Part 1 (25 points): Simple test cases.

a. XOR (https://en.wikipedia.org/wiki/Exclusive_or)

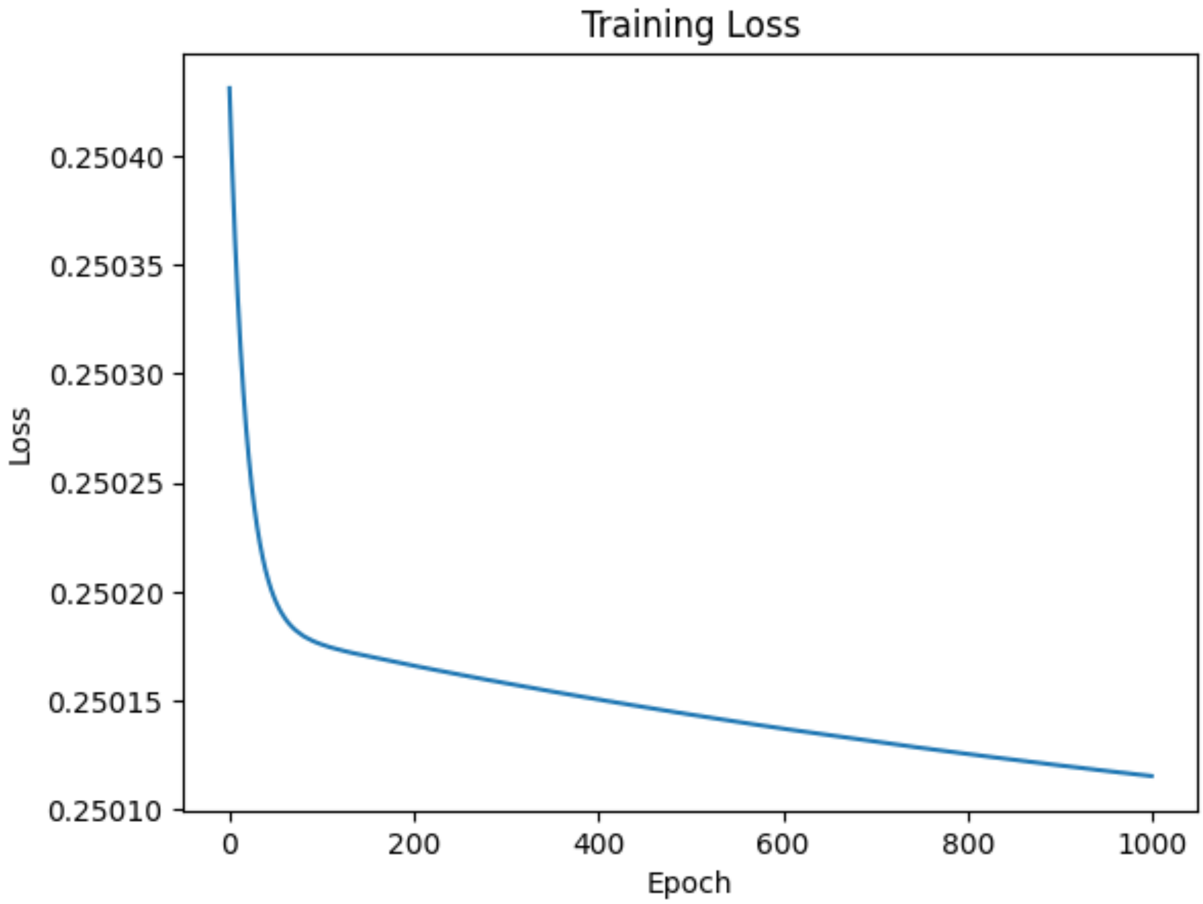
Two input variables, one output variable

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression
import torchvision
import torchvision.transforms as transforms
```

In [2]:

```python
# Define XOR dataset
X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)
y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)

# Define neural network model
class XORModel(nn.Module):
    def __init__(self):
        super(XORModel, self).__init__()
        self.fc1 = nn.Linear(2, 4)  # Input layer to hidden layer
        self.fc2 = nn.Linear(4, 1)  # Hidden layer to output layer
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.fc1(x))
        x = self.sigmoid(self.fc2(x))
        return x

# Instantiate the model
model = XORModel()

# Define loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

# Training loop
epochs = 1000
losses = []
for epoch in range(epochs):
    optimizer.zero_grad()  # Zero the gradients
    outputs = model(X)  # Forward pass
    loss = criterion(outputs, y)  # Calculate the loss
    loss.backward()  # Backward pass
    optimizer.step()  # Optimize weights

    # Print loss every 100 epochs
    if (epoch+1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
    losses.append(loss.item())

# Plot loss vs. epoch
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.show()

# Test the model
with torch.no_grad():
    predicted = model(X)
    predicted = predicted.round()  # Round predictions to 0 or 1
    print(f'Predicted: {predicted.squeeze().tolist()}')
```

```
Epoch [100/1000], Loss: 0.2502
Epoch [200/1000], Loss: 0.2502
Epoch [300/1000], Loss: 0.2502
Epoch [400/1000], Loss: 0.2502
Epoch [500/1000], Loss: 0.2501
Epoch [600/1000], Loss: 0.2501
Epoch [700/1000], Loss: 0.2501
Epoch [800/1000], Loss: 0.2501
Epoch [900/1000], Loss: 0.2501
Epoch [1000/1000], Loss: 0.2501
```




```
Predicted: [0.0, 1.0, 0.0, 1.0]
```

b. Sine with additive white gaussian noise (make it a small standard deviation so you can still see a sine when plotted)

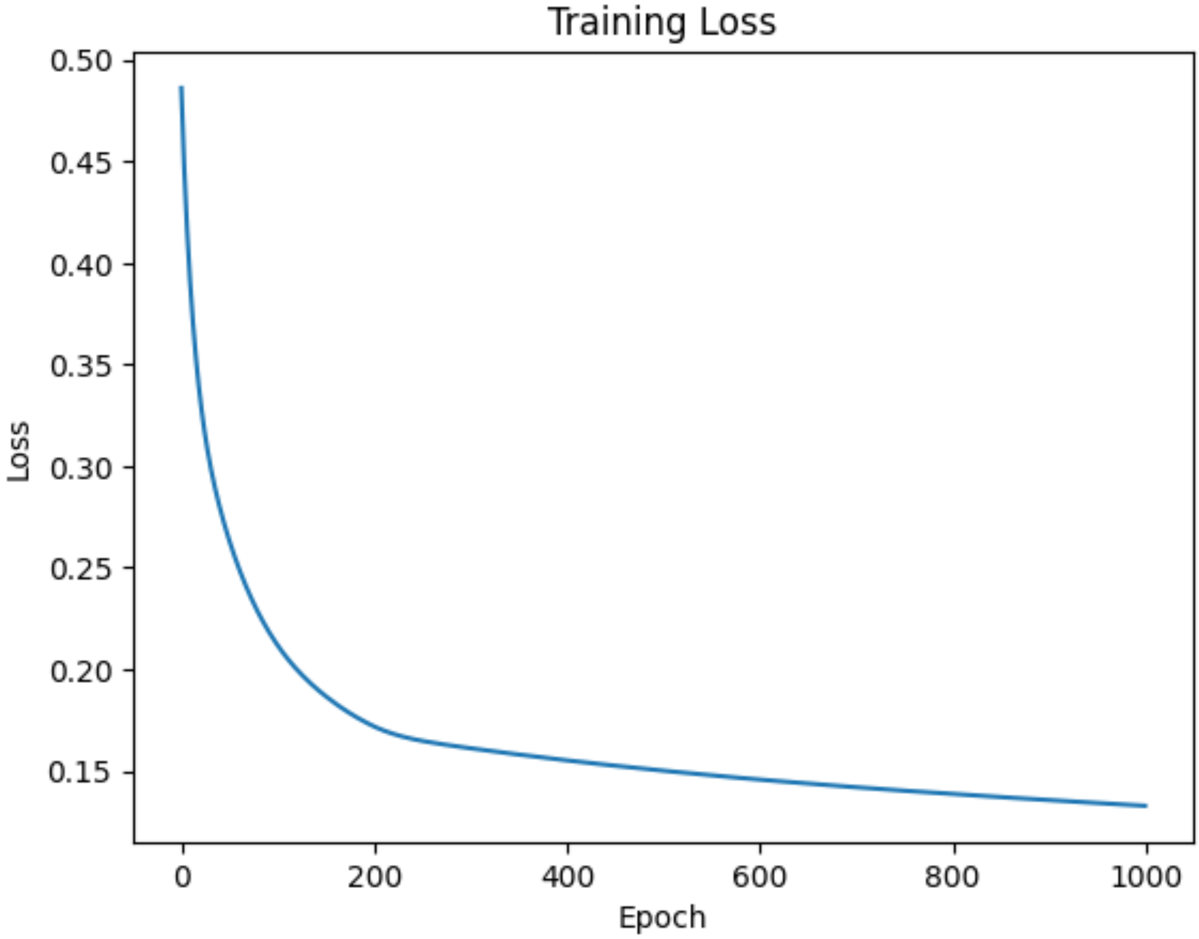
One input variable, one output variable

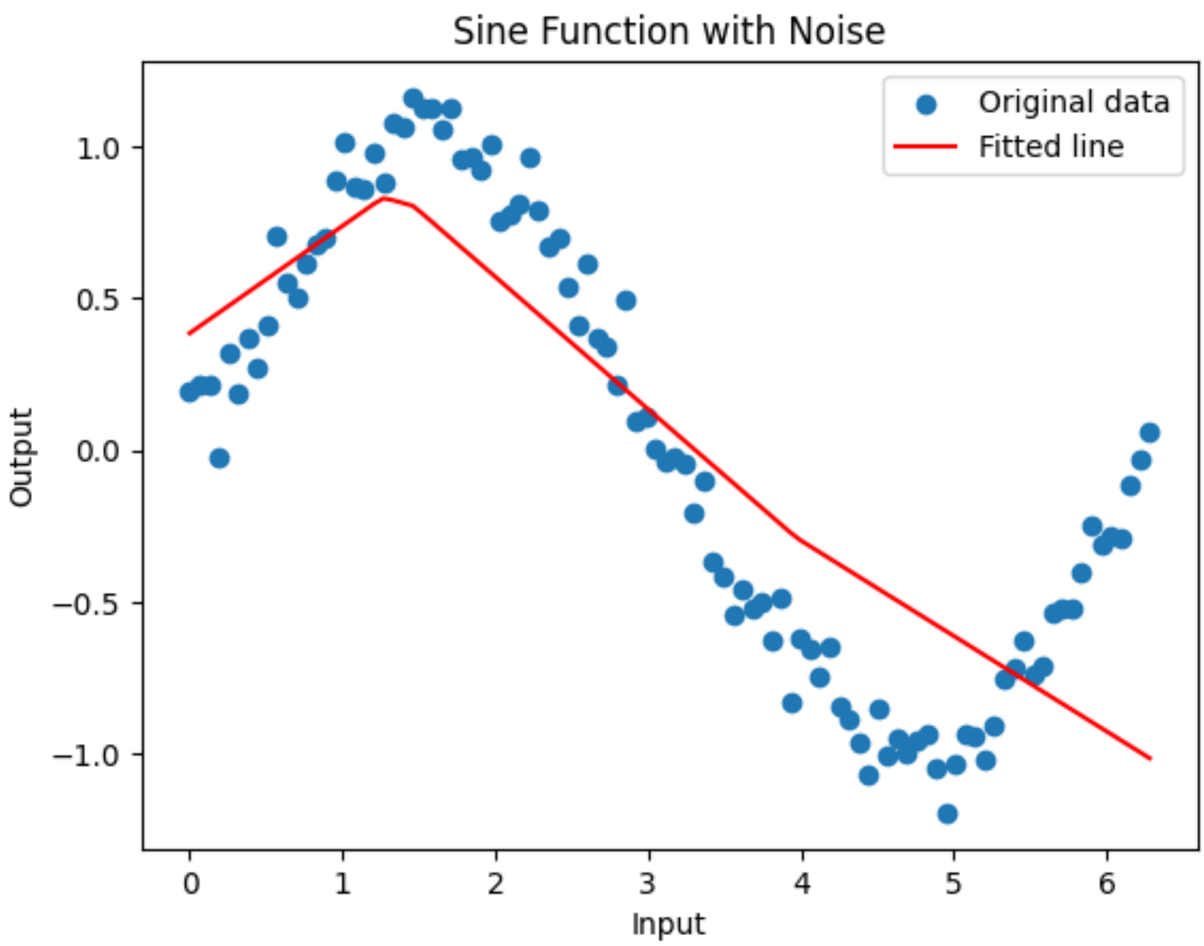
In [3]:
```
# Generate sine with additive white Gaussian noise
np.random.seed(42)
torch.manual_seed(42)

# Number of data points
num_points = 100
```

```
# Generate random input values
X = torch.linspace(0, 2*np.pi, num_points).reshape(-1, 1)

# Generate corresponding output values with sine function and additive no
y = torch.sin(X) + torch.randn_like(X) * 0.1  # Adding Gaussian noise wit

# Define neural network model
class SineModel(nn.Module):
    def __init__(self):
        super(SineModel, self).__init__()
        self.fc1 = nn.Linear(1, 10)  # Input layer to hidden layer
        self.fc2 = nn.Linear(10, 1)   # Hidden layer to output layer
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Instantiate the model
model = SineModel()

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

# Training loop
epochs = 1000
losses = []
for epoch in range(epochs):
    optimizer.zero_grad()  # Zero the gradients
    outputs = model(X)     # Forward pass
    loss = criterion(outputs, y)  # Calculate the loss
    loss.backward()       # Backward pass
    optimizer.step()      # Optimize weights

    # Print loss every 100 epochs
    if (epoch+1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
    losses.append(loss.item())

# Plot loss vs. epoch
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.show()

# Test the model
with torch.no_grad():
    predicted = model(X)
    plt.scatter(X, y, label='Original data')
    plt.plot(X, predicted, 'r-', label='Fitted line')
    plt.xlabel('Input')
    plt.ylabel('Output')
    plt.title('Sine Function with Noise')
    plt.legend()
```

```
    plt.show()
```

```
Epoch [100/1000], Loss: 0.2124
Epoch [200/1000], Loss: 0.1720
Epoch [300/1000], Loss: 0.1612
Epoch [400/1000], Loss: 0.1552
Epoch [500/1000], Loss: 0.1501
Epoch [600/1000], Loss: 0.1457
Epoch [700/1000], Loss: 0.1419
Epoch [800/1000], Loss: 0.1386
Epoch [900/1000], Loss: 0.1356
Epoch [1000/1000], Loss: 0.1328
```





If we increase nuerons we can get good output but I am just leaving as long as it is working

Construct the following two fully connected neural networks with any activation function and train them using SGD using Binary Cross Entropy (BCE) Loss for the XOR and MSE loss for the sine(because it's regression). Report out hyperparameters used along with any adjustments required to make them work. These are meant to be simple cases without a lot of effort spent tuning them. Do a simple test/train split (ie. for XOR maybe train on 3 and test on 1) (10 points each)

1. Single hidden layer neural network with a hidden layer dimension between 2 and 50.

2. Two hidden layer neural network with hidden layer dimensions between 2 and 50.

In [4]:
```
# Define XOR dataset
```

```
X_xor = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float3
y_xor = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)

# Define Sine dataset
X_sine = torch.linspace(0, 2*np.pi, 100).view(-1, 1)
y_sine = torch.sin(X_sine)

# Define Single Hidden Layer Neural Network
class SHLNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(SHLNN, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.fc1(x))
        x = self.sigmoid(self.fc2(x))
        return x

# Hyperparameters
input_dim = 2  # for XOR
hidden_dim = 10
output_dim = 1
lr = 0.1
epochs = 1000

# Training XOR
model_xor = SHLNN(input_dim, hidden_dim, output_dim)
criterion_xor = nn.BCELoss()
optimizer_xor = optim.SGD(model_xor.parameters(), lr=lr)

for epoch in range(epochs):
    optimizer_xor.zero_grad()
    output = model_xor(X_xor)
    loss = criterion_xor(output, y_xor)
    loss.backward()
    optimizer_xor.step()

# Training Sine Regression
input_dim = 1  # for Sine
hidden_dim = 10
output_dim = 1

model_sine = SHLNN(input_dim, hidden_dim, output_dim)
criterion_sine = nn.MSELoss()
optimizer_sine = optim.SGD(model_sine.parameters(), lr=lr)

for epoch in range(epochs):
    optimizer_sine.zero_grad()
    output = model_sine(X_sine)
    loss = criterion_sine(output, y_sine)
    loss.backward()
    optimizer_sine.step()

# Testing XOR
test_output_xor = model_xor(X_xor)
```

```python
print("XOR Predictions:", test_output_xor.round())

# Testing Sine Regression
test_output_sine = model_sine(X_sine)
```

```
XOR Predictions: tensor([[0.],
        [1.],
        [0.],
        [1.]], grad_fn=<RoundBackward0>)
```

In [5]:
```python
# Define Two Hidden Layer Neural Network
class THLNN(nn.Module):
    def __init__(self, input_dim, hidden_dim1, hidden_dim2, output_dim):
        super(THLNN, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim1)
        self.fc2 = nn.Linear(hidden_dim1, hidden_dim2)
        self.fc3 = nn.Linear(hidden_dim2, output_dim)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.fc1(x))
        x = self.sigmoid(self.fc2(x))
        x = self.sigmoid(self.fc3(x))
        return x

# Hyperparameters
input_dim = 2  # for XOR
hidden_dim1 = 10
hidden_dim2 = 10
output_dim = 1
lr = 0.1
epochs = 1000

# Training XOR
model_xor = THLNN(input_dim, hidden_dim1, hidden_dim2, output_dim)
criterion_xor = nn.BCELoss()
optimizer_xor = optim.SGD(model_xor.parameters(), lr=lr)

for epoch in range(epochs):
    optimizer_xor.zero_grad()
    output = model_xor(X_xor)
    loss = criterion_xor(output, y_xor)
    loss.backward()
    optimizer_xor.step()

# Training Sine Regression
input_dim = 1  # for Sine
hidden_dim1 = 10
hidden_dim2 = 10
output_dim = 1

model_sine = THLNN(input_dim, hidden_dim1, hidden_dim2, output_dim)
criterion_sine = nn.MSELoss()
optimizer_sine = optim.SGD(model_sine.parameters(), lr=lr)

for epoch in range(epochs):
    optimizer_sine.zero_grad()
```

```
    output = model_sine(X_sine)
    loss = criterion_sine(output, y_sine)
    loss.backward()
    optimizer_sine.step()

# Testing XOR
test_output_xor = model_xor(X_xor)
print("XOR Predictions:", test_output_xor.round())

# Testing Sine Regression
test_output_sine = model_sine(X_sine)
```

```
XOR Predictions: tensor([[0.],
        [0.],
        [1.],
        [1.]], grad_fn=<RoundBackward0>)
```

Answer:

XOR Task: A single hidden layer neural network with enough neurons (usually 2 neurons are enough for XOR) can learn the XOR function because XOR is linearly inseparable. Similarly, a two hidden layer neural network with appropriate dimensions can also learn XOR.

Sine Regression Task: A neural network, whether single or multi-layered, can approximate a continuous function such as sine given enough capacity. As long as the network has enough neurons and is trained properly, it should be able to learn the sine function

Part 2: Model training (25 points)

Same concept as before, we'll make two datasets:

Devise a neural network for each of these - between 2 and 10 layers, hidden dimension sizes between 10 and 1000, your choice of activation functions, optimizers, etc. Use simple test/train splits. Train your models and report on training performance w.r.t. loss and metric(s) chosen. Justify your choices of loss, metrics (if any), and any hyperparameters. The focus is on making something work and that you can improve the model, not make it work perfectly.

a. Random multi-dimensional data (8 points)

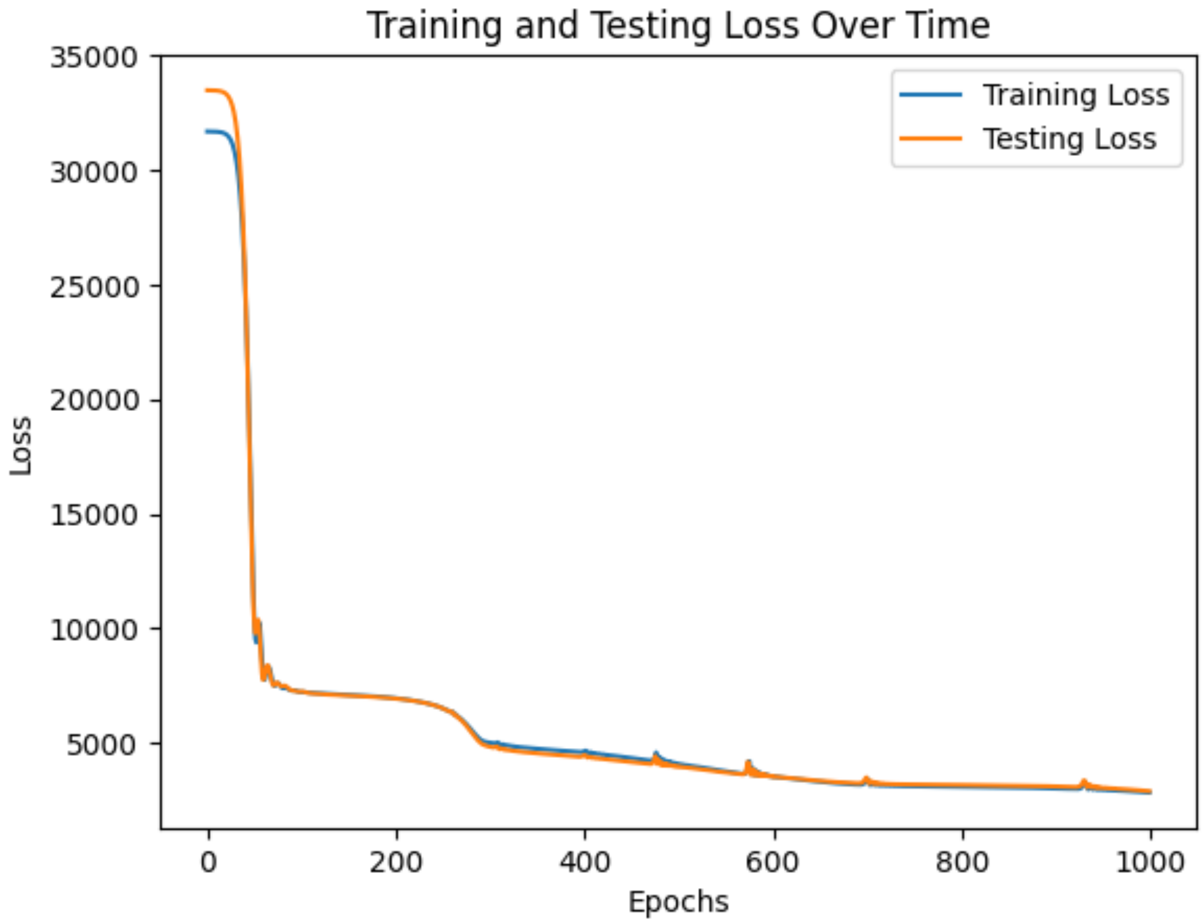
16 input variables/features, 10 output variables, 1000 samples

In [6]:
```
# Generate random multi-dimensional data
```

```
X, y = make_regression(n_samples=1000, n_features=16, n_targets=10, noise

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)

# Define neural network architecture
class MultiLayerNN(nn.Module):
    def __init__(self, input_dim, output_dim, num_layers, hidden_dim):
        super(MultiLayerNN, self).__init__()
        self.input_layer = nn.Linear(input_dim, hidden_dim)
        self.hidden_layers = nn.ModuleList([nn.Linear(hidden_dim, hidden_
        self.output_layer = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.input_layer(x))
        for layer in self.hidden_layers:
            x = self.relu(layer(x))
        x = self.output_layer(x)
        return x

# Define training function
def train_model(model, criterion, optimizer, X_train, y_train, X_test, y_
    train_losses = []
    test_losses = []
    for epoch in range(num_epochs):
        optimizer.zero_grad()
        outputs = model(X_train)
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()
        train_losses.append(loss.item())

        # Compute test loss
        model.eval()
        with torch.no_grad():
            test_outputs = model(X_test)
            test_loss = criterion(test_outputs, y_test)
            test_losses.append(test_loss.item())

        model.train()

        if (epoch+1) % 100 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {loss.ite
    return train_losses, test_losses

# Hyperparameters
input_dim = X_train.shape[1]
output_dim = y_train.shape[1]
num_layers = 5  # Vary between 2 and 10
hidden_dim = 100  # Vary between 10 and 1000
lr = 0.001
batch_size = 32
```

```
num_epochs = 1000

# Create and train the model
model = MultiLayerNN(input_dim, output_dim, num_layers, hidden_dim)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=lr)

train_losses, test_losses = train_model(model, criterion, optimizer, X_tr

# Plotting training and testing losses
plt.plot(train_losses, label='Training Loss')
plt.plot(test_losses, label='Testing Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Testing Loss Over Time')
plt.legend()
plt.show()
```

```
Epoch [100/1000], Train Loss: 7215.7002, Test Loss: 7236.5122
Epoch [200/1000], Train Loss: 6944.6313, Test Loss: 6930.0557
Epoch [300/1000], Train Loss: 5003.3955, Test Loss: 4829.8721
Epoch [400/1000], Train Loss: 4612.2817, Test Loss: 4445.0293
Epoch [500/1000], Train Loss: 4066.9561, Test Loss: 3953.0774
Epoch [600/1000], Train Loss: 3537.3125, Test Loss: 3513.8450
Epoch [700/1000], Train Loss: 3384.3840, Test Loss: 3481.7651
Epoch [800/1000], Train Loss: 3071.6438, Test Loss: 3149.9534
Epoch [900/1000], Train Loss: 3003.3904, Test Loss: 3085.9910
Epoch [1000/1000], Train Loss: 2822.9780, Test Loss: 2880.3965
```



b. MNIST (8 points)

In [7]:
```
# Define transformations to be applied to the dataset
transform = transforms.Compose([
    transforms.ToTensor(),  # Convert PIL Image to tensor
    transforms.Normalize((0.5,), (0.5,))  # Normalize the pixel values to
])

# Load MNIST dataset
trainset = torchvision.datasets.MNIST(root='./data', train=True, download
testset = torchvision.datasets.MNIST(root='./data', train=False, download

# Define data loaders
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffl
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=

# Define neural network architecture
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)
        self.relu = nn.ReLU()
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten the input tensor
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return self.softmax(x)

# Initialize the model, loss function, and optimizer
model = MLP()
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training the model
num_epochs = 10
train_losses = []
test_losses = []

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in trainloader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    train_loss = running_loss / len(trainloader)
```

```
    train_losses.append(train_loss)

    # Evaluate on test set
    model.eval()
    test_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testloader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            test_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    test_loss /= len(testloader)
    test_losses.append(test_loss)
    print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f},

# Plotting training and testing losses
plt.plot(train_losses, label='Training Loss')
plt.plot(test_losses, label='Testing Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Testing Loss Over Time')
plt.legend()
plt.show()
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz t
o ./data/MNIST/raw/train-images-idx3-ubyte.gz
```

```
100%|███████████████████████████████████████████████████████████████████
██████████████████████████████████| 9912422/9912422 [00:00<00:00, 1149371
2.29it/s]
```

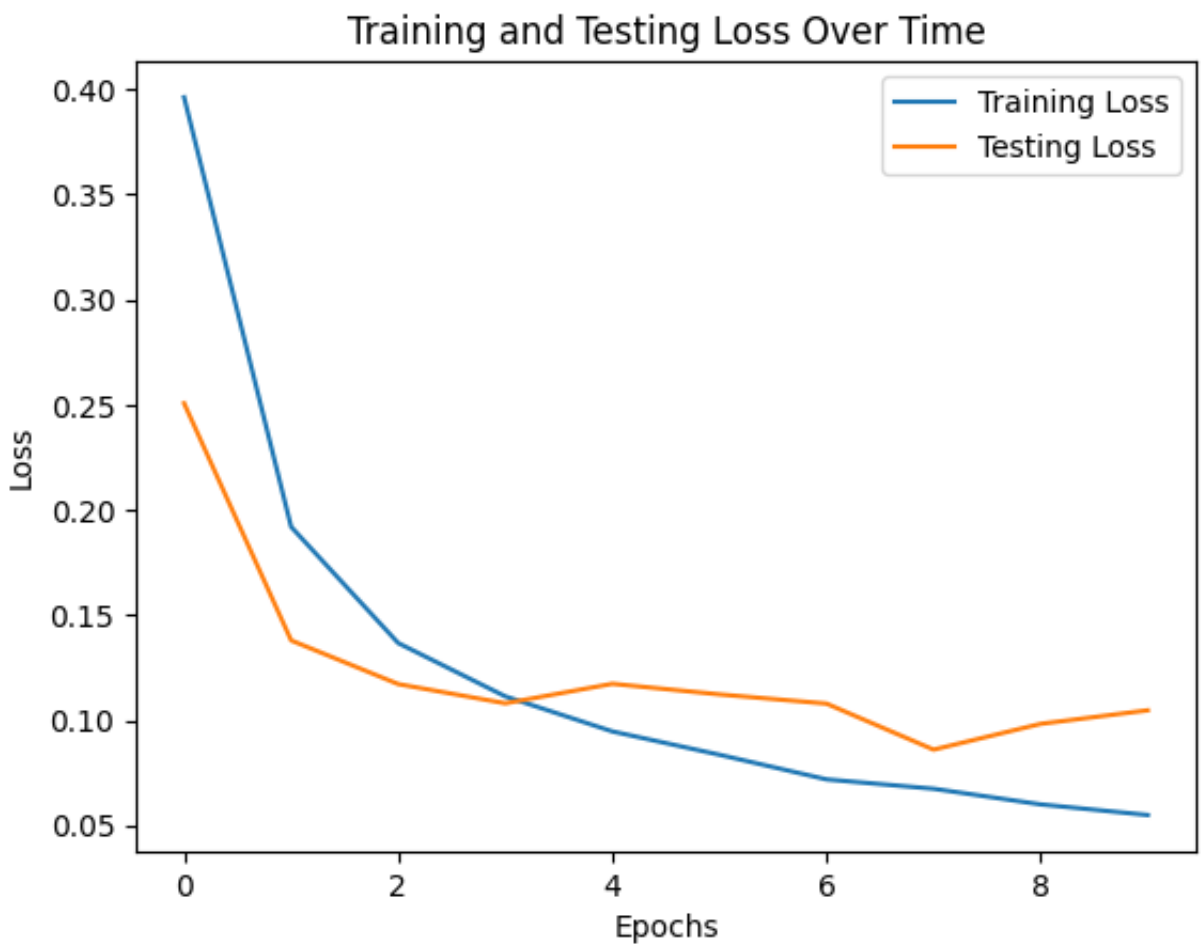
```
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/ra
w

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz t
o ./data/MNIST/raw/train-labels-idx1-ubyte.gz
```

```
100%|███████████████████████████████████████████████████████████████████
██████████████████████████████████████| 28881/28881 [00:00<00:00, 4882535
0.19it/s]
```

```
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/ra
w

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
./data/MNIST/raw/t10k-images-idx3-ubyte.gz
```

```
100%|███████████████████████████████████████████████████████████████████
██████████████████████████████████| 1648877/1648877 [00:00<00:00, 1064958
1.38it/s]
```

```
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|████████████████████████████████████████████████████████████████████
████████████████████████████████████████| 4542/4542 [00:00<00:00, 1241076
7.93it/s]
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw

Epoch [1/10], Train Loss: 0.3961, Test Loss: 0.2507, Test Accuracy: 92.4
4%
Epoch [2/10], Train Loss: 0.1919, Test Loss: 0.1379, Test Accuracy: 95.7
2%
Epoch [3/10], Train Loss: 0.1367, Test Loss: 0.1172, Test Accuracy: 96.4
1%
Epoch [4/10], Train Loss: 0.1113, Test Loss: 0.1080, Test Accuracy: 96.6
5%
Epoch [5/10], Train Loss: 0.0947, Test Loss: 0.1173, Test Accuracy: 96.1
6%
Epoch [6/10], Train Loss: 0.0836, Test Loss: 0.1123, Test Accuracy: 96.4
4%
Epoch [7/10], Train Loss: 0.0719, Test Loss: 0.1079, Test Accuracy: 96.5
6%
Epoch [8/10], Train Loss: 0.0674, Test Loss: 0.0860, Test Accuracy: 97.4
3%
Epoch [9/10], Train Loss: 0.0600, Test Loss: 0.0983, Test Accuracy: 96.8
9%
Epoch [10/10], Train Loss: 0.0549, Test Loss: 0.1047, Test Accuracy: 96.6
5%
```

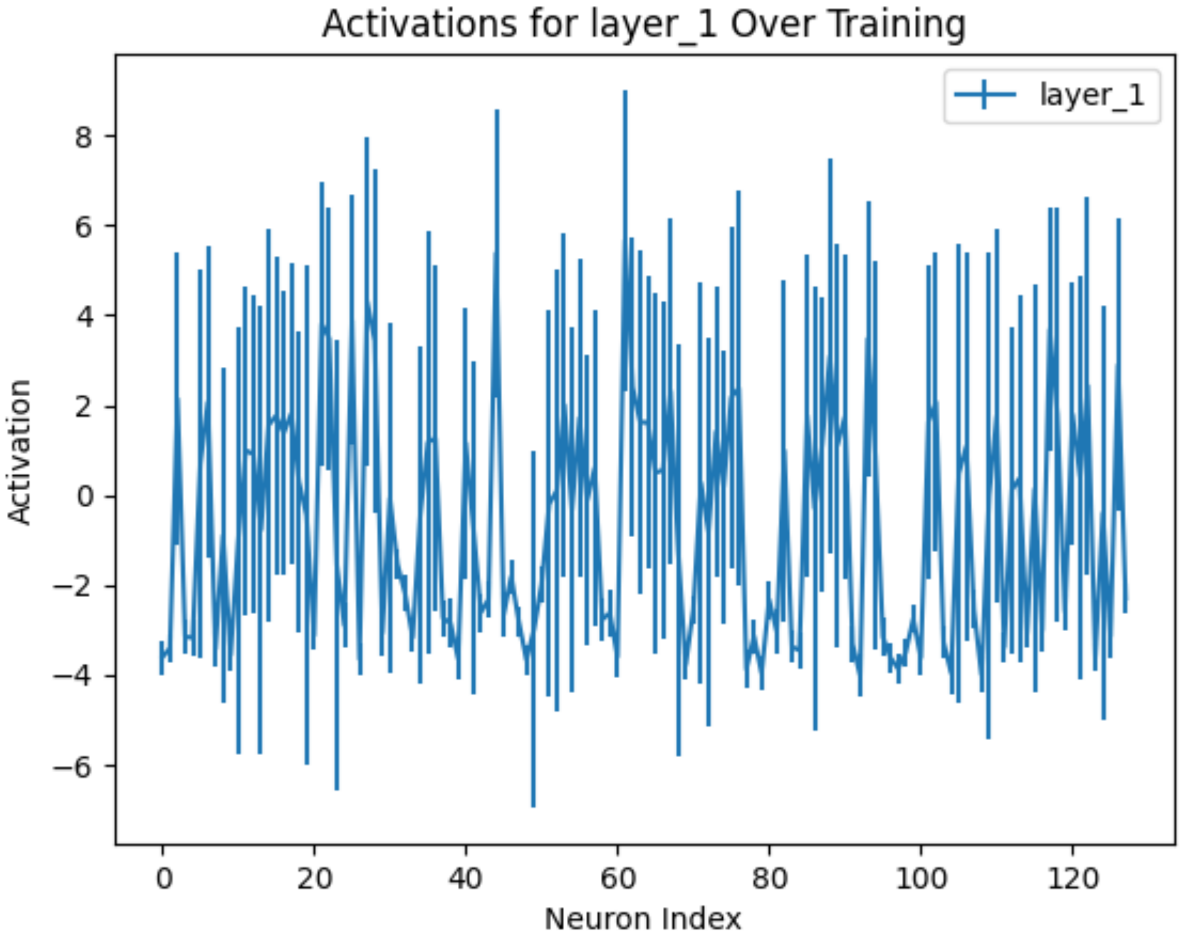


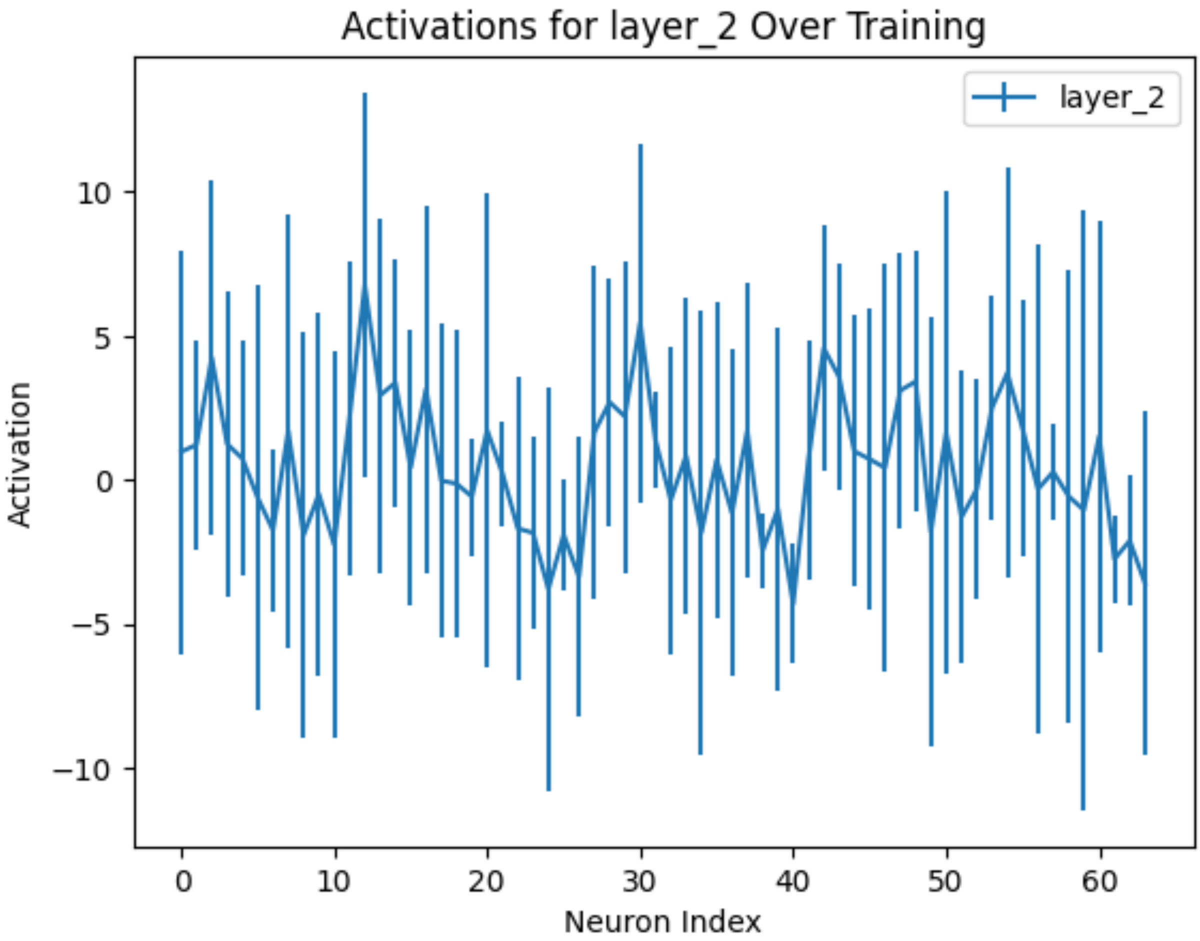
Deeper knowledge of model performance (9 points)

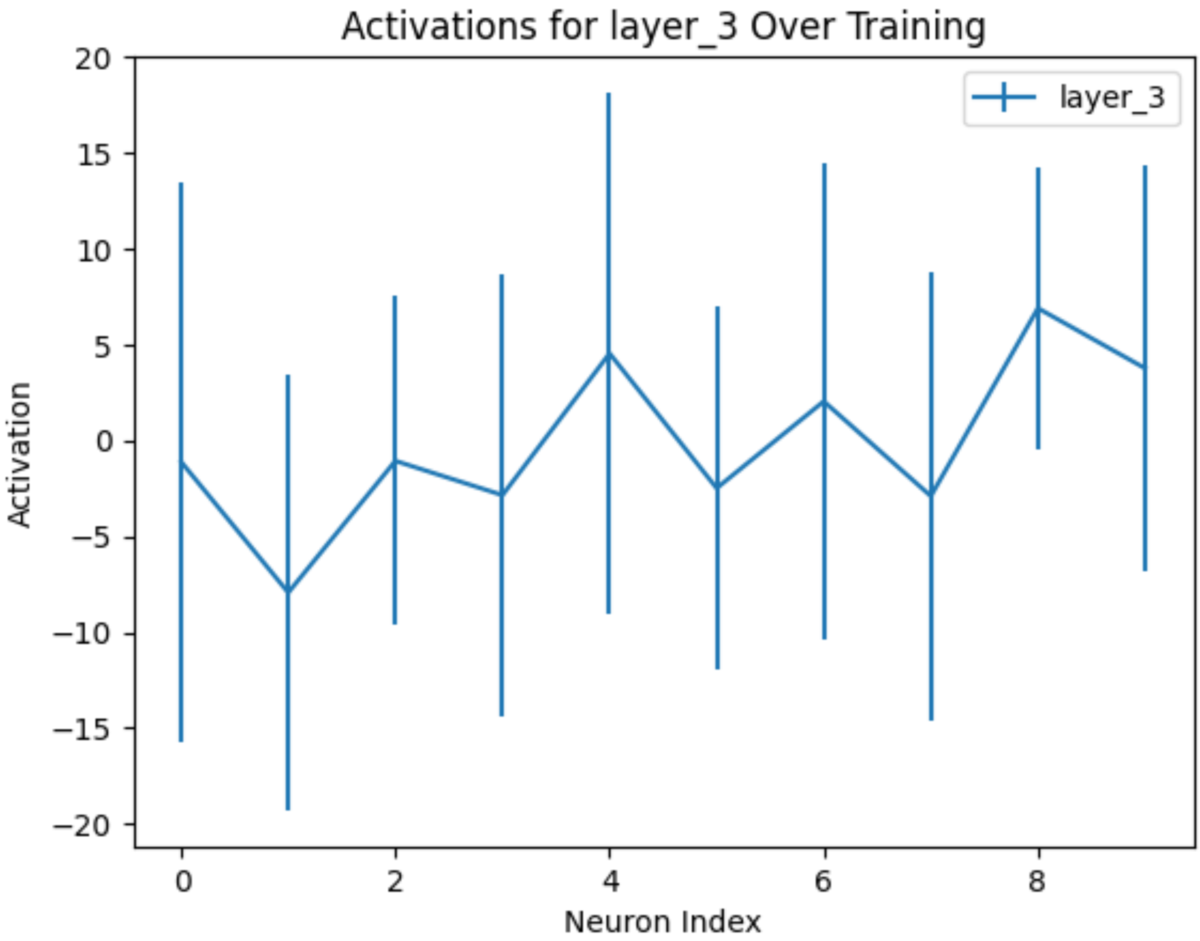
Pick one of the previous test cases in this section. Try visualizing the gradients and/or activations for different layers. What can you learn from this?

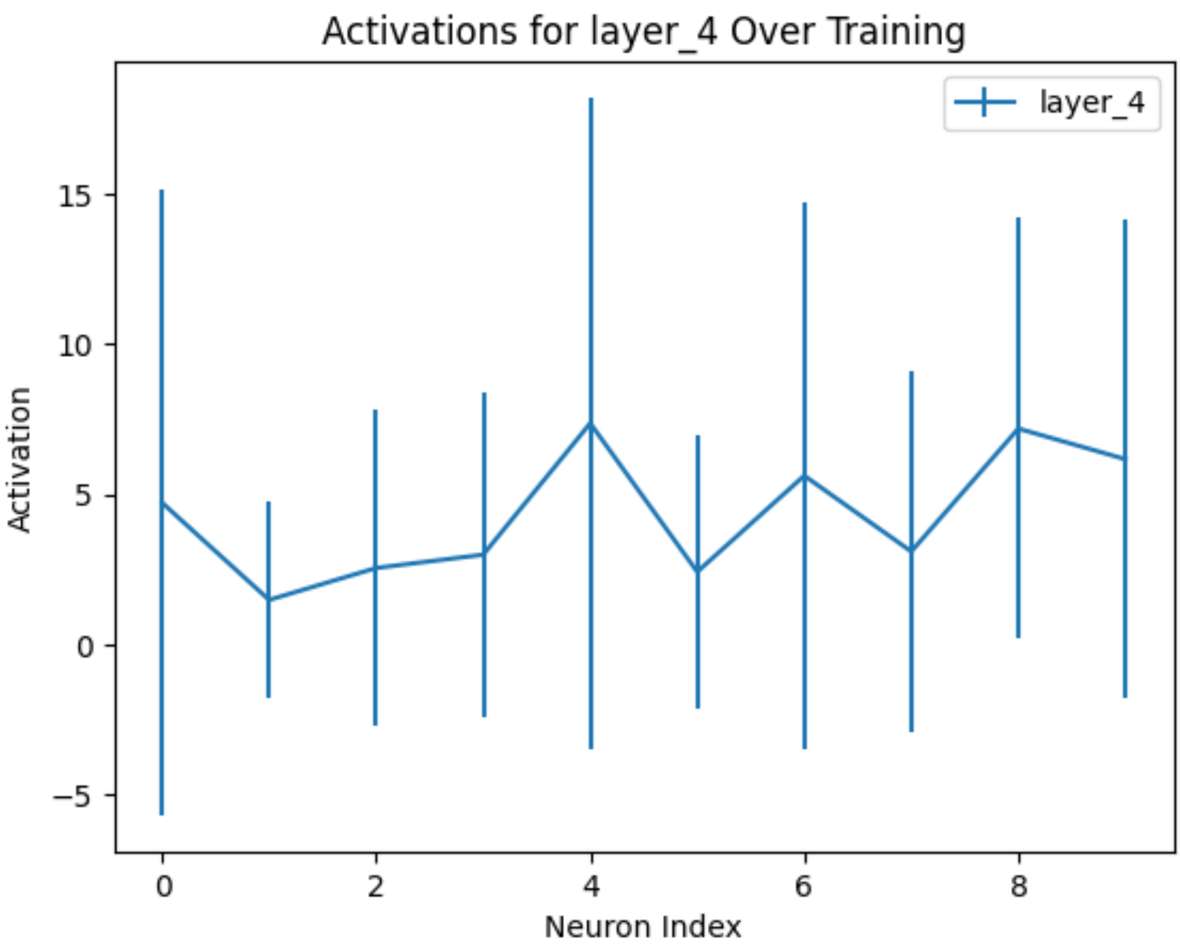
In [12]:
```
def train_model_with_activations(model, criterion, optimizer, trainloader
    train_losses = []
    num_layers = len(list(model.children()))
    activations = {f'layer_{i}': [] for i in range(1, num_layers + 1)}  #
    for epoch in range(num_epochs):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data
            optimizer.zero_grad()

            # Flatten the input images
            inputs = inputs.view(inputs.size(0), -1)

            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

            # Save activations for each layer
            layer_input = inputs
            for i, layer in enumerate(model.children(), 1):
                if isinstance(layer, nn.Linear) or isinstance(layer, nn.R
                    layer_input = layer(layer_input)
                    if layer_input.numel() > 0:  # Check if activations a
                        activations[f'layer_{i}'].append(layer_input.deta
        train_loss = running_loss / len(trainloader)
        train_losses.append(train_loss)
        print(f"Epoch {epoch+1}/{num_epochs}, Training Loss: {train_loss:
    return train_losses, activations

# Create and train the model with activations
model = MLP()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
train_losses, activations = train_model_with_activations(model, criterion

# Plot activations for each layer
for layer_name, layer_activations in activations.items():
    if layer_activations:  # Check if activations are non-empty
        layer_activations = np.concatenate(layer_activations, axis=0)
        mean_activation = np.mean(layer_activations, axis=0)
        std_activation = np.std(layer_activations, axis=0)

        plt.errorbar(range(mean_activation.shape[0]), mean_activation, ye
        plt.xlabel('Neuron Index')
        plt.ylabel('Activation')
        plt.title(f'Activations for {layer_name} Over Training')
        plt.legend()
        plt.show()
    else:
        print(f"No activations saved for {layer_name}.")
```

```
Epoch 1/5, Training Loss: 0.4024
Epoch 2/5, Training Loss: 0.1903
Epoch 3/5, Training Loss: 0.1358
Epoch 4/5, Training Loss: 0.1081
Epoch 5/5, Training Loss: 0.0919
```




Activations for layer_2 Over Training
layer_2
Activation
Neuron Index
10
5
0
−5
−10
0
10
20
30
40
50
60


Activations for layer_3 Over Training
layer_3
Activation
Neuron Index
20
15
10
5
0
−5
−10
−15
−20
0
2
4
6
8




```
No activations saved for layer_5.
```

By plotting the training loss over epochs, we can monitor how the loss decreases over time. This helps us understand if the model is learning and converging towards a solution.

Visualizing activations for different layers helps us understand how information flows through the network. We can observe which neurons are activated more frequently and how their activations change during training.

Analyzing gradients can help detect issues like vanishing or exploding gradients, which can hinder training. If gradients vanish, it indicates that the model is having difficulty updating the weights of certain layers, possibly due to saturation of activation functions or deep network architectures.

Part 3: Intro to Convolutional Neural Networks (25 points)

Recreate the LeNet-5 (https://en.wikipedia.org/wiki/LeNet Links to an external site.) and train it on MNIST. Explain the construction of your model and report test and training loss and accuracy. 20 points for getting the base model working, 5 points for showing that you iterated your hyperparameters to improve performance in some way, possibly looking at loss curves to inform your decision.

The LeNet-5 architecture consists of the following layers:

```
Convolutional Layer 1: 6 filters of size 5x5 with a stride
of 1, followed by ReLU activation.
Max Pooling Layer 1: 2x2 kernel with a stride of 2.
Convolutional Layer 2: 16 filters of size 5x5 with a
stride of 1, followed by ReLU activation.
Max Pooling Layer 2: 2x2 kernel with a stride of 2.
Fully Connected Layer 1: 120 units with ReLU activation.
Fully Connected Layer 2: 84 units with ReLU activation.
Output Layer: 10 units corresponding to the 10 classes in
the MNIST dataset.
```

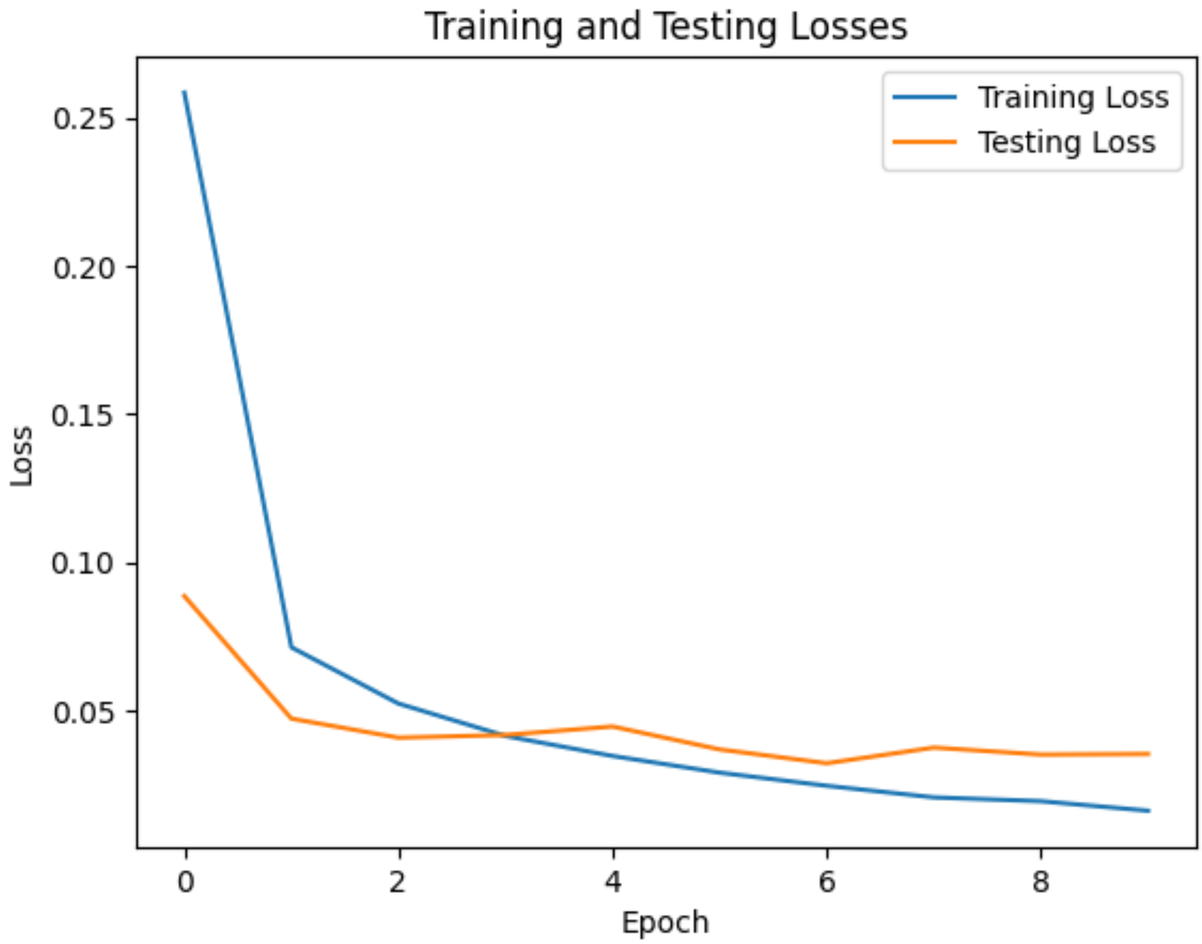
In [18]:

```python
# Define LeNet-5 architecture
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.relu3 = nn.ReLU()
        self.fc2 = nn.Linear(120, 84)
        self.relu4 = nn.ReLU()
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)
        x = x.view(-1, 16 * 4 * 4)
        x = self.fc1(x)
        x = self.relu3(x)
        x = self.fc2(x)
        x = self.relu4(x)
```

```
        x = self.fc3(x)
        return x


# Load MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

trainset = torchvision.datasets.MNIST(root='./data', train=True, download
testset = torchvision.datasets.MNIST(root='./data', train=False, download

trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffl
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=

# Initialize LeNet-5 model
model = LeNet5()

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Lists to store training and testing losses
train_losses = []
test_losses = []

# Train the model
num_epochs = 10
for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    # Calculate and store training loss
    train_loss = running_loss / len(trainloader)
    train_losses.append(train_loss)
    print(f"Epoch {epoch+1}/{num_epochs}, Training Loss: {train_loss:.4f}

    # Test the model
    test_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            outputs = model(images)
            loss = criterion(outputs, labels)
            test_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
```

```
    # Calculate and store testing loss
    test_loss /= len(testloader)
    test_losses.append(test_loss)
    print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {100 * correct / t

# Plot training and testing losses
plt.plot(train_losses, label='Training Loss')
plt.plot(test_losses, label='Testing Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Testing Losses')
plt.legend()
plt.show()

# Print final test accuracy
print(f"Final Test Accuracy: {100 * correct / total:.2f}%")
```

```
Epoch 1/10, Training Loss: 0.2584
Test Loss: 0.0886, Test Accuracy: 97.10%
Epoch 2/10, Training Loss: 0.0713
Test Loss: 0.0473, Test Accuracy: 98.27%
Epoch 3/10, Training Loss: 0.0524
Test Loss: 0.0409, Test Accuracy: 98.69%
Epoch 4/10, Training Loss: 0.0414
Test Loss: 0.0418, Test Accuracy: 98.72%
Epoch 5/10, Training Loss: 0.0347
Test Loss: 0.0446, Test Accuracy: 98.66%
Epoch 6/10, Training Loss: 0.0291
Test Loss: 0.0369, Test Accuracy: 98.74%
Epoch 7/10, Training Loss: 0.0246
Test Loss: 0.0322, Test Accuracy: 98.88%
Epoch 8/10, Training Loss: 0.0207
Test Loss: 0.0375, Test Accuracy: 98.95%
Epoch 9/10, Training Loss: 0.0195
Test Loss: 0.0352, Test Accuracy: 98.95%
Epoch 10/10, Training Loss: 0.0162
Test Loss: 0.0354, Test Accuracy: 98.93%
```




```
Final Test Accuracy: 98.93%
```

```
In [ ]:
```