Robustness of AI Models Against Adversarial Attacks – MNIST (PyTorch)

Task 01
Code

Rajavardhan_Ramya_Assignment_03.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text

Task 1: Implementation of a CNN in PyTorch for MNIST Classification

Step 1: Load Packages and Import Required Libraries

```
[2] import torch  # PyTorch library
    import torch.nn as nn  # Neural network module
    import torch.optim as optim  # Optimization algorithms
    import torch.nn.functional as F  # Activation functions
    from torch.utils.data import DataLoader, TensorDataset, random_split  # Data loading utilities
    from torchvision import datasets, transforms  # Datasets and transformations
    import matplotlib.pyplot as plt  # Visualization library
```

Step 2: Load the Dataset into the Google Colab & Data Preprocessing

```
[3] # Check if GPU (CUDA) is available and set the device accordingly
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # Data transformation: Convert images to tensors and normalize them
    transform = transforms.Compose([
        transforms.ToTensor(),  # Convert image from [0, 255] range to [0.0, 1.0]
        transforms.Normalize((0.5,), (0.5,))  # Normalize: mean=0.5, std=0.5
    ])

    # Define batch size for training
    BATCH_SIZE = 64

    # Download and load the MNIST dataset (train & test)
    train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
    test_dataset = datasets.MNIST(root='./data', train=False, transform=transform, download=True)

    # Split training dataset into training (80%) and validation (20%) sets
    train_size = int(0.8 * len(train_dataset))  # 80% for training
    val_size = len(train_dataset) - train_size  # 20% for validation
    train_dataset, val_dataset = random_split(train_dataset, [train_size, val_size])  # Randomly split data

    # Create data loaders for training, validation, and testing
    train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

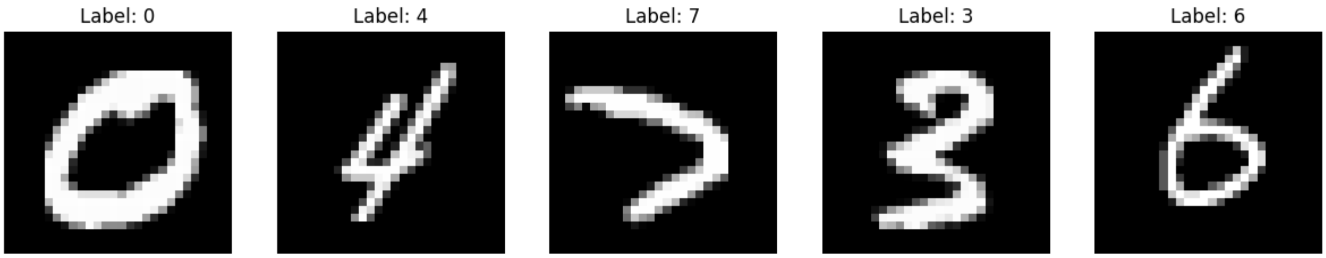
```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9.91M/9.91M [00:01<00:00, 5.25MB/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28.9k/28.9k [00:00<00:00, 153kB/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 1.65M/1.65M [00:01<00:00, 1.46MB/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
Failed to download (trying next):
HTTP Error 404: Not Found
```

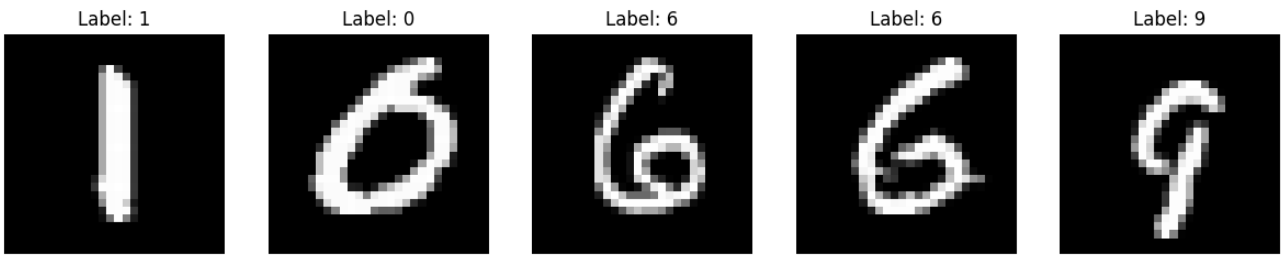
Step 3: Print Information to Understand the MNIST Dataset

[4]

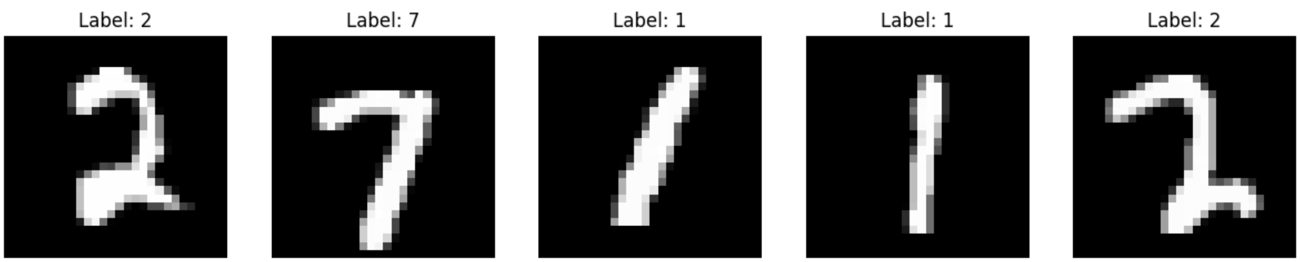
```python
# Print dataset sizes
print(f"Training Dataset Length: {len(train_dataset)}")
print(f"Validation Dataset Length: {len(val_dataset)}")
print(f"Test Dataset Length: {len(test_dataset)}")

# Function to visualize a few images from the dataset
# Print 10 samples in each dataset and their corresponding labels
def show_images_and_labels(dataset, num_images=10):
    plt.figure(figsize=(15, 10))
    for i in range(num_images):
        image, label = dataset[i]  # Get image and label
        plt.subplot(2, 5, i + 1)  # Arrange images in 2 rows, 5 columns
        plt.imshow(image.squeeze(), cmap="gray")  # Show grayscale image
        plt.title(f"Label: {label}")  # Display corresponding label
        plt.axis('off')  # Hide axis lines
    plt.show()

# Display samples from datasets
print("Training Dataset Samples:")
show_images_and_labels(train_dataset)

print("Validation Dataset Samples:")
show_images_and_labels(val_dataset)

print("Test Dataset Samples:")
show_images_and_labels(test_dataset)
```

```
Training Dataset Length: 48000
Validation Dataset Length: 12000
Test Dataset Length: 10000
Training Dataset Samples:
```

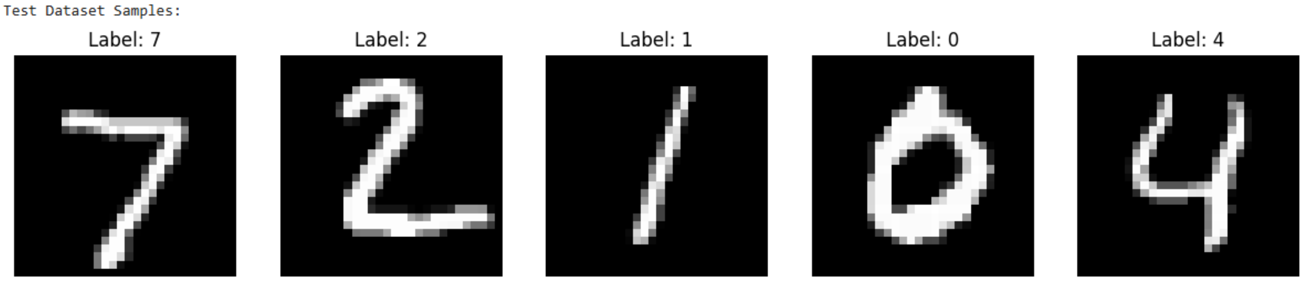


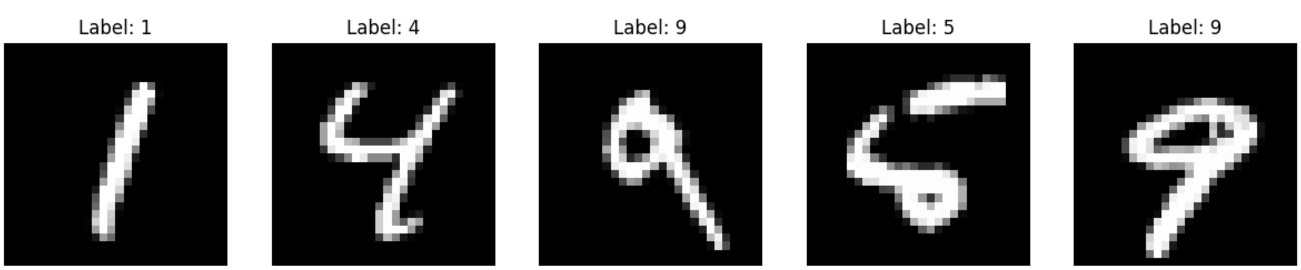



```
Validation Dataset Samples:
```




```
Test Dataset Samples:
```





Step 4: Define CNN Model

```
# Define a CNN Model for MNIST classification
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # First convolutional layer: Input (1 channel), Output (32 channels), Kernel size 3x3
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)  # Max pooling layer (2x2)
        # Second convolutional layer: Input (32 channels), Output (64 channels)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        # Fully connected layers
        self.fc1 = nn.Linear(64 * 7 * 7, 128)  # Flattened input to 128 neurons
        self.dropout = nn.Dropout(0.2)  # Dropout layer (prevents overfitting)
        self.fc2 = nn.Linear(128, 64)  # Fully connected layer (64 neurons)
        self.fc3 = nn.Linear(64, 10)  # Output layer (10 classes for digits 0-9)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))  # Conv1 -> ReLU -> MaxPool
        x = self.pool(F.relu(self.conv2(x)))  # Conv2 -> ReLU -> MaxPool
        x = x.view(-1, 64 * 7 * 7)  # Flatten the output for the fully connected layer
        x = F.relu(self.fc1(x))  # Fully connected layer with ReLU
        x = self.dropout(x)  # Apply dropout
        x = F.relu(self.fc2(x))  # Second fully connected layer with ReLU
        x = self.fc3(x)  # Output layer (no activation, raw scores)
        return x
```

Step 5: Instantiate our CNN Model and Train the Model

```
# Create CNN model instance and move it to the selected device (CPU/GPU)
model = CNN().to(device)

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

# Training function
def train_model(model, train_loader, val_loader, optimizer, num_epochs, model_save_path):
    best_val_loss = float('inf')  # Track lowest validation loss
    for epoch in range(num_epochs):
        model.train()  # Set model to training mode
        train_loss = 0.0
        for data, target in train_loader:
            data, target = data.to(device), target.to(device)  # Move data to device
            optimizer.zero_grad()  # Reset gradients
            output = model(data)  # Forward pass
            loss = F.cross_entropy(output, target)  # Compute loss
            loss.backward()  # Backpropagation
            optimizer.step()  # Update weights
            train_loss += loss.item()  # Accumulate training loss
```

```python
    # Validation step
    val_loss = 0.0
    correct = 0
    total = 0
    model.eval()  # Set model to evaluation mode
    with torch.no_grad():
        for data, target in val_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            loss = F.cross_entropy(output, target)
            val_loss += loss.item()

            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
            total += target.size(0)

    avg_train_loss = train_loss / len(train_loader)
    avg_val_loss = val_loss / len(val_loader)
    val_accuracy = correct / total

    print(f"Epoch {epoch+1}/{num_epochs}, Training Loss: {avg_train_loss:.4f}, Validation Loss: {avg_val_loss:.4f}, Validation Accuracy: {val_accuracy:.4f}"

    # Save the best model based on validation loss
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        torch.save(model.state_dict(), model_save_path)
        print(f"Model saved at Epoch {epoch+1}")
```

```python
# Train the model for 10 epochs and save the best model
model_save_path = 'best_model.pth'
train_model(model, train_loader, val_loader, optimizer, num_epochs=10, model_save_path=model_save_path)
```

```
Epoch 1/10, Training Loss: 0.2446, Validation Loss: 0.0703, Validation Accuracy: 0.9777
Model saved at Epoch 1
Epoch 2/10, Training Loss: 0.0675, Validation Loss: 0.0567, Validation Accuracy: 0.9824
Model saved at Epoch 2
Epoch 3/10, Training Loss: 0.0505, Validation Loss: 0.0399, Validation Accuracy: 0.9877
Model saved at Epoch 3
Epoch 4/10, Training Loss: 0.0409, Validation Loss: 0.0415, Validation Accuracy: 0.9871
Epoch 5/10, Training Loss: 0.0328, Validation Loss: 0.0461, Validation Accuracy: 0.9849
Epoch 6/10, Training Loss: 0.0268, Validation Loss: 0.0481, Validation Accuracy: 0.9858
Epoch 7/10, Training Loss: 0.0232, Validation Loss: 0.0432, Validation Accuracy: 0.9888
Epoch 8/10, Training Loss: 0.0214, Validation Loss: 0.0453, Validation Accuracy: 0.9877
Epoch 9/10, Training Loss: 0.0171, Validation Loss: 0.0356, Validation Accuracy: 0.9889
Model saved at Epoch 9
Epoch 10/10, Training Loss: 0.0188, Validation Loss: 0.0456, Validation Accuracy: 0.9885
```

The accuracy achieved for our best model saved is 98.85%

Step 6: Evaluate the Trained Model on Testing Dataset

```python
# Function to test the trained model on the test dataset
def test_model(model, test_loader, num_samples=100):
    model.eval()  # Set model to evaluation mode (no weight updates)
    test_loss = 0.0
    correct = 0
    sample_count = 0  # To track how many sample predictions are printed

    with torch.no_grad():  # Disable gradient calculation for efficiency
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)  # Move data to GPU/CPU
            output = model(data)  # Forward pass
            test_loss += F.cross_entropy(output, target).item()  # Compute loss
            pred = output.argmax(dim=1, keepdim=True)  # Get predicted class index
            correct += pred.eq(target.view_as(pred)).sum().item()  # Count correct predictions
```

```python
            # Print a few sample predictions (up to 'num_samples')
            if sample_count < num_samples:
                print("\nSample predictions:")
                for i in range(data.size(0)):  # Iterate through batch
                    if sample_count >= num_samples:
                        break  # Stop if we've printed enough samples
                    print(f"Sample {sample_count + 1}:")
                    print(f"  True label: {target[i].item()}")
                    print(f"  Predicted raw output: {output[i]}")
                    print(f"  Predicted label: {pred[i].item()}")
                    sample_count += 1

    # Compute average test loss and accuracy
    avg_test_loss = test_loss / len(test_loader.dataset)
    accuracy = correct / len(test_loader.dataset)

    # Print final results
    print(f"\nTest Loss: {avg_test_loss:.4f}, Accuracy: {accuracy:.4f}")

# Function to load a saved model and evaluate it on the test dataset
def load_and_test_model(model, model_load_path, test_loader):
    model.load_state_dict(torch.load(model_load_path))  # Load trained model weights
    model.to(device)  # Move model to GPU/CPU
    test_model(model, test_loader)  # Run testing function

# Define device (CPU/GPU) again for consistency
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```python
# Create a new instance of the CNN model and move it to the device
model = CNN().to(device)

# Path to the best saved model
model_load_path = 'best_model.pth'

# Load the saved model and evaluate it on the test set
load_and_test_model(model, model_load_path, test_loader)
```

```
<ipython-input-7-b1fefecd0347>:37: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default valu
  model.load_state_dict(torch.load(model_load_path))  # Load trained model weights

Sample predictions:
Sample 1:
  True label: 7
  Predicted raw output: tensor([ -4.2453,   0.5440,   0.5945,   1.1885,  -0.4651,  -6.1687, -17.1389,
         17.5853,  -4.3814,   1.3302], device='cuda:0')
  Predicted label: 7
Sample 2:
  True label: 2
  Predicted raw output: tensor([  0.6863,   1.3256,  26.3435, -15.1304,   3.3406, -13.5158,  -7.6641,
         -1.9197,  -9.8477,  -9.3949], device='cuda:0')
  Predicted label: 2
Sample 3:
  True label: 1
  Predicted raw output: tensor([-6.1622, 13.0728, -4.1660, -8.5453,  0.2181, -1.6540, -4.1899,  1.3943,
        -2.4331, -3.1920], device='cuda:0')
  Predicted label: 1
Sample 4:
  True label: 0
```

```
  Predicted label: 4
Sample 97:
  True label: 1
  Predicted raw output: tensor([-3.2667,  8.1907, -5.0556, -3.2043,  0.2178, -0.3725, -8.8388, -0.1465,
        -5.2179,  1.0343], device='cuda:0')
  Predicted label: 1
Sample 98:
  True label: 7
  Predicted raw output: tensor([-11.9003,   2.0603,   8.1632,  -2.0614,   0.5244,  -8.6684, -16.3370,
         16.1216,  -1.0232,  -2.0974], device='cuda:0')
  Predicted label: 7
Sample 99:
  True label: 6
  Predicted raw output: tensor([  6.5024,  -7.9505, -12.7344, -13.0487,  -6.3549,   3.0197,  17.1773,
        -19.9466,   4.0353,  -7.4932], device='cuda:0')
  Predicted label: 6
Sample 100:
  True label: 9
  Predicted raw output: tensor([ -6.1895, -15.5481,  -6.2737,  -4.6080,   3.5611,  -3.8814, -18.3659,
         -3.0806,  -0.5489,  20.6045], device='cuda:0')
  Predicted label: 9

Test Loss: 0.0006, Accuracy: 0.9901
```

The Accuracy achieved for our model is 99%.

Task 02

Code

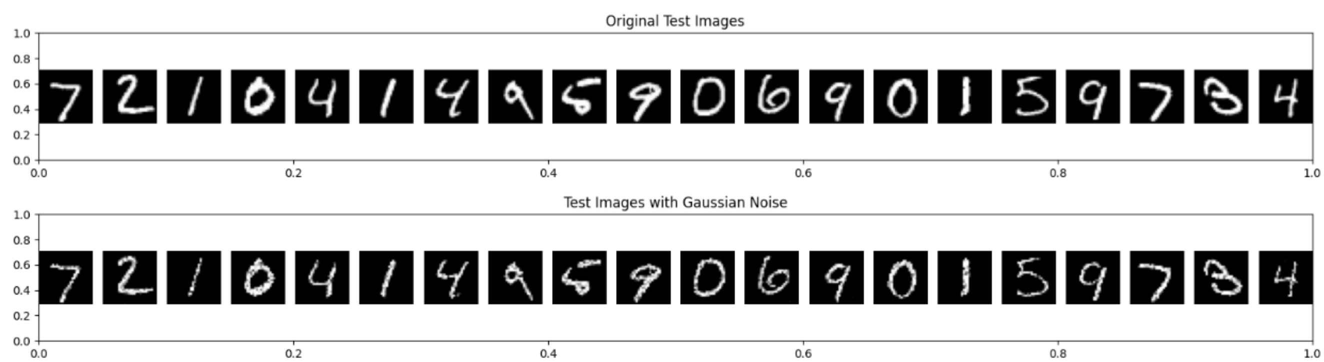
Task 2: Add Gaussian Noises

Step 1: Define the Function & Apply Noise to the Dataset

```python
# Function to add Gaussian noise to an image
def add_gaussian_noise(img, mean=0., std=0.1):
    """
    Adds Gaussian noise to a given image.

    Parameters:
        img (Tensor): Input image tensor.
        mean (float): Mean of the Gaussian noise.
        std (float): Standard deviation (amount of noise added).

    Returns:
        Tensor: Noisy image tensor, clamped to [0,1] range to keep valid pixel values.
    """
    noise = torch.randn(img.size()) * std + mean  # Generate Gaussian noise
    noisy_img = img + noise  # Add noise to the image
    return noisy_img.clamp(0, 1)  # Clamp values to keep them between [0,1]

# Define transformation pipeline with Gaussian noise
transform_with_noise = transforms.Compose([
    transforms.ToTensor(),  # Convert image to PyTorch tensor
    transforms.Normalize((0.5,), (0.5,)),  # Normalize image (mean=0.5, std=0.5)
    transforms.Lambda(lambda x: add_gaussian_noise(x, std=0.3))  # Apply Gaussian noise (std = 0.3)
```

```python
# Load the MNIST test dataset with Gaussian noise applied
test_dataset_with_noise = datasets.MNIST(
    root='./data', train=False, transform=transform_with_noise, download=True
)

# Create DataLoader for noisy test dataset
test_loader_with_noise = DataLoader(test_dataset_with_noise, batch_size=BATCH_SIZE, shuffle=False)

# Function to display images with or without noise
def show_images(dataset, idxs, title):
    """
    Displays a set of images from the dataset.

    Parameters:
        dataset: The dataset containing images.
        idxs (list): Indices of images to display.
        title (str): Title of the plot.
    """
    plt.figure(figsize=(20, 2))  # Set figure size
    plt.title(title)  # Set title
    for i, idx in enumerate(idxs):  # Loop through selected images
        ax = plt.subplot(1, len(idxs), i + 1)  # Create subplot
        plt.imshow(dataset[idx][0].squeeze(), cmap='gray')  # Display image in grayscale
        plt.axis('off')  # Remove axis
    plt.show()  # Show images
```

```python
# Select first 20 images for visualization
idxs = range(20)

# Show original test images (without noise)
show_images(test_dataset, idxs, "Original Test Images")

# Show test images with Gaussian noise
show_images(test_dataset_with_noise, idxs, "Test Images with Gaussian Noise")
```



Step 2: Evaluate the Model on Noisy Data

```python
print("Testing on noisy data:")
test_model(model, test_loader_with_noise)
```

```
Testing on noisy data:

Sample predictions:
Sample 1:
  True label: 7
  Predicted raw output: tensor([-3.3661,  0.2635,  1.2000,  1.7661, -0.9481, -3.0286, -8.6401,  8.8784,
        -2.5072,  0.2393], device='cuda:0')
  Predicted label: 7
Sample 2:
  True label: 2
  Predicted raw output: tensor([-0.9762,  1.0440, 12.2544, -5.7156,  0.1110, -5.8290, -3.1969, -0.9706,
        -2.4361, -3.6590], device='cuda:0')
  Predicted label: 2
```

```
Sample 98:
  True label: 7
  Predicted raw output: tensor([-5.1224,  0.7929,  4.4320, -0.8378, -1.1478, -2.7462, -7.2831,  5.6359,
         2.6706, -1.5681], device='cuda:0')
  Predicted label: 7
Sample 99:
  True label: 6
  Predicted raw output: tensor([  1.4011,  -3.8564,  -6.5564,  -6.2705,  -4.1876,   3.1007,   7.5644,
        -10.1826,   2.2553,  -3.4920], device='cuda:0')
  Predicted label: 6
Sample 100:
  True label: 9
  Predicted raw output: tensor([-3.7927, -5.7390, -1.5760, -0.8605,  0.4140, -1.2089, -8.6618, -0.0648,
        -1.9444,  7.8907], device='cuda:0')
  Predicted label: 9

Test Loss: 0.0020, Accuracy: 0.9607
```

Here, you can see the accuracy is reduced to 96% after adding the Gaussian noise.

Task 03

Code

Task 3: Add FGSM (Fast Gradient Sign Method) Noises

Step 1: Define the Function and Appply Noise to the Dataset

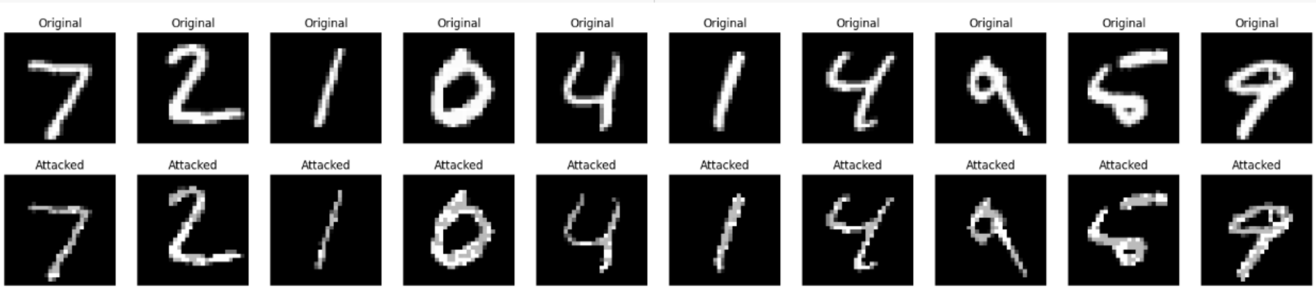
```
[10] # Import necessary libraries
     import torch  # PyTorch library
     import torch.nn.functional as F  # PyTorch functions (e.g., cross-entropy loss)
     import matplotlib.pyplot as plt  # For visualization
     from torchvision import datasets, transforms  # PyTorch datasets and transformations
     from torch.utils.data import DataLoader  # Data loading utilities
```

```
# Function to apply FGSM attack
def fgsm_attack(image, epsilon, data_grad):
    """
    Generates an adversarial example using the Fast Gradient Sign Method (FGSM).

    Parameters:
        image (Tensor): The original input image.
        epsilon (float): The attack strength (higher = more distortion).
        data_grad (Tensor): The gradient of the loss w.r.t. the input image.

    Returns:
        Tensor: The perturbed adversarial image, clamped between [0,1] to keep valid pixel values.
    """
    sign_data_grad = data_grad.sign()  # Get the sign of the gradient
    perturbed_image = image + epsilon * sign_data_grad  # Add perturbation to the image
    perturbed_image = torch.clamp(perturbed_image, 0, 1)  # Ensure values stay in valid range
    return perturbed_image

# Function to apply FGSM attack on the entire test dataset
def test_fgsm_attack(model, device, test_loader, epsilon):
    """
    Generates adversarial examples for all images in the test dataset using FGSM.

    Parameters:
        model (nn.Module): The trained CNN model.
        device (torch.device): The device (CPU or GPU) to use.
        test_loader (DataLoader): DataLoader for the test dataset.
        epsilon (float): The attack strength.
```

```python
    Returns:
        list: A list of adversarial images.
    """
    model.eval()  # Set model to evaluation mode (no training)
    attacked_images = []  # List to store adversarial images

    for data, target in test_loader:
        data, target = data.to(device), target.to(device)  # Move data to GPU/CPU
        data.requires_grad = True  # Enable gradient computation for FGSM attack

        output = model(data)  # Forward pass
        loss = F.cross_entropy(output, target)  # Compute loss
        model.zero_grad()  # Reset gradients
        loss.backward()  # Compute gradients w.r.t. input image
        data_grad = data.grad.data  # Get gradient values

        perturbed_data = fgsm_attack(data, epsilon, data_grad)  # Generate adversarial image
        attacked_images.extend(perturbed_data.detach().cpu())  # Store images in list (convert to CPU first)

    return attacked_images  # Return adversarial images
```

```python
# Function to display original and attacked images side-by-side
def show_images(image_list1, image_list2, title1, title2, num_images=10):
    """
    Displays a set of original and adversarial images side-by-side for comparison.

    Parameters:
        image_list1 (list): List of original images.
        image_list2 (list): List of adversarial (FGSM-attacked) images.
        title1 (str): Title for the original images.
        title2 (str): Title for the adversarial images.
        num_images (int): Number of images to display.
    """
    plt.figure(figsize=(25, 5))  # Set figure size
    for i in range(num_images):  # Loop through selected images
        plt.subplot(2, num_images, i+1)  # Create subplot (row 1: original)
        plt.imshow(image_list1[i].squeeze(), cmap='gray')  # Display original image
        plt.title(title1)  # Set title
        plt.axis('off')  # Remove axis

        plt.subplot(2, num_images, num_images+i+1)  # Create subplot (row 2: adversarial)
        plt.imshow(image_list2[i].squeeze(), cmap='gray')  # Display attacked image
        plt.title(title2)  # Set title
        plt.axis('off')  # Remove axis

    plt.show()  # Show images
```

```python
# Define attack strength (epsilon)
epsilon = 0.25  # Adjust to increase/decrease attack severity

# Generate FGSM-attacked images
attacked_images = test_fgsm_attack(model, device, test_loader, epsilon)

# Select first 10 images for visualization
num_images = 10
original_test_images = [test_dataset[i][0] for i in range(num_images)]  # Extract original images

# Show original vs. FGSM-attacked images
show_images(original_test_images, attacked_images, "Original", "Attacked", num_images)
```



Step 2: Evaluate the Model on Attacked Data

[11]
```python
# Import TensorDataset (to create dataset from adversarial images)
from torch.utils.data import TensorDataset

# Function to test the CNN model on FGSM-attacked images
def test_model_on_attacked(model, attacked_loader, num_samples=100):
    """
    Evaluates the CNN model on adversarial (FGSM-attacked) images.

    Parameters:
        model (nn.Module): The trained CNN model.
        attacked_loader (DataLoader): DataLoader for the adversarial dataset.
        num_samples (int): Number of samples to print with detailed output.

    Prints:
        - Sample predictions (up to `num_samples`).
        - Accuracy on first `num_samples` images.
        - Overall accuracy on the entire attacked dataset.
    """
    model.eval()  # Set model to evaluation mode (no gradient updates)
    correct = 0  # Track correct predictions
    total = 0  # Track total images
    detailed_sample_count = 0  # Track number of printed samples
```

```python
    with torch.no_grad():  # Disable gradient calculations for efficiency
        for data, target in attacked_loader:
            data, target = data.to(device), target.to(device)  # Move data to GPU/CPU
            output = model(data)  # Forward pass
            pred = output.argmax(dim=1, keepdim=True)  # Get predicted class
            correct_preds = pred.eq(target.view_as(pred))  # Check correct predictions

            # Print sample predictions (up to `num_samples`)
            for idx in range(data.size(0)):
                if detailed_sample_count < num_samples:
                    print(f"Sample {detailed_sample_count + 1}:")
                    print(f"  True label: {target[idx].item()}")
                    print(f"  Predicted raw output: {output[idx]}")  # Print logits
                    print(f"  Predicted label: {pred[idx].item()}")
                    detailed_sample_count += 1  # Increment sample counter

                correct += correct_preds[idx].item()  # Count correct predictions
                total += 1  # Increment total images processed

    # Compute accuracy
    accuracy = correct / total

    # Print results
    print(f"\nAccuracy on first {num_samples} attacked images: {accuracy:.2f}")
    print(f"Overall accuracy on attacked dataset: {correct / total:.2f}")
```

```python
# Convert FGSM-attacked images into a dataset
attacked_data = torch.stack(attacked_images)  # Convert list of tensors into a batch
attacked_targets = torch.tensor(test_dataset.targets[:len(attacked_images)])  # Get corresponding labels
attacked_dataset = TensorDataset(attacked_data, attacked_targets)  # Create dataset

# Create DataLoader for adversarial dataset
attacked_loader = DataLoader(attacked_dataset, batch_size=BATCH_SIZE, shuffle=False)

# Evaluate model on FGSM-attacked images
test_model_on_attacked(model, attacked_loader, num_samples=100)
```

```
[11]  True label: 1
      Predicted raw output: tensor([-2.6330,  2.6985,  0.5222, -0.8751, -2.1388, -0.4871, -1.6659,  0.2540,
              1.9714, -1.6462], device='cuda:0')
      Predicted label: 1
    Sample 91:
      True label: 3
      Predicted raw output: tensor([-3.2800, -1.1516, -0.9978,  4.7035, -3.7512,  1.2308, -4.3705,  0.1955,
              0.2190, -0.3050], device='cuda:0')
      Predicted label: 3
    Sample 92:
      True label: 6
      Predicted raw output: tensor([-3.8525, -4.9968, -7.5822, -3.6130, -4.7130,  6.7501,  5.7607, -9.1084,
             -1.1289, -2.6118], device='cuda:0')
      Predicted label: 5
    Sample 93:
      True label: 9
      Predicted raw output: tensor([-4.1791, -3.3339, -2.9708, -0.9500,  3.2858, -0.7782, -4.2782, -1.0848,
              1.4808,  1.9368], device='cuda:0')
      Predicted label: 4
    Sample 94:
      True label: 3
      Predicted raw output: tensor([-3.7615,  1.2901, -3.0206,  2.9499, -3.9625,  2.7794, -5.2704,  1.2358,
             -3.2276, -0.8870], device='cuda:0')
      Predicted label: 3
```

```
Sample 99:
  True label: 6
  Predicted raw output: tensor([ 2.5188, -3.3020, -3.9545, -4.4294, -3.9291,  2.2701,  5.9133, -7.8701,
          2.1447, -3.3586], device='cuda:0')
  Predicted label: 6
Sample 100:
  True label: 9
  Predicted raw output: tensor([-3.5929, -8.0505, -2.8235, -3.5802,  2.9081, -1.8062, -8.8202, -1.9363,
         -0.6200,  9.8809], device='cuda:0')
  Predicted label: 9

Accuracy on first 100 attacked images: 0.82
Overall accuracy on attacked dataset: 0.82
```

Here, you can see the overall accuracy is reduced to 82% after adding FGSM noise.

Task 04

Code

Task 4: Implement Countermeasure

Approach 1: Adversarial Training

Step 1: Import Required Libraries

```
[12] # Import necessary libraries
     import torch
     import torch.nn as nn
     import torch.optim as optim
     import torch.nn.functional as F
     from torch.utils.data import DataLoader, TensorDataset, random_split
     from torchvision import datasets, transforms
     import matplotlib.pyplot as plt
```

Step 2: Define FGSM Attack Function

```python
[13] # Define FGSM attack function
def fgsm_attack(image, epsilon, data_grad):
    """
    Generates an adversarial example using FGSM.

    Parameters:
        image (Tensor): The input image.
        epsilon (float): The attack strength.
        data_grad (Tensor): The gradient of loss w.r.t. input image.

    Returns:
        Tensor: The adversarial image.
    """
    perturbed_image = image + epsilon * data_grad.sign()  # Apply perturbation
    return torch.clamp(perturbed_image, 0, 1)  # Ensure values stay within valid range
```

Step 3: Generate Adversarial Training Data

```python
[14] # Function to create an adversarial dataset for training
def generate_adversarial_training_data(model, train_loader, epsilon=0.25):
    """
    Creates an adversarial training dataset with 70% clean and 30% FGSM-attacked images.

    Parameters:
        model (nn.Module): The trained CNN model.
        train_loader (DataLoader): DataLoader for the training dataset.
        epsilon (float): Strength of FGSM attack.

    Returns:
        TensorDataset: New dataset with both clean and adversarial examples.
    """
    model.eval()  # Set model to evaluation mode
    clean_data, clean_labels, adv_data, adv_labels = [], [], [], []

    for data, target in train_loader:
        data, target = data.to(device), target.to(device)
        clean_data.extend(data)  # Store original images
        clean_labels.extend(target)
```

```python
        # Generate adversarial examples
        data.requires_grad = True  # Enable gradient tracking
        output = model(data)
        loss = F.cross_entropy(output, target)
        model.zero_grad()
        loss.backward(retain_graph=True)  # Compute gradients w.r.t. input image, preserving the graph
        data_grad = data.grad.data
        perturbed_data = fgsm_attack(data, epsilon, data_grad)
        perturbed_data = perturbed_data.detach()  # Detach to free computation graph

        adv_data.extend(perturbed_data)  # Store adversarial images
        adv_labels.extend(target)

    # Mix clean (70%) and adversarial (30%) data
    num_clean = int(0.7 * len(clean_data))
    num_adv = int(0.3 * len(adv_data))
    mixed_data = clean_data[:num_clean] + adv_data[:num_adv]
    mixed_labels = clean_labels[:num_clean] + adv_labels[:num_adv]

    return TensorDataset(torch.stack(mixed_data), torch.tensor(mixed_labels))
```

Step 4: Load and Prepare Training Dataset

[15]
```
# Load clean MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transforms.ToTensor(), download=True)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Generate adversarial training dataset
adv_train_dataset = generate_adversarial_training_data(model, train_loader)
adv_train_loader = DataLoader(adv_train_dataset, batch_size=64, shuffle=True)
```

Step 5: Train the CNN using Adversarial Training

[16]
```
# Train the CNN with adversarial data
def train_adversarial_model(model, train_loader, val_loader, optimizer, num_epochs=10):
    """
    Trains a CNN model using adversarial training.

    Parameters:
        model (nn.Module): CNN model.
        train_loader (DataLoader): Adversarial training dataset.
        val_loader (DataLoader): Validation dataset.
        optimizer (torch.optim): Optimizer for training.
        num_epochs (int): Number of training epochs.
    """
    model.train()
    for epoch in range(num_epochs):
        train_loss = 0.0
        for data, target in train_loader:
            data, target = data.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(data)
            loss = F.cross_entropy(output, target)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        print(f"Epoch {epoch+1}/{num_epochs}, Training Loss: {train_loss / len(train_loader):.4f}")
```

Step 6: Create Model Instance & Train

[17]
```
# Create a new model instance and optimizer
adv_model = CNN().to(device)
optimizer = optim.Adam(adv_model.parameters(), lr=0.001)

# Train the model using adversarial training
train_adversarial_model(adv_model, adv_train_loader, val_loader, optimizer)
```

```
Epoch 1/10, Training Loss: 0.2954
Epoch 2/10, Training Loss: 0.0849
Epoch 3/10, Training Loss: 0.0552
Epoch 4/10, Training Loss: 0.0401
Epoch 5/10, Training Loss: 0.0320
Epoch 6/10, Training Loss: 0.0282
Epoch 7/10, Training Loss: 0.0222
Epoch 8/10, Training Loss: 0.0187
Epoch 9/10, Training Loss: 0.0172
Epoch 10/10, Training Loss: 0.0144
```

Step 7: Evaluate the Model on FGSM-Attacked Test Data

```
[18] # Evaluate the adversarially trained model on FGSM-attacked test images
     test_model_on_attacked(adv_model, attacked_loader, num_samples=100)
```

```
Sample 90:
  True label: 1
  Predicted raw output: tensor([-4.8833,  7.1421, -2.2799, -1.6620, -4.0859, -2.5126, -1.3613,  0.2485,
        -1.0304, -3.9015], device='cuda:0')
  Predicted label: 1
Sample 91:
  True label: 3
  Predicted raw output: tensor([ -9.9706,  -4.3112,  -6.3949,  12.8470, -12.0261,   1.0527, -13.3311,
         -2.9866,  -1.2844,  -2.0471], device='cuda:0')
  Predicted label: 3
Sample 92:
  True label: 6
  Predicted raw output: tensor([ -0.5159,  -7.6570, -12.6306, -11.2620,  -6.8326,   2.1091,  17.8358,
        -14.9577,  -5.1734,  -4.3013], device='cuda:0')
  Predicted label: 6
```

```
Sample 99:
  True label: 6
  Predicted raw output: tensor([ -0.2349,  -7.6284,  -9.4580,  -9.6908,  -3.4577,   3.0798,  11.5083,
        -10.1360,  -3.7428,  -2.0099], device='cuda:0')
  Predicted label: 6
Sample 100:
  True label: 9
  Predicted raw output: tensor([-11.0789, -14.3334,  -9.2240,  -3.3958,   2.9509,  -5.6411, -20.8849,
         -1.9024,  -1.1758,  19.3513], device='cuda:0')
  Predicted label: 9

Accuracy on first 100 attacked images: 0.95
Overall accuracy on attacked dataset: 0.95
```

Here, the accuracy of our model is 95%.

Task 05

Code

Approach 2: Variational Auto Encoder (VAE) for Data Reconstruction

Step 1: Define the VAE Model

```
[19] # Define Variational Autoencoder (VAE)
     class VAE(nn.Module):
         def __init__(self):
             super(VAE, self).__init__()

             # Encoder: Compress input into a lower-dimensional latent space
             self.encoder = nn.Sequential(
                 nn.Linear(28 * 28, 400),  # Input: 784 pixels (28x28 flattened) → Hidden layer: 400 neurons
                 nn.ReLU(),  # Activation function (introduces non-linearity)
                 nn.Linear(400, 20)  # Output layer (20 neurons: first 10 for mean, next 10 for variance)
             )

             # Decoder: Reconstruct input from the latent representation
             self.decoder = nn.Sequential(
                 nn.Linear(10, 400),  # Input: 10 latent variables → Hidden layer: 400 neurons
                 nn.ReLU(),  # Activation function
                 nn.Linear(400, 28 * 28),  # Output layer: reconstruct 784 pixels (28x28 image)
                 nn.Sigmoid()  # Apply sigmoid to keep values between [0,1] (valid pixel range)
             )
```

```python
    def reparameterize(self, mu, log_var):
        """
        Reparameterization trick: Convert mean & variance into a sample from a Gaussian distribution.

        Parameters:
            mu (Tensor): Mean of the latent distribution.
            log_var (Tensor): Log variance of the latent distribution.

        Returns:
            Tensor: Sample from the latent space.
        """
        std = torch.exp(0.5 * log_var)  # Convert log variance to standard deviation
        eps = torch.randn_like(std)  # Generate random noise with same shape as std
        return mu + eps * std  # Compute the latent variable using reparameterization trick
```

```python
    def forward(self, x):
        """
        Forward pass through VAE.

        Parameters:
            x (Tensor): Input image (flattened).

        Returns:
            Tensor: Reconstructed image.
        """
        x = x.view(-1, 28 * 28)  # Flatten input image (28x28 → 784 pixels)
        mu_logvar = self.encoder(x).chunk(2, dim=1)  # Split output into mean & variance (first 10 neurons = mu, last 10 = log_var)
        z = self.reparameterize(*mu_logvar)  # Sample from latent space using mean and variance
        x_reconstructed = self.decoder(z)  # Decode back to image format
        return x_reconstructed.view(-1, 1, 28, 28)  # Reshape output to image format (batch_size, 1 channel, 28, 28)
```

Step 2: Train the VAE

[20]

```python
# Initialize VAE model and optimizer
vae = VAE().to(device)  # Create a VAE instance and move it to GPU if available
vae_optimizer = optim.Adam(vae.parameters(), lr=0.001)  # Use Adam optimizer with learning rate 0.001

def train_vae(vae, train_loader, optimizer, num_epochs=10):
    """
    Trains the VAE on clean MNIST images.

    Parameters:
        vae (nn.Module): Variational Autoencoder model.
        train_loader (DataLoader): DataLoader for the training dataset.
        optimizer (torch.optim): Optimizer for training.
        num_epochs (int): Number of training epochs.
    """
    vae.train()  # Set model to training mode
    for epoch in range(num_epochs):
        total_loss = 0.0  # Initialize total loss for the epoch

        for data, _ in train_loader:  # No need for labels (unsupervised learning)
            data = data.to(device)  # Move images to GPU if available
            optimizer.zero_grad()  # Reset gradients
```

```python
            reconstructed_data = vae(data)  # Forward pass: reconstruct input images
            loss = F.mse_loss(reconstructed_data, data)  # Compute Mean Squared Error (MSE) loss
            loss.backward()  # Backpropagation
            optimizer.step()  # Update model weights

            total_loss += loss.item()  # Accumulate total loss

        # Print average loss for this epoch
        print(f"Epoch {epoch+1}/{num_epochs}, VAE Loss: {total_loss / len(train_loader):.4f}")

# Train the VAE for 10 epochs
train_vae(vae, train_loader, vae_optimizer)
```

[20]

```
Epoch 1/10, VAE Loss: 0.0312
Epoch 2/10, VAE Loss: 0.0198
Epoch 3/10, VAE Loss: 0.0179
Epoch 4/10, VAE Loss: 0.0168
Epoch 5/10, VAE Loss: 0.0161
Epoch 6/10, VAE Loss: 0.0156
Epoch 7/10, VAE Loss: 0.0152
Epoch 8/10, VAE Loss: 0.0149
Epoch 9/10, VAE Loss: 0.0147
Epoch 10/10, VAE Loss: 0.0145
```

Step 3: Reconstruct FGSM-Attacked Images using the Trained VAE

[21]

```python
# Function to reconstruct attacked images using VAE
def reconstruct_images(vae, attacked_loader):
    """
    Passes FGSM-attacked images through VAE for denoising.

    Parameters:
        vae (nn.Module): Trained Variational Autoencoder.
        attacked_loader (DataLoader): DataLoader for FGSM-attacked images.

    Returns:
        list: Reconstructed images.
    """
    vae.eval()  # Set VAE to evaluation mode
    reconstructed_images = []
```

[21]

```python
    with torch.no_grad():  # No gradient tracking for inference
        for data, _ in attacked_loader:
            data = data.to(device)  # Move to GPU if available
            recon_data = vae(data)  # Forward pass through VAE
            reconstructed_images.extend(recon_data.cpu())  # Convert to CPU and store

    return reconstructed_images

# Reconstruct FGSM-attacked images
reconstructed_images = reconstruct_images(vae, attacked_loader)
```

Step 4: Evaluate CNN Model on Reconstructed Data

[22]

```python
# Evaluate the CNN on reconstructed images
reconstructed_data = torch.stack(reconstructed_images)
reconstructed_dataset = TensorDataset(reconstructed_data, attacked_targets)
reconstructed_loader = DataLoader(reconstructed_dataset, batch_size=BATCH_SIZE, shuffle=False)

# Test CNN on reconstructed data
test_model_on_attacked(model, reconstructed_loader, num_samples=100)
```

```
Sample 1:
  True label: 7
  Predicted raw output: tensor([-1.5733,  0.4008,  0.2058,  1.2772, -0.8958, -1.4220, -5.3713,  5.0196,
        -1.8843,  0.1853], device='cuda:0')
  Predicted label: 7
Sample 2:
  True label: 2
  Predicted raw output: tensor([-0.2153,  0.2927,  9.2639, -2.8586, -0.7363, -3.8592, -2.4503, -0.3820,
        -1.9230, -3.8446], device='cuda:0')
  Predicted label: 2
Sample 3:
  True label: 1
  Predicted raw output: tensor([-1.1469,  1.2435,  0.3843, -1.0774, -0.4821, -0.7330, -0.8952,  0.4698,
         1.0509, -0.6012], device='cuda:0')
  Predicted label: 1
```

```
Sample 99:
  True label: 6
  Predicted raw output: tensor([ 0.5710, -2.3261, -3.4726, -3.4457, -1.6817,  1.6779,  3.3572, -5.1982,
         0.1781, -1.0365], device='cuda:0')
  Predicted label: 6
Sample 100:
  True label: 9
  Predicted raw output: tensor([-2.8969, -5.0421, -2.4995, -1.8890,  1.5840, -1.5479, -8.9793,  0.6748,
        -1.3847,  8.1968], device='cuda:0')
  Predicted label: 9

Accuracy on first 100 attacked images: 0.87
Overall accuracy on attacked dataset: 0.87
```

The achieved accuracy is 87%

Task 06 (Bonus)

Code

Approach 3: GAN Based Anomaly Detection for FGSM Noise

Step 1: Load and Preprocess Clean MNIST Data

```
# Load the clean MNIST dataset for training GAN
transform = transforms.Compose([
    transforms.ToTensor(),  # Convert image to tensor
    transforms.Normalize((0.5,), (0.5,))  # Normalize pixel values
])

# Load only the CLEAN MNIST training dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
```

Step 2: Define GAN Model

```
[25] # Define the Generator Model
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(100, 256),  # Input: 100-dimensional noise
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 28 * 28),  # Output: 28x28 image
            nn.Tanh()  # Output values in range [-1,1]
        )

    def forward(self, z):
        return self.model(z).view(-1, 1, 28, 28)  # Reshape output to image format
```

```python
# Define the Discriminator Model
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(28 * 28, 512),  # Input: 28x28 image flattened
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 1),  # Output: Probability of being real
            nn.Sigmoid()  # Output value in range [0,1]
        )

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # Flatten input
        return self.model(x)
```

Step 3: Train the GAN on MNIST Data

[26]

```python
# Initialize models and optimizers
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

generator = Generator().to(device)
discriminator = Discriminator().to(device)

optimizer_G = optim.Adam(generator.parameters(), lr=0.0002)
optimizer_D = optim.Adam(discriminator.parameters(), lr=0.0002)

loss_function = nn.BCELoss()  # Binary Cross-Entropy Loss for classification

# Training loop for GAN
num_epochs = 10

for epoch in range(num_epochs):
    for real_images, _ in train_loader:
        real_images = real_images.to(device)
        batch_size = real_images.size(0)

        # Create labels for real and fake images
        real_labels = torch.ones(batch_size, 1).to(device)
        fake_labels = torch.zeros(batch_size, 1).to(device)
```

```python
        # Train the Discriminator
        optimizer_D.zero_grad()

        output_real = discriminator(real_images)  # Classify real images
        loss_real = loss_function(output_real, real_labels)

        noise = torch.randn(batch_size, 100).to(device)  # Generate noise
        fake_images = generator(noise)  # Generate fake images
        output_fake = discriminator(fake_images.detach())  # Classify fake images
        loss_fake = loss_function(output_fake, fake_labels)

        loss_D = loss_real + loss_fake  # Total loss for discriminator
        loss_D.backward()
        optimizer_D.step()
```

```
        # Train the Generator
        optimizer_G.zero_grad()

        output_fake = discriminator(fake_images)  # Classify fake images
        loss_G = loss_function(output_fake, real_labels)  # Try to fool the discriminator

        loss_G.backward()
        optimizer_G.step()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss D: {loss_D.item():.4f}, Loss G: {loss_G.item():.4f}")
```

```
Epoch 1/10, Loss D: 0.1122, Loss G: 4.7896
Epoch 2/10, Loss D: 0.1291, Loss G: 4.1214
Epoch 3/10, Loss D: 0.8779, Loss G: 3.0290
Epoch 4/10, Loss D: 0.2379, Loss G: 3.5368
Epoch 5/10, Loss D: 0.7788, Loss G: 2.4702
Epoch 6/10, Loss D: 0.7806, Loss G: 3.1763
Epoch 7/10, Loss D: 0.8716, Loss G: 2.2718
Epoch 8/10, Loss D: 0.7225, Loss G: 1.8449
Epoch 9/10, Loss D: 0.2487, Loss G: 3.0844
Epoch 10/10, Loss D: 0.6848, Loss G: 3.7169
```

Step 4: Use Discriminator to Detect FGSM Attacked Images

```
# Function to detect anomalies using the trained Discriminator
def detect_anomalies(discriminator, attacked_loader):
    """
    Pass FGSM-attacked images through the Discriminator to compute anomaly scores.

    Parameters:
        discriminator (nn.Module): Trained Discriminator.
        attacked_loader (DataLoader): DataLoader containing FGSM-attacked images.

    Returns:
        list: List of anomaly scores (Discriminator output probability).
    """
    discriminator.eval()
    anomaly_scores = []

    with torch.no_grad():
        for data, _ in attacked_loader:
            data = data.to(device)
            output = discriminator(data)  # Get probability of being real
            anomaly_scores.extend(output.cpu().numpy())  # Store anomaly scores

    return anomaly_scores
```

```
# Compute anomaly scores for FGSM-attacked images
anomaly_scores = detect_anomalies(discriminator, attacked_loader)

# Display sample anomaly scores
print("Sample Anomaly Scores (Discriminator Output Probability):")
print(anomaly_scores[:10])
```

```
Sample Anomaly Scores (Discriminator Output Probability):
[array([6.343477e-08], dtype=float32), array([2.0826292e-07], dtype=float32), array([0.00048917], dtype=float32), array([2.0229265e-09], dtype=float32),
```

Step 5: Use Anomaly Scores to Improve Model's Robustness

[28]
```python
# Define a threshold for anomaly detection
threshold = 0.5  # If Discriminator probability < 0.5, mark as adversarial

# Count detected anomalies
num_anomalies = sum(1 for score in anomaly_scores if score < threshold)
total_images = len(anomaly_scores)

print(f"Detected {num_anomalies}/{total_images} adversarial images ({(num_anomalies/total_images)*100:.2f}%)")
```

```
Detected 10000/10000 adversarial images (100.00%)
```

Step 6: Compute Detection Accuracy

[29]
```python
# Function to compute detection accuracy of the Discriminator
def compute_detection_accuracy(discriminator, attacked_loader, threshold=0.5):
    """
    Evaluates how well the Discriminator detects FGSM-attacked images as anomalies.

    Parameters:
        discriminator (nn.Module): Trained Discriminator.
        attacked_loader (DataLoader): DataLoader containing FGSM-attacked images.
        threshold (float): Decision threshold (default = 0.5).

    Returns:
        float: Detection accuracy (percentage of FGSM images detected as anomalies).
    """
    discriminator.eval()  # Set Discriminator to evaluation mode
    correct_detections = 0  # Track correctly classified anomalies
    total_samples = 0  # Track total adversarial samples

    with torch.no_grad():  # No gradient tracking needed
        for data, _ in attacked_loader:
            data = data.to(device)
            output = discriminator(data)  # Get probability of being real
            predictions = (output < threshold).float()  # Mark as anomaly if probability < threshold
            correct_detections += predictions.sum().item()  # Count correctly classified anomalies
            total_samples += data.size(0)  # Update total sample count

    # Compute accuracy
    detection_accuracy = (correct_detections / total_samples) * 100  # Convert to percentage
    return detection_accuracy

# Compute and print the detection accuracy
detection_accuracy = compute_detection_accuracy(discriminator, attacked_loader)
print(f"Discriminator Detection Accuracy on FGSM-Attacked Images: {detection_accuracy:.2f}%")
```

```
Discriminator Detection Accuracy on FGSM-Attacked Images: 100.00%
```

THE END