor
rand_4d = torch.rand(2, 3, 2, 2)
print(f"rand(2, 3, 2, 2) shape: {rand_4d.shape}")
print(f"First 'image' in batch:\n{rand_4d[0]}")
print()

print("===== Generating Tensors with torch.randn =====")
# Create a tensor with random values from normal distribution (mean=0, std=1
randn_tensor = torch.randn(3, 4)#values drawn from a normal (Gaussian) distr
print(f"randn(3, 4):\n{randn_tensor}")
print(f"Shape: {randn_tensor.shape}")
```

```
===== Generating Tensors with torch.zeros =====
zeros(3, 4):
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
Shape: torch.Size([3, 4])
Data type: torch.float32

zeros(2, 3, dtype=torch.int64):
tensor([[0, 0, 0],
        [0, 0, 0]])
Data type: torch.int64

Template tensor:
tensor([[1, 2, 3],
        [4, 5, 6]])
zeros_like(template):
tensor([[0, 0, 0],
        [0, 0, 0]])

===== Generating Tensors with torch.rand =====
rand(3, 4):
tensor([[0.7185, 0.8516, 0.3691, 0.1213],
        [0.3782, 0.4707, 0.1844, 0.5367],
        [0.5235, 0.4620, 0.3459, 0.2705]])
Shape: torch.Size([3, 4])
Min value: 0.12129795551300049
Max value: 0.8516107797622681

rand(2, 3, 2, 2) shape: torch.Size([2, 3, 2, 2])
First 'image' in batch:
tensor([[[0.5233, 0.3188],
         [0.1103, 0.7910]],

        [[0.2674, 0.7958],
         [0.5242, 0.2103]],

        [[0.1214, 0.7934],
         [0.9446, 0.2035]]])

===== Generating Tensors with torch.randn =====
randn(3, 4):
tensor([[-1.6164,  0.7057,  0.2027, -0.2428],
        [-0.1130,  1.2576,  0.0937, -0.3704],
        [-0.4633,  1.6488,  0.0371,  0.7304]])
Shape: torch.Size([3, 4])
```

In [24]:

```python
#3. Tensor manipulations
import torch

# 1. Create tensors
print("Creating tensors:")
# Create a 2x3 tensor
tensor_a = torch.tensor([[1, 2, 3],
                         [4, 5, 6]], dtype=torch.float32)
print("Tensor A (2x3):")
```

```python
print(tensor_a)

# Create a 3x2 tensor
tensor_b = torch.tensor([[7, 8],
                         [9, 10],
                         [11, 12]], dtype=torch.float32)
print("\nTensor B (3x2):")
print(tensor_b)

# Create a 2x3 tensor for addition
tensor_c = torch.tensor([[2, 2, 2],
                         [2, 2, 2]], dtype=torch.float32)
print("\nTensor C (2x3):")
print(tensor_c)

# 2. Transpose operation
print("\n1. Transpose Operation:")
# Transpose tensor_a from 2x3 to 3x2
tensor_a_transposed = torch.transpose(tensor_a, 0, 1)
print("Tensor A transposed (3x2):")
print(tensor_a_transposed)

# Alternative method using .t() for 2D tensors
tensor_a_transposed_alt = tensor_a.t()
print("\nTensor A transposed using .t() method (3x2):")
print(tensor_a_transposed_alt)

# 3. Matrix multiplication
print("\n2. Matrix Multiplication:")
# Multiply tensor_a (2x3) with tensor_b (3x2) to get a 2x2 tensor
matrix_mul_result = torch.matmul(tensor_a, tensor_b)
print("tensor_a (2x3) @ tensor_b (3x2) = result (2x2):")
print(matrix_mul_result)

# Alternative method using the @ operator
matrix_mul_alt = tensor_a @ tensor_b
print("\nUsing @ operator:")
print(matrix_mul_alt)

# 4. Addition
print("\n3. Addition:")
# Add tensor_a and tensor_c (both 2x3)
addition_result = tensor_a + tensor_c
print("tensor_a + tensor_c:")
print(addition_result)

# Using torch.add
addition_alt = torch.add(tensor_a, tensor_c)
print("\nUsing torch.add:")
print(addition_alt)

# 5. View (reshaping)
print("\n4. View Operations:")
# Flatten tensor_a (2x3) to a 1D tensor of size 6
flattened = tensor_a.view(-1)
print("Tensor A flattened to 1D:")
```

```python
print(flattened)

# torch.Tensor.view
reshaped_3x2 = tensor_a.view(3, 2)
print("\nTensor A reshaped to 3x2:")
print(reshaped_3x2)

# torch.Tensor.view
reshaped_1x6 = tensor_a.view(1, 6)
print("\nTensor A reshaped to 1x6:")
print(reshaped_1x6)

# torch.Tensor.view
reshaped_6x1 = tensor_a.view(6, 1)
print("\nTensor A reshaped to 6x1:")
print(reshaped_6x1)

# 6. Additional example: combining operations
print("\n5. Combining Operations:")
# Transpose, then matrix multiply, then reshape
result = torch.matmul(tensor_a, tensor_b).view(-1)
print("Result of matmul(tensor_a, tensor_b).view(-1):")
print(result)
```

```
Creating tensors:
Tensor A (2x3):
tensor([[1., 2., 3.],
        [4., 5., 6.]])

Tensor B (3x2):
tensor([[ 7.,  8.],
        [ 9., 10.],
        [11., 12.]])

Tensor C (2x3):
tensor([[2., 2., 2.],
        [2., 2., 2.]])

1. Transpose Operation:
Tensor A transposed (3x2):
tensor([[1., 4.],
        [2., 5.],
        [3., 6.]])

Tensor A transposed using .t() method (3x2):
tensor([[1., 4.],
        [2., 5.],
        [3., 6.]])

2. Matrix Multiplication:
tensor_a (2x3) @ tensor_b (3x2) = result (2x2):
tensor([[ 58.,  64.],
        [139., 154.]])

Using @ operator:
tensor([[ 58.,  64.],
        [139., 154.]])

3. Addition:
tensor_a + tensor_c:
tensor([[3., 4., 5.],
        [6., 7., 8.]])

Using torch.add:
tensor([[3., 4., 5.],
        [6., 7., 8.]])

4. View Operations:
Tensor A flattened to 1D:
tensor([1., 2., 3., 4., 5., 6.])

Tensor A reshaped to 3x2:
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])

Tensor A reshaped to 1x6:
tensor([[1., 2., 3., 4., 5., 6.]])

Tensor A reshaped to 6x1:
```

```
tensor([[1.],
        [2.],
        [3.],
        [4.],
        [5.],
        [6.]])

5. Combining Operations:
Result of matmul(tensor_a, tensor_b).view(-1):
tensor([ 58.,  64., 139., 154.])
```

In [3]:

```python
import torch

# Create a basic tensor
tensor = torch.tensor([[1, 2, 3],
                       [4, 5, 6]], dtype=torch.float32)
print("Original tensor:")
print(tensor)
print(f"Original dtype: {tensor.dtype}")
print(f"Original device: {tensor.device}")
print("-" * 50)

# 1. Change the tensor's dtype
# Convert to int64
tensor_int = tensor.to(torch.int64)
print("Tensor converted to int64:")
print(tensor_int)
print(f"New dtype: {tensor_int.dtype}")
print("-" * 50)

# Convert to float64 (double precision)
tensor_double = tensor.to(torch.float64)
print("Tensor converted to float64:")
print(tensor_double)
print(f"New dtype: {tensor_double.dtype}")
print("-" * 50)

# Convert to boolean
tensor_bool = tensor.to(torch.bool)
print("Tensor converted to boolean:")
print(tensor_bool)
print(f"New dtype: {tensor_bool.dtype}")
print("-" * 50)

# 2. Move tensor between devices
# Check if Metal (MPS) is available for Mac
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")

if device.type == "mps":
    # Move tensor to GPU (MPS)
    tensor_gpu = tensor.to(device)
    print("Tensor moved to MPS (Mac GPU):")
    print(tensor_gpu)
    print(f"Device: {tensor_gpu.device}")

    # Move tensor back to CPU
```

```
    tensor_cpu = tensor_gpu.to('cpu')
    print("Tensor moved back to CPU:")
    print(tensor_cpu)
    print(f"Device: {tensor_cpu.device}")
else:
    print("MPS (GPU) is not available on this system.")

    # For demonstration (simulate device changes with dtype changes)
    print("Simulating device movement with dtype changes instead:")

    # Change dtype and demonstrate chaining .to() calls
    tensor_complex = tensor.to(dtype=torch.complex64)
    print("Tensor converted to complex64:")
    print(tensor_complex)
    print(f"New dtype: {tensor_complex.dtype}")

    # Chain operations: change device (theoretically) and dtype in one call
    tensor_chain = tensor.to(device='cpu', dtype=torch.int32)
    print("Tensor with chained .to() operations:")
    print(tensor_chain)
    print(f"Device: {tensor_chain.device}, dtype: {tensor_chain.dtype}")
print("-" * 50)

# 3. Additional examples: combining device and dtype changes
if device.type == "mps":
    # Move to GPU and change dtype in one operation
    tensor_gpu_int = tensor.to(device=device, dtype=torch.int32)
    print("Tensor moved to MPS and converted to int32:")
    print(tensor_gpu_int)
    print(f"Device: {tensor_gpu_int.device}, dtype: {tensor_gpu_int.dtype}")
else:
    # Just change dtype for demonstration
    tensor_alt_dtype = tensor.to(dtype=torch.int16)
    print("Tensor converted to int16:")
    print(tensor_alt_dtype)
    print(f"Device: {tensor_alt_dtype.device}, dtype: {tensor_alt_dtype.dtyp
```

```
Original tensor:
tensor([[1., 2., 3.],
        [4., 5., 6.]])
Original dtype: torch.float32
Original device: cpu
--------------------------------------------------
Tensor converted to int64:
tensor([[1, 2, 3],
        [4, 5, 6]])
New dtype: torch.int64
--------------------------------------------------
Tensor converted to float64:
tensor([[1., 2., 3.],
        [4., 5., 6.]], dtype=torch.float64)
New dtype: torch.float64
--------------------------------------------------
Tensor converted to boolean:
tensor([[True, True, True],
        [True, True, True]])
New dtype: torch.bool
--------------------------------------------------
MPS (GPU) is not available on this system.
Simulating device movement with dtype changes instead:
Tensor converted to complex64:
tensor([[1.+0.j, 2.+0.j, 3.+0.j],
        [4.+0.j, 5.+0.j, 6.+0.j]])
New dtype: torch.complex64
Tensor with chained .to() operations:
tensor([[1, 2, 3],
        [4, 5, 6]], dtype=torch.int32)
Device: cpu, dtype: torch.int32
--------------------------------------------------
Tensor converted to int16:
tensor([[1, 2, 3],
        [4, 5, 6]], dtype=torch.int16)
Device: cpu, dtype: torch.int16
```

In [4]:

```
import torch
!pip install --upgrade torch torchvision torchaudio
```

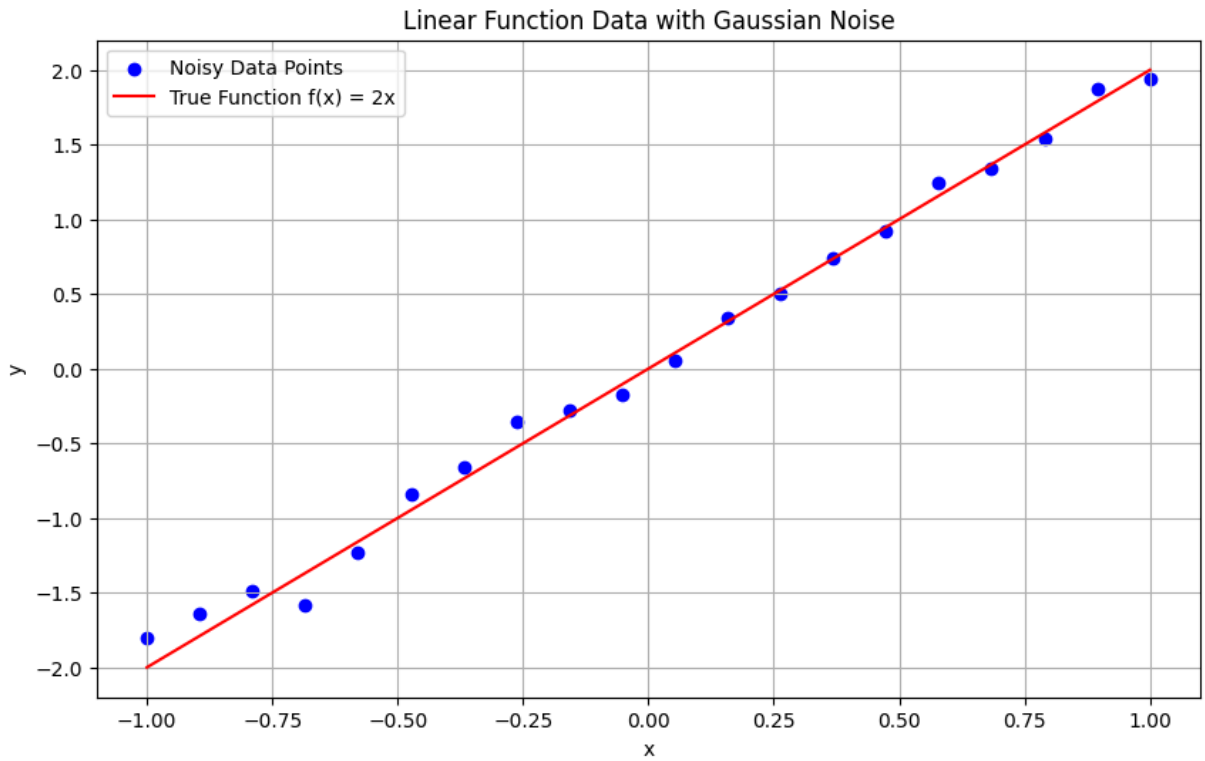
```
Requirement already satisfied: torch in /usr/local/lib/python3.11/dist-packa
ges (2.6.0)
Requirement already satisfied: torchvision in /usr/local/lib/python3.11/dist
-packages (0.21.0)
Requirement already satisfied: torchaudio in /usr/local/lib/python3.11/dist-
packages (2.6.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-pa
ckages (from torch) (3.17.0)
Requirement already satisfied: typing-extensions>=4.10.0 in /usr/local/lib/p
ython3.11/dist-packages (from torch) (4.12.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-pa
ckages (from torch) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-pack
ages (from torch) (3.1.5)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-pack
ages (from torch) (2024.10.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.4.127 in /usr/loca
l/lib/python3.11/dist-packages (from torch) (12.4.127)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.4.127 in /usr/lo
cal/lib/python3.11/dist-packages (from torch) (12.4.127)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.4.127 in /usr/loca
l/lib/python3.11/dist-packages (from torch) (12.4.127)
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /usr/local/li
b/python3.11/dist-packages (from torch) (9.1.0.70)
Requirement already satisfied: nvidia-cublas-cu12==12.4.5.8 in /usr/local/li
b/python3.11/dist-packages (from torch) (12.4.5.8)
Requirement already satisfied: nvidia-cufft-cu12==11.2.1.3 in /usr/local/li
b/python3.11/dist-packages (from torch) (11.2.1.3)
Requirement already satisfied: nvidia-curand-cu12==10.3.5.147 in /usr/local/
lib/python3.11/dist-packages (from torch) (10.3.5.147)
Requirement already satisfied: nvidia-cusolver-cu12==11.6.1.9 in /usr/local/
lib/python3.11/dist-packages (from torch) (11.6.1.9)
Requirement already satisfied: nvidia-cusparse-cu12==12.3.1.170 in /usr/loca
l/lib/python3.11/dist-packages (from torch) (12.3.1.170)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/local/l
ib/python3.11/dist-packages (from torch) (0.6.2)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/py
thon3.11/dist-packages (from torch) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/
python3.11/dist-packages (from torch) (12.4.127)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.4.127 in /usr/loca
l/lib/python3.11/dist-packages (from torch) (12.4.127)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/di
st-packages (from torch) (3.2.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/di
st-packages (from torch) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.
11/dist-packages (from sympy==1.13.1->torch) (1.3.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packa
ges (from torchvision) (1.26.4)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/pytho
n3.11/dist-packages (from torchvision) (11.1.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/
dist-packages (from jinja2->torch) (3.0.2)
```

In [5]:
```
import torch
print(torch.__version__)  # Should be 1.12.0 or newer for MPS support
```

```
2.6.0+cu124
```

In [ ]:
```
#5. Linear regression
```

In [6]:
```
import torch
import matplotlib.pyplot as plt
###
# Set random seed for reproducibility
torch.manual_seed(42)

# Sample 20 training points uniformly in [-1, 1]
x = torch.linspace(-1, 1, 20).unsqueeze(1)

# Define the function f(x) = 2x
def f(x):
    return 2 * x

# Generate labels
y = f(x)

# Add small Gaussian noise
noise = torch.randn_like(y) * 0.1
y_noisy = y + noise

# Plot the data
plt.figure(figsize=(10, 6))
plt.scatter(x.numpy(), y_noisy.numpy(), color='blue', label='Noisy Data Poir
plt.plot(x.numpy(), y.numpy(), color='red', label='True Function f(x) = 2x')
plt.title('Linear Function Data with Gaussian Noise')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()

# Print out the data for verification
print("X values:")
print(x.flatten())
print("\nTrue Y values:")
print(y.flatten())
print("\nNoisy Y values:")
print(y_noisy.flatten())
```




```
X values:
tensor([-1.0000, -0.8947, -0.7895, -0.6842, -0.5789, -0.4737, -0.3684, -0.26
32,
        -0.1579, -0.0526,  0.0526,  0.1579,  0.2632,  0.3684,  0.4737,  0.57
89,
         0.6842,  0.7895,  0.8947,  1.0000])

True Y values:
tensor([-2.0000, -1.7895, -1.5789, -1.3684, -1.1579, -0.9474, -0.7368, -0.52
63,
        -0.3158, -0.1053,  0.1053,  0.3158,  0.5263,  0.7368,  0.9474,  1.15
79,
         1.3684,  1.5789,  1.7895,  2.0000])

Noisy Y values:
tensor([-1.8073, -1.6407, -1.4889, -1.5790, -1.2337, -0.8395, -0.6568, -0.35
83,
        -0.2802, -0.1739,  0.0559,  0.3399,  0.5032,  0.7410,  0.9222,  1.24
39,
         1.3374,  1.5394,  1.8698,  1.9378])
```

In [14]:

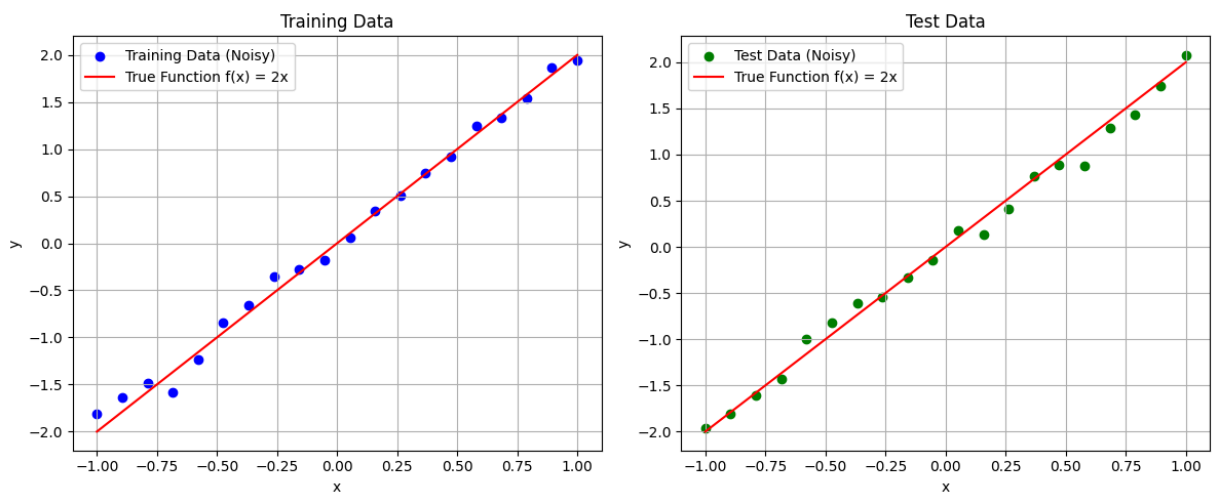
```python
#5 Linear regression Test Data
import torch
import matplotlib.pyplot as plt

# Set random seed for reproducibility
torch.manual_seed(42)

# Generate Training Data
def generate_data(num_points=20, noise_std=0.1):
    # Sample points uniformly in [-1, 1]
    x = torch.linspace(-1, 1, num_points).unsqueeze(1)
```

```
    # Define the function f(x) = 2x
    def f(x):
        return 2 * x

    # Generate labels
    y = f(x)

    # Add small Gaussian noise
    noise = torch.randn_like(y) * noise_std
    y_noisy = y + noise

    return x, y, y_noisy

# Generate Training Data
x_train, y_train, y_train_noisy = generate_data(num_points=20, noise_std=0.1

# Generate Test Data (with different random seed)
torch.manual_seed(123)  # Different seed for test data
x_test, y_test, y_test_noisy = generate_data(num_points=20, noise_std=0.1)

# Plot Training Data
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.scatter(x_train.numpy(), y_train_noisy.numpy(), color='blue', label='Tra
plt.plot(x_train.numpy(), y_train.numpy(), color='red', label='True Functior
plt.title('Training Data')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)

# Plot Test Data
plt.subplot(1, 2, 2)
plt.scatter(x_test.numpy(), y_test_noisy.numpy(), color='green', label='Test
plt.plot(x_test.numpy(), y_test.numpy(), color='red', label='True Function f
plt.title('Test Data')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

# Print out the data for verification
print("Training Data:")
print("X values:", x_train.flatten())
print("True Y values:", y_train.flatten())
print("Noisy Y values:", y_train_noisy.flatten())

print("\nTest Data:")
print("X values:", x_test.flatten())
print("True Y values:", y_test.flatten())
print("Noisy Y values:", y_test_noisy.flatten())

##
```

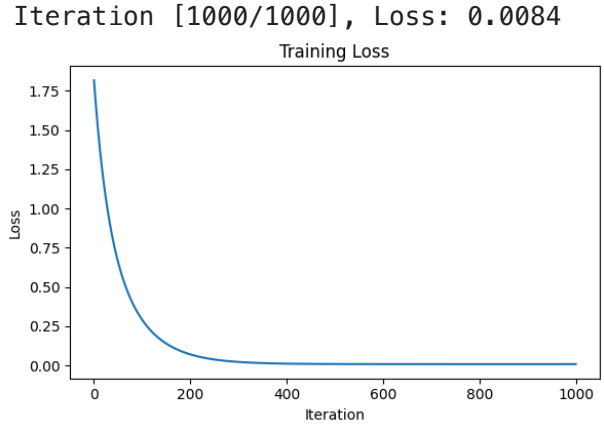
```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# 1. Use torch.nn.Module
class LinearRegressionModel(nn.Module):
    # 2. Use nn.Linear
    def __init__(self, input_dim=1, output_dim=1):
        """
        Constructor using nn.Module

        Args:
            input_dim (int): Number of input features
            output_dim (int): Number of output features
        """
        super().__init__()
        # Create linear layer
        self.linear = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        """
        Forward pass method

        Args:
            x (torch.Tensor): Input tensor

        Returns:
            torch.Tensor: Output after linear transformation
        """
        return self.linear(x)

def generate_data(num_points=20, noise_std=0.1, seed=None):
    """
    Generate synthetic data

    Args:
        num_points (int): Number of data points
        noise_std (float): Standard deviation of noise
        seed (int, optional): Random seed

    Returns:
        tuple: (x, y, y_noisy) tensors
    """
    # Set random seed if provided
    if seed is not None:
        torch.manual_seed(seed)

    # Sample x uniformly in [-1, 1]
    x = torch.linspace(-1, 1, num_points).unsqueeze(1)

    # Define true function f(x) = 2x
    def f(x):
        return 2 * x

    # Generate true labels
```

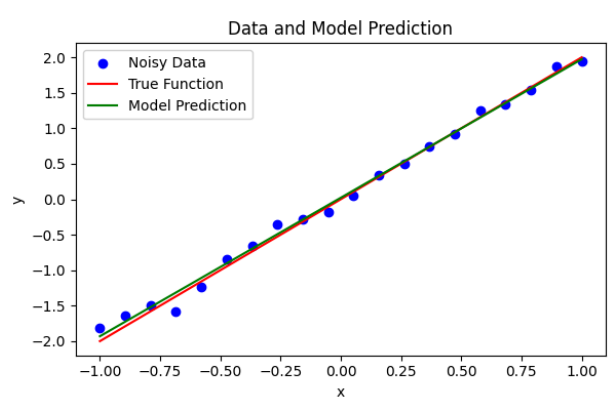
```
    y = f(x)

    # Add Gaussian noise
    noise = torch.randn_like(y) * noise_std
    y_noisy = y + noise

    return x, y, y_noisy

def main():
    # Generate training data
    x_train, y_train, y_train_noisy = generate_data(num_points=20, noise_stc

    # Create model instance
    model = LinearRegressionModel(input_dim=1, output_dim=1)

    # 3. Use nn.MSELoss
    criterion = nn.MSELoss(reduction='mean')

    # 4. Use optim.SGD
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    # Training parameters
    num_iterations = 1000

    # Store loss values
    losses = []

    # Training loop
    for iteration in range(num_iterations):
        # 5. Zero gradients
        optimizer.zero_grad()

        # 6. Forward pass
        y_pred = model(x_train)

        # 7. Compute loss
        loss = criterion(y_pred, y_train_noisy)

        # 8. Compute gradients
        loss.backward()

        # 9. Take descent step
        optimizer.step()

        # 10. Measure and store loss
        losses.append(loss.item())

        # Print loss every 100 iterations
        if (iteration + 1) % 100 == 0:
            print(f'Iteration [{iteration+1}/{num_iterations}], Loss: {loss.

    # Visualize training process
    plt.figure(figsize=(12, 4))

    # Loss plot
    plt.subplot(1, 2, 1)
```

```
    plt.plot(losses)
    plt.title('Training Loss')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')

    # Data and prediction plot
    plt.subplot(1, 2, 2)
    plt.scatter(x_train.numpy(), y_train_noisy.numpy(), color='blue', label=

    # True function
    plt.plot(x_train.numpy(), y_train.numpy(), color='red', label='True Func

    # Model prediction
    with torch.no_grad():
        y_pred = model(x_train)
    plt.plot(x_train.numpy(), y_pred.numpy(), color='green', label='Model Pr

    plt.title('Data and Model Prediction')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()

    plt.tight_layout()
    plt.show()

    # Print final model parameters
    print("\nFinal Model Parameters:")
    for name, param in model.named_parameters():
        print(f"{name}: {param.data}")

if __name__ == "__main__":
    main()
```



```
Training Data:
X values: tensor([-1.0000, -0.8947, -0.7895, -0.6842, -0.5789, -0.4737, -0.3
684, -0.2632,
        -0.1579, -0.0526,  0.0526,  0.1579,  0.2632,  0.3684,  0.4737,  0.57
89,
         0.6842,  0.7895,  0.8947,  1.0000])
True Y values: tensor([-2.0000, -1.7895, -1.5789, -1.3684, -1.1579, -0.9474,
-0.7368, -0.5263,
        -0.3158, -0.1053,  0.1053,  0.3158,  0.5263,  0.7368,  0.9474,  1.15
79,
         1.3684,  1.5789,  1.7895,  2.0000])
Noisy Y values: tensor([-1.8073, -1.6407, -1.4889, -1.5790, -1.2337, -0.839
5, -0.6568, -0.3583,
        -0.2802, -0.1739,  0.0559,  0.3399,  0.5032,  0.7410,  0.9222,  1.24
39,
         1.3374,  1.5394,  1.8698,  1.9378])

Test Data:
X values: tensor([-1.0000, -0.8947, -0.7895, -0.6842, -0.5789, -0.4737, -0.3
684, -0.2632,
        -0.1579, -0.0526,  0.0526,  0.1579,  0.2632,  0.3684,  0.4737,  0.57
89,
         0.6842,  0.7895,  0.8947,  1.0000])
True Y values: tensor([-2.0000, -1.7895, -1.5789, -1.3684, -1.1579, -0.9474,
-0.7368, -0.5263,
        -0.3158, -0.1053,  0.1053,  0.3158,  0.5263,  0.7368,  0.9474,  1.15
79,
         1.3684,  1.5789,  1.7895,  2.0000])
Noisy Y values: tensor([-1.9663, -1.8073, -1.6093, -1.4272, -0.9998, -0.817
3, -0.6093, -0.5464,
        -0.3318, -0.1454,  0.1748,  0.1352,  0.4104,  0.7694,  0.8842,  0.87
39,
         1.2899,  1.4380,  1.7487,  2.0795])
Iteration [100/1000], Loss: 0.2950
Iteration [200/1000], Loss: 0.0713
Iteration [300/1000], Loss: 0.0227
Iteration [400/1000], Loss: 0.0117
Iteration [500/1000], Loss: 0.0092
Iteration [600/1000], Loss: 0.0086
Iteration [700/1000], Loss: 0.0085
Iteration [800/1000], Loss: 0.0084
Iteration [900/1000], Loss: 0.0084
Iteration [1000/1000], Loss: 0.0084
```





```
Final Model Parameters:
linear.weight: tensor([[1.9522]])
linear.bias: tensor([0.0216])
```

In [15]:

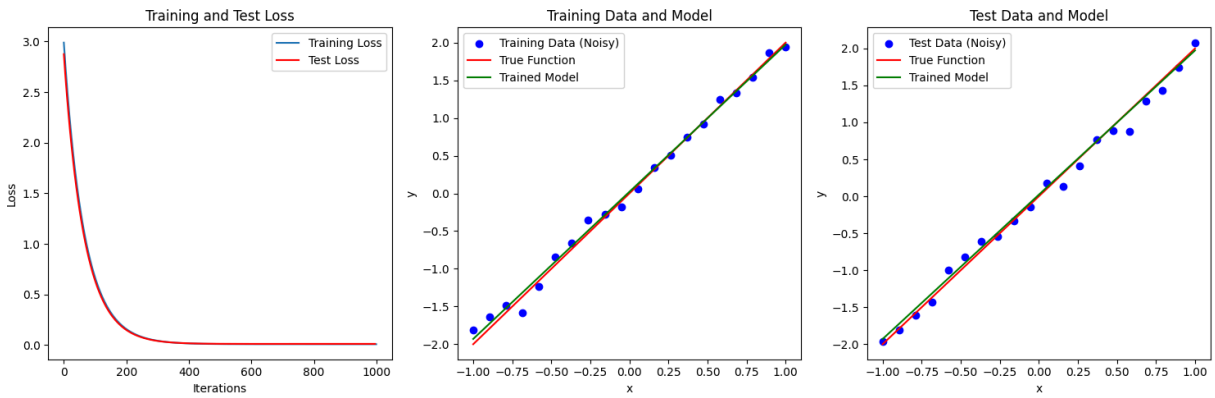
```python
#5
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

class LinearRegressionModel(nn.Module):
    def __init__(self, input_dim=1, output_dim=1):
        super().__init__()
        self.linear = nn.Linear(input_dim, output_dim)

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

def generate_data(num_points=20, noise_std=0.1, seed=None):
    """
    Generate synthetic data for linear regression

    Args:
        num_points (int): Number of data points
        noise_std (float): Standard deviation of noise
        seed (int, optional): Random seed

    Returns:
        tuple: (x, y, y_noisy) tensors
    """
    if seed is not None:
        torch.manual_seed(seed)

    # Sample x uniformly in [-1, 1]
    x = torch.linspace(-1, 1, num_points).unsqueeze(1)

    # True function f(x) = 2x
    def f(x):
        return 2 * x

    # Generate true labels
    y = f(x)

    # Add Gaussian noise
    noise = torch.randn_like(y) * noise_std
    y_noisy = y + noise

    return x, y, y_noisy

def analytical_solution(x, y):
    """
    Compute the analytical solution for linear regression

    Args:
        x (torch.Tensor): Input features
```

```
        y (torch.Tensor): Target values

    Returns:
        tuple: (weight, bias)
    """
    # Convert to numpy for computation
    x_np = x.numpy()
    y_np = y.numpy()

    # Add a column of ones for the intercept term
    X = np.column_stack([x_np, np.ones_like(x_np)])

    # Compute the analytical solution using least squares
    # (X^T X)^-1 X^T y
    weights = np.linalg.inv(X.T @ X) @ X.T @ y_np

    return weights[0][0], weights[1][0]

def main():
    # Generate training and test data
    x_train, y_train, y_train_noisy = generate_data(num_points=20, noise_std
    x_test, y_test, y_test_noisy = generate_data(num_points=20, noise_std=0.

    # Initialize model, loss function, and optimizer
    model = LinearRegressionModel()
    criterion = nn.MSELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    # Training parameters
    num_iterations = 1000

    # Lists to store losses
    training_losses = []
    test_losses = []

    # Training loop
    for iteration in range(num_iterations):
        # Zero the gradients
        optimizer.zero_grad()

        # Forward pass on training data
        y_pred = model(x_train)

        # Compute training loss
        loss = criterion(y_pred, y_train_noisy)

        # Backward pass to compute gradients
        loss.backward()

        # Update model parameters
        optimizer.step()

        # Store training loss
        training_losses.append(loss.item())

        # Compute and store test loss every 5 iterations
```

```
        if iteration % 5 == 0:
            with torch.no_grad():
                test_pred = model(x_test)
                test_loss = criterion(test_pred, y_test_noisy)
                test_losses.append(test_loss.item())

    # Compute analytical solution
    analytical_weight, analytical_bias = analytical_solution(x_train, y_trai

    # Print trained model parameters
    print("\nTrained Model Parameters:")
    for name, param in model.named_parameters():
        print(f"{name}: {param.data}")

    print("\nAnalytical Solution:")
    print(f"Weight: {analytical_weight}")
    print(f"Bias: {analytical_bias}")

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

    # Training and Test Loss Plot
    plt.subplot(1, 3, 1)
    plt.plot(range(num_iterations), training_losses, label='Training Loss')
    plt.plot(range(0, num_iterations, 5), test_losses, label='Test Loss', co
    plt.title('Training and Test Loss')
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.legend()

    # Training Data and Model Prediction
    plt.subplot(1, 3, 2)
    plt.scatter(x_train.numpy(), y_train_noisy.numpy(), color='blue', label=
    plt.plot(x_train.numpy(), y_train.numpy(), color='red', label='True Func

    # Predict with trained model
    with torch.no_grad():
        y_pred = model(x_train)
    plt.plot(x_train.numpy(), y_pred.numpy(), color='green', label='Trained
    plt.title('Training Data and Model')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()

    # Test Data and Model Prediction
    plt.subplot(1, 3, 3)
    plt.scatter(x_test.numpy(), y_test_noisy.numpy(), color='blue', label='T
    plt.plot(x_test.numpy(), y_test.numpy(), color='red', label='True Functi

    # Predict with trained model
    with torch.no_grad():
        y_pred_test = model(x_test)
    plt.plot(x_test.numpy(), y_pred_test.numpy(), color='green', label='Trai
    plt.title('Test Data and Model')
    plt.xlabel('x')
    plt.ylabel('y')
```

```
    plt.legend()

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    main()
```

```
Trained Model Parameters:
linear.weight: tensor([[1.9516]])
linear.bias: tensor([0.0216])

Analytical Solution:
Weight: 1.953325629234314
Bias: 0.02161584049463272
```



In [ ]:

```
#6
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

# 1. Create Transform
transform = transforms.Compose([
    transforms.ToTensor()  # Converts image to tensor and scales to [0, 1]
])

# 2. Instantiate MNIST Datasets
def load_and_visualize_mnist():
    # Load training and testing datasets
    train_dataset = torchvision.datasets.MNIST(
        root='./data',
        train=True,
        download=True,
        transform=transform
    )

    test_dataset = torchvision.datasets.MNIST(
        root='./data',
        train=False,
        download=True,
```

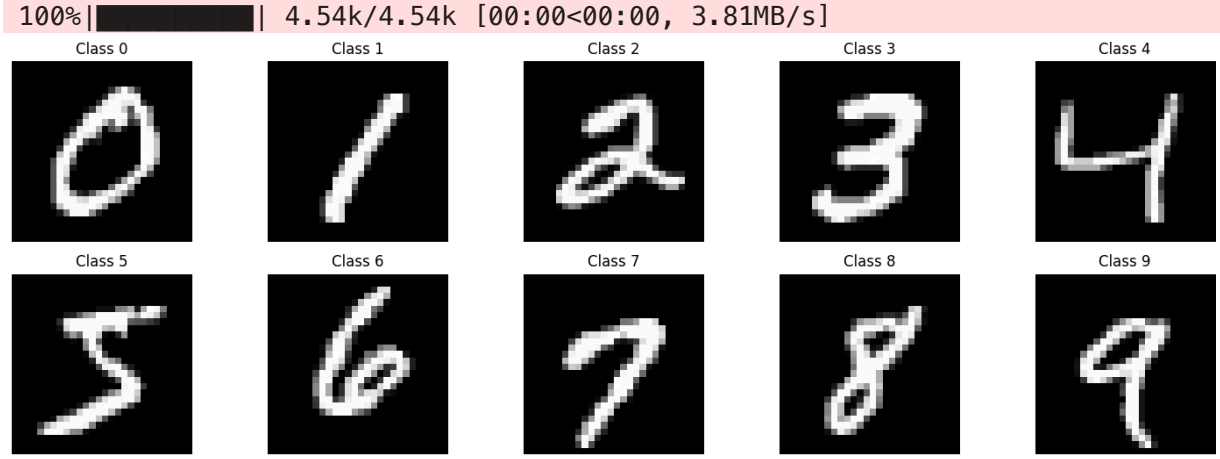
```
        transform=transform
    )

    # Visualize images from each class
    plt.figure(figsize=(15, 5))
    class_names = list(range(10))  # MNIST classes are 0-9

    # Keep track of images per class
    class_counts = {i: 0 for i in range(10)}

    for i in range(len(train_dataset)):
        image, label = train_dataset[i]

        # Only plot one image per class
        if class_counts[label] == 0:
            plt.subplot(2, 5, label + 1)
            plt.imshow(image.squeeze().numpy(), cmap='gray')
            plt.title(f'Class {label}')
            plt.axis('off')

            class_counts[label] += 1

        # Stop when we've found one image for each class
        if all(count > 0 for count in class_counts.values()):
            break

    plt.tight_layout()
    plt.show()

    return train_dataset, test_dataset

# 3. Create Data Loaders
def create_data_loaders(train_dataset, test_dataset, batch_size=64):
    # Determine device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")

    # Create data loaders
    train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=batch_size,
        shuffle=True,
        num_workers=2
    )

    test_loader = torch.utils.data.DataLoader(
        test_dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=2
    )

    return train_loader, test_loader

# 4. Multi-Layer Perceptron Model
class MNISTClassificationModel(nn.Module):
```

```python
    def __init__(self, input_size=784, hidden_size=50, num_classes=10):
        """
        Multi-Layer Perceptron for MNIST Classification

        Args:
            input_size (int): Number of input features (28*28 = 784)
            hidden_size (int): Number of neurons in hidden layers
            num_classes (int): Number of output classes
        """
        super().__init__()

        # Network architecture
        self.flatten = nn.Flatten()  # Flatten 28x28 images to 784-dim vecto

        # Two hidden layers with ReLU activations
        self.network = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, num_classes)
        )

    def forward(self, x):
        # Flatten the input
        x = self.flatten(x)

        # Pass through the network
        return self.network(x)

# 5 & 6. Training Loop
def train_and_evaluate(model, train_loader, test_loader, num_epochs=20):
    # Move model to GPU if available
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = model.to(device)

    # Loss function and Optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

    # Tracking lists
    train_losses = []
    test_accuracies = []

    # Training loop
    for epoch in range(num_epochs):
        # Training mode
        model.train()
        total_train_loss = 0.0

        # Train on batches
        for batch_idx, (data, target) in enumerate(train_loader):
            # Move to GPU
            data, target = data.to(device), target.to(device)

            # Zero gradients
```

```
            optimizer.zero_grad()

            # Forward pass
            output = model(data)

            # Compute loss
            loss = criterion(output, target)

            # Backward pass
            loss.backward()

            # Optimizer step
            optimizer.step()

            # Track training loss
            total_train_loss += loss.item()

        # Average training loss for epoch
        avg_train_loss = total_train_loss / len(train_loader)
        train_losses.append(avg_train_loss)

        # Evaluation mode
        model.eval()
        correct = 0
        total = 0

        # Test on batches
        with torch.no_grad():
            for data, target in test_loader:
                # Move to GPU
                data, target = data.to(device), target.to(device)

                # Forward pass
                output = model(data)

                # Predictions
                _, predicted = torch.max(output.data, 1)

                # Total predictions
                total += target.size(0)

                # Correct predictions
                correct += (predicted == target).sum().item()

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

        # Print epoch summary
        print(f'Epoch [{epoch+1}/{num_epochs}]')
        print(f'Training Loss: {avg_train_loss:.4f}')
        print(f'Test Accuracy: {test_accuracy:.2f}%')

    # Plotting
    plt.figure(figsize=(12, 5))
```

```python
    # Training Loss Plot
    plt.subplot(1, 2, 1)
    plt.plot(train_losses)
    plt.title('Training Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')

    # Test Accuracy Plot
    plt.subplot(1, 2, 2)
    plt.plot(test_accuracies)
    plt.title('Test Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy (%)')

    plt.tight_layout()
    plt.show()

    return train_losses, test_accuracies

def main():
    # Load and visualize MNIST datasets
    train_dataset, test_dataset = load_and_visualize_mnist()

    # Create data loaders
    train_loader, test_loader = create_data_loaders(train_dataset, test_data

    # Create model
    model = MNISTClassificationModel()

    # Train and evaluate
    train_and_evaluate(model, train_loader, test_loader)

if __name__ == "__main__":
    main()
```

```
100%|██████████| 9.91M/9.91M [00:00<00:00, 14.2MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 470kB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 3.46MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 3.81MB/s]
```




```
Using device: cpu
Epoch [1/20]
Training Loss: 0.5398
Test Accuracy: 93.13%
```