Lab_2

December 12, 2024

1 1. Single Perceptron

[1]:
```
import torch
import random
import matplotlib.pyplot as plt

# Check if CUDA is available, else use CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

```
Using device: cuda
```

[2]:
```
def generate_one_like_pattern():
    # Start with random 0/1 pattern
    pattern = (torch.rand(25) > 0.3).float()  # About 70% chance to be 1
    # (Optional) enforce a vertical line in the center column:
    # index of center column pixels in a flattened 5x5: these indices are␣
↪2,7,12,17,22
    center_col_indices = [2,7,12,17,22]
    pattern[center_col_indices] = 1.0
    return pattern

# Function to create a random pattern that resembles "0"
def generate_zero_like_pattern():
    # Generate a pattern with a border of 1s and random inside
    pattern = torch.zeros(5,5)
    # Set border to 1
    pattern[0,:] = 1
    pattern[-1,:] = 1
    pattern[:,0] = 1
    pattern[:,-1] = 1
    # Inside random 0/1 with lower probability of 1
    inside = (torch.rand(3,3) > 0.8).float()  # About 20% chance to be 1 inside
    pattern[1:4,1:4] = inside
    return pattern.flatten()

def run_simulation(ones, zeros, alpha=0.1, max_epochs=1000):
    # Randomly select training and test patterns
```

```
    one_indices = torch.randperm(len(ones))
    zero_indices = torch.randperm(len(zeros))

    train_ones = [ones[i] for i in one_indices[:4]]
    test_ones = [ones[i] for i in one_indices[4:]]
    train_zeros = [zeros[i] for i in zero_indices[:4]]
    test_zeros = [zeros[i] for i in zero_indices[4:]]

    X_train = torch.stack(train_ones + train_zeros)
    d_train = torch.tensor([1]*4 + [-1]*4, dtype=torch.float32)
    X_test = torch.stack(test_ones + test_zeros)
    d_test = torch.tensor([1]*len(test_ones) + [-1]*len(test_zeros),␣
↪dtype=torch.float32)

    # Initialize weights and bias
    w = torch.randn(25, dtype=torch.float32) * 0.01
    b = torch.randn(1, dtype=torch.float32) * 0.01

    for epoch in range(max_epochs):
        total_errors = 0
        indices = torch.randperm(len(X_train))
        for i in indices:
            x = X_train[i]
            d = d_train[i].item()

            z = torch.dot(w, x) + b
            y = 1.0 if z.item() >= 0 else -1.0

            error = d - y
            if error != 0:
                w = w + alpha * error * x
                b = b + alpha * error
                total_errors += 1

        if total_errors == 0:
            # Training converged
            break

    # Test the trained perceptron
    correct = 0
    for i, x in enumerate(X_test):
        z = torch.dot(w, x) + b
        y = 1.0 if z.item() >= 0 else -1.0
        if y == d_test[i].item():
            correct += 1

    accuracy = correct / len(X_test) * 100
```

```
    return accuracy
```

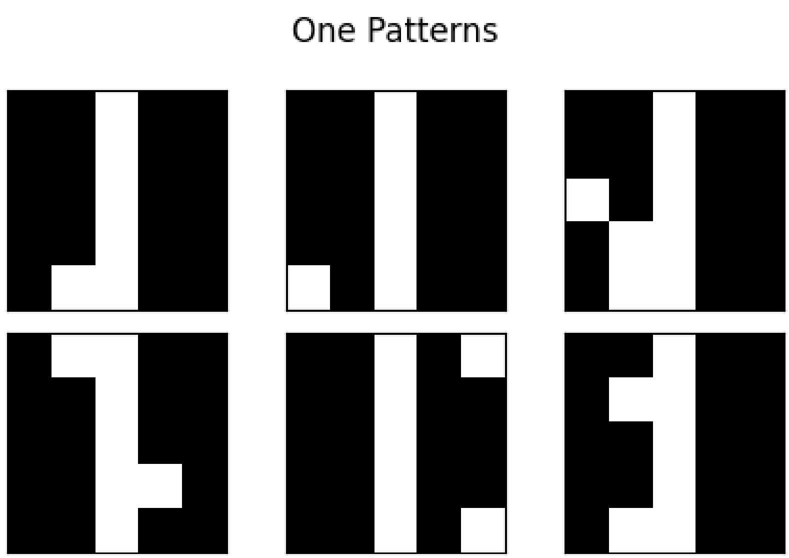
[3]:
```
def generate_one_like_pattern():
    # Create a 5x5 array with mostly zeros
    pattern = torch.zeros(5,5)
    # Make a vertical line down the center column
    # Center column is column index 2 (0-based)
    pattern[:,2] = 1

    # Optionally, add some random noise: randomly flip some 0s to 1s
    # to create variation between patterns
    noise = (torch.rand(5,5) > 0.9).float()  # ~10% chance to flip
    pattern = torch.clamp(pattern + noise, 0, 1)

    return pattern.flatten().float()

def generate_zero_like_pattern():
    # Create a 5x5 array with a border of 1's and inside 0's
    pattern = torch.zeros(5,5)
    pattern[0,:] = 1
    pattern[-1,:] = 1
    pattern[:,0] = 1
    pattern[:,-1] = 1

    # Add some noise inside to vary the patterns
    # The inside is a 3x3 area at indices [1:4,1:4]
    inside_noise = (torch.rand(3,3) > 0.8).float()  # ~20% chance for inside␣
↪pixels to be 1
    pattern[1:4,1:4] = inside_noise

    return pattern.flatten().float()

def show_patterns_grid(patterns, rows=2, cols=3, title="Patterns Grid"):
    fig, axes = plt.subplots(rows, cols, figsize=(cols*1.5, rows*1.5))
    fig.suptitle(title)

    # Flatten axes array if it's 2D for easy iteration
    axes = axes.flatten()

    for ax, pattern in zip(axes, patterns):
        img = pattern.reshape(5,5)
        ax.imshow(img, cmap='gray', interpolation='nearest')
        ax.set_xticks([])
        ax.set_yticks([])

    # If there are more axes than patterns, turn off the extra ones
    for ax in axes[len(patterns):]:
```

```
        ax.axis('off')

    plt.tight_layout()
    # Adjust spacing so that the main title doesn't overlap
    plt.subplots_adjust(top=0.85)
    plt.show()
```

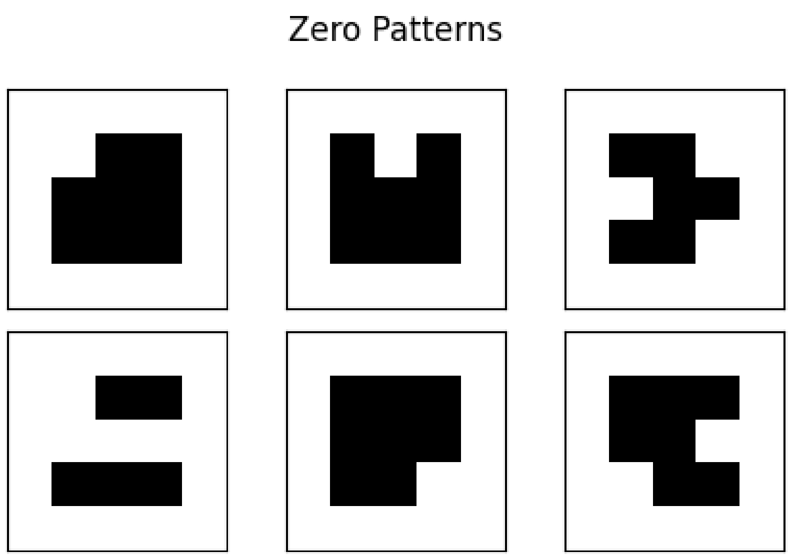
[4]:

```
# Generate lists of patterns for "1" and "0"
ones_list = [generate_one_like_pattern() for _ in range(6)]
zeros_list = [generate_zero_like_pattern() for _ in range(6)]

# Print out the generated patterns
print("Ones patterns:")
show_patterns_grid(ones_list, rows=2, cols=3, title="One Patterns")

print("\nZeros patterns:")
show_patterns_grid(zeros_list, rows=2, cols=3, title="Zero Patterns")
```

```
Ones patterns:
```



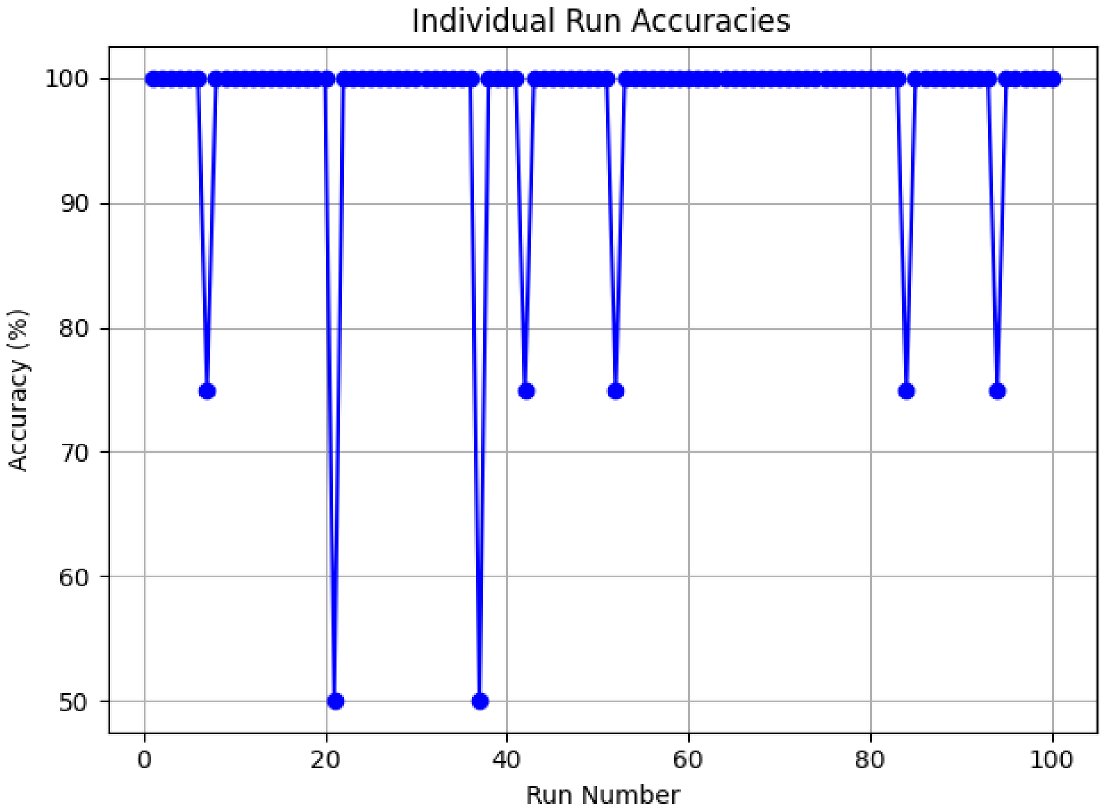

```
Zeros patterns:
```



[5]:
```
# Run multiple simulations and record performance
n_runs = 100
results = []
for run_id in range(n_runs):
    acc = run_simulation(ones_list, zeros_list, alpha=0.1, max_epochs=1000)
    results.append(acc)
```

[6]:
```
avg_accuracy = sum(results) / len(results)
print(f"Average accuracy over {n_runs} runs: {avg_accuracy:.2f}%")
fig, ax = plt.subplots()

# You can plot a simple line plot of the results:
ax.plot(range(1, len(results)+1), results, marker='o', color='b')

# Alternatively, for a bar chart, you could do:
# ax.bar(range(1, len(results)+1), results, color='skyblue')

# Label axes and title
ax.set_xlabel('Run Number')
ax.set_ylabel('Accuracy (%)')
ax.set_title('Individual Run Accuracies')

# Optionally add grid and tight layout
ax.grid(True)
plt.tight_layout()

# Show the plot
plt.show()
```

```
Average accuracy over 100 runs: 97.75%
```



- **Average Performance**: When running the simulation multiple times (e.g., 10 runs), each with different random initial weights and random training/testing splits, the average accuracy provides a more stable measure of the perceptron's true performance. This average smooths out the effects of any single "lucky" or "unlucky" run.
- **Variability in Results**: Individual runs may show significant variation. Some runs might achieve high accuracy if initial conditions and chosen training patterns are favorable, while others may perform poorly.
- **Influence of Training Patterns & Parameters**: The selection of training patterns, learning rate, and number of epochs can all affect variability. More representative training sets, well-tuned learning rates, or longer training can reduce variability and increase the average accuracy.

Takeaway: Multiple runs highlight that performance is not deterministic. Reporting the mean accuracy (and possibly standard deviation) over several runs gives a more reliable assessment of the perceptron's generalization ability.

2 2. Single Layer Perceptrons

[7]:
```
import torch.nn.functional as F
from scipy.io import loadmat
```

[8]:
```
def one_hot(labels, num_classes=10):
    return F.one_hot(labels, num_classes=num_classes).float()

def forward(X, w, b):
    v = torch.matmul(X, w) + b
    y = 1 / (1 + torch.exp(-v))  # Sigmoid
    return y

def train_with_criteria(X, Y, w, b, alpha=0.1, max_epochs=10,␣
↪target_train_error=0.0):

    Y_onehot = one_hot(Y, num_classes=10)
    N = X.shape[0]
    losses = []
    train_accuracies = []

    for epoch in range(max_epochs):
        y = forward(X, w, b)
        error = Y_onehot - y
        delta = error * y * (1 - y)

        grad_w = torch.matmul(X.T, delta) / N
        grad_b = delta.mean(dim=0)

        w += alpha * grad_w
        b += alpha * grad_b

        # Compute mean squared error loss
        loss = (error**2).mean().item()
        losses.append(loss)

        # Compute training accuracy
        train_acc = evaluate(X, Y, w, b)
        train_accuracies.append(train_acc)

        # print(f"Epoch {epoch+1}/{max_epochs}, Loss: {loss:.4f}, Training␣
↪Accuracy: {train_acc:.2f}%")

        # Check early stopping criterion
        train_error_rate = 100 - train_acc
        if train_error_rate <= target_train_error:
```

```
            print(f"Stopping early as training error reached {train_error_rate:.
↪2f}%")
            break

    return w, b, losses, train_accuracies

def evaluate(X, Y, w, b):
    y = forward(X, w, b)
    preds = y.argmax(dim=1)
    correct = (preds == Y).sum().item()
    accuracy = correct / X.shape[0] * 100
    return accuracy
```

[9]:
```
data = loadmat('mnist.mat')
trainX = torch.tensor(data['trainX'], dtype=torch.float32)  # shape (60000,784)
trainY = torch.tensor(data['trainY'].flatten(), dtype=torch.long) # shape␣
↪(60000,)
testX = torch.tensor(data['testX'], dtype=torch.float32)   # shape (10000,784)
testY = torch.tensor(data['testY'].flatten(), dtype=torch.long)  # shape␣
↪(10000,)
```

[10]:
```
# Normalize the input images
trainX /= 255.0
testX  /= 255.0

# Initialize weights and biases
w = torch.randn(784, 10)*0.01
b = torch.randn(10)*0.01

base_lr = 0.1
alpha = 0.01

max_epochs = 100
target_train_error = 0
```

[11]:
```
learning_rate_s = []
final_train_acc_s = []
final_test_acc_s = []
loss_dict = {}
acc_dict = {}

for i in range(25+1):
    lr = base_lr + alpha * i

    w, b, losses, train_accuracies = train_with_criteria(trainX, trainY, w, b,␣
↪alpha=lr, max_epochs=max_epochs, target_train_error=target_train_error)
```

```python
    loss_dict[lr]=losses
    acc_dict[lr]=train_accuracies

    # Evaluate on training and test sets
    final_train_acc = evaluate(trainX, trainY, w, b)
    final_test_acc  = evaluate(testX, testY, w, b)
    learning_rate_s.append(lr)
    final_train_acc_s.append(final_train_acc)
    final_test_acc_s.append(final_test_acc)

    print(f"Learning Rate: {lr:.4f}")
    print(f"Final Training Accuracy: {final_train_acc:.2f}%")
    print(f"Test Accuracy: {final_test_acc:.2f}%")
    print("------------------------------------")
    # # Plot losses
    # plt.figure()
    # plt.plot(losses, marker='o')
    # plt.title("Training Loss Over Epochs")
    # plt.xlabel("Epoch")
    # plt.ylabel("MSE Loss")
    # plt.grid(True)
    # plt.show()
    #
    # # Plot training accuracies
    # plt.figure()
    # plt.plot(train_accuracies, marker='o')
    # plt.title("Training Accuracy Over Epochs")
    # plt.xlabel("Epoch")
    # plt.ylabel("Accuracy (%)")
    # plt.grid(True)
    # plt.show()
```

```
Learning Rate: 0.1000
Final Training Accuracy: 68.73%
Test Accuracy: 69.52%
------------------------------------
Learning Rate: 0.1100
Final Training Accuracy: 80.36%
Test Accuracy: 81.45%
------------------------------------
Learning Rate: 0.1200
Final Training Accuracy: 83.29%
Test Accuracy: 84.21%
------------------------------------
Learning Rate: 0.1300
Final Training Accuracy: 84.52%
Test Accuracy: 85.15%
------------------------------------
```

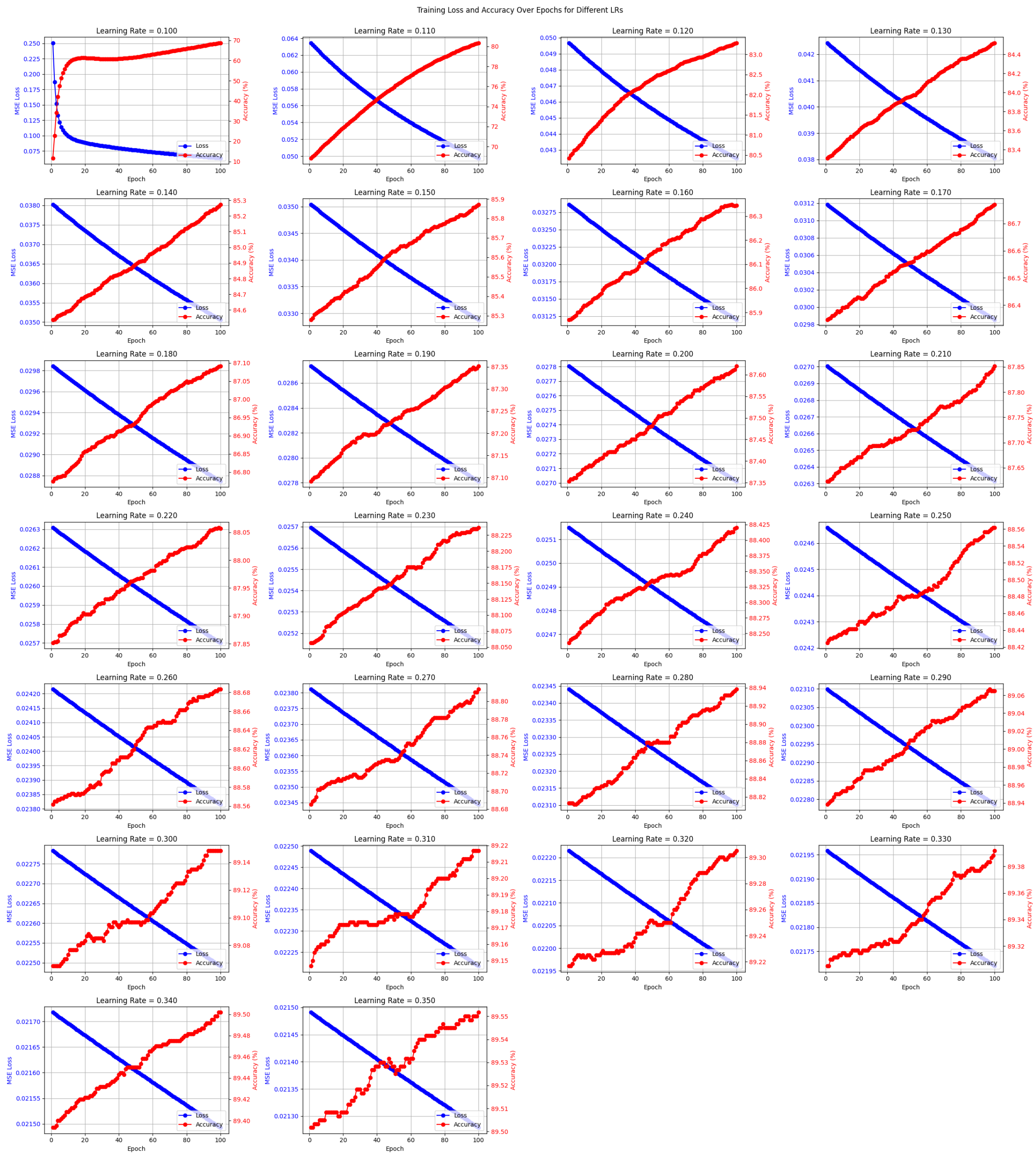
```
Learning Rate: 0.1400
Final Training Accuracy: 85.27%
Test Accuracy: 86.17%
------------------------------------
Learning Rate: 0.1500
Final Training Accuracy: 85.87%
Test Accuracy: 86.87%
------------------------------------
Learning Rate: 0.1600
Final Training Accuracy: 86.34%
Test Accuracy: 87.35%
------------------------------------
Learning Rate: 0.1700
Final Training Accuracy: 86.77%
Test Accuracy: 87.66%
------------------------------------
Learning Rate: 0.1800
Final Training Accuracy: 87.09%
Test Accuracy: 87.93%
------------------------------------
Learning Rate: 0.1900
Final Training Accuracy: 87.35%
Test Accuracy: 88.29%
------------------------------------
Learning Rate: 0.2000
Final Training Accuracy: 87.62%
Test Accuracy: 88.49%
------------------------------------
Learning Rate: 0.2100
Final Training Accuracy: 87.85%
Test Accuracy: 88.77%
------------------------------------
Learning Rate: 0.2200
Final Training Accuracy: 88.06%
Test Accuracy: 89.02%
------------------------------------
Learning Rate: 0.2300
Final Training Accuracy: 88.24%
Test Accuracy: 89.19%
------------------------------------
Learning Rate: 0.2400
Final Training Accuracy: 88.42%
Test Accuracy: 89.36%
------------------------------------
Learning Rate: 0.2500
Final Training Accuracy: 88.56%
Test Accuracy: 89.40%
------------------------------------
```

```
Learning Rate: 0.2600
Final Training Accuracy: 88.68%
Test Accuracy: 89.52%
----------------------------------------
Learning Rate: 0.2700
Final Training Accuracy: 88.81%
Test Accuracy: 89.60%
----------------------------------------
Learning Rate: 0.2800
Final Training Accuracy: 88.94%
Test Accuracy: 89.74%
----------------------------------------
Learning Rate: 0.2900
Final Training Accuracy: 89.07%
Test Accuracy: 89.85%
----------------------------------------
Learning Rate: 0.3000
Final Training Accuracy: 89.15%
Test Accuracy: 89.96%
----------------------------------------
Learning Rate: 0.3100
Final Training Accuracy: 89.22%
Test Accuracy: 90.04%
----------------------------------------
Learning Rate: 0.3200
Final Training Accuracy: 89.31%
Test Accuracy: 90.15%
----------------------------------------
Learning Rate: 0.3300
Final Training Accuracy: 89.39%
Test Accuracy: 90.27%
----------------------------------------
Learning Rate: 0.3400
Final Training Accuracy: 89.50%
Test Accuracy: 90.39%
----------------------------------------
Learning Rate: 0.3500
Final Training Accuracy: 89.55%
Test Accuracy: 90.43%
----------------------------------------
```

[29]:

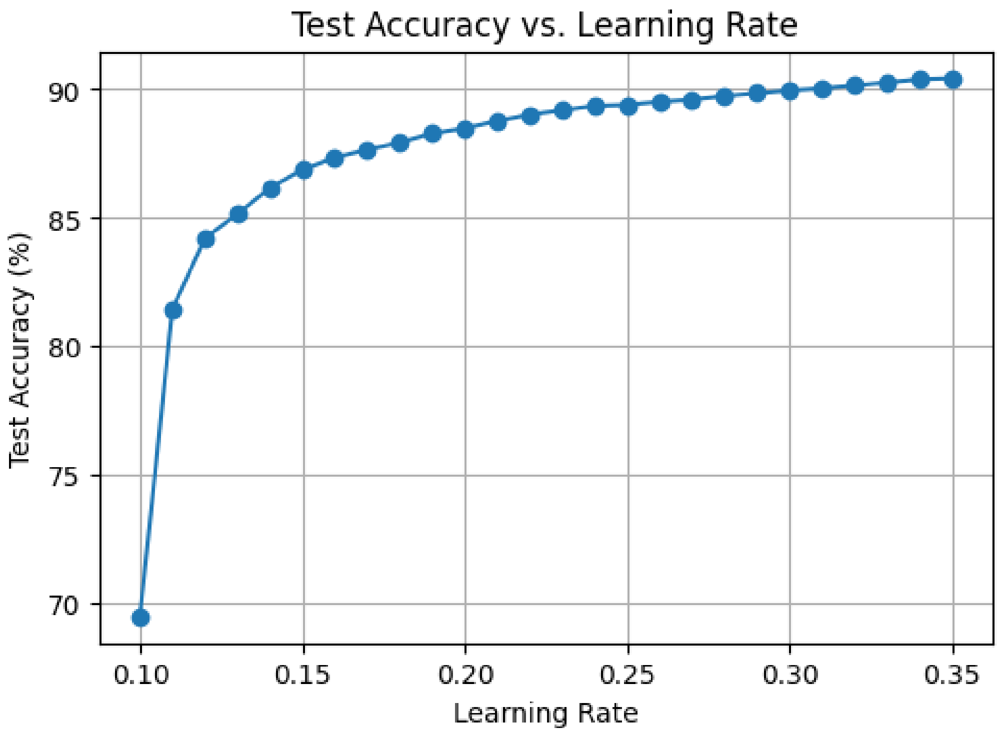
```
epochs = range(1, max_epochs+1)

fig, axes = plt.subplots(nrows=7, ncols=4, figsize=(24,28))
fig.suptitle("Training Loss and Accuracy Over Epochs for Different LRs", y=0.95)

# If there's only one LR, make sure axes is a list
```

```
axes = axes.flatten()
for i, lr in enumerate(learning_rate_s):
    ax1 = axes[i]
    # Plot loss on ax1
    ax1.plot(epochs, loss_dict[lr], marker='o', color='blue', label='Loss')
    ax1.set_title(f"Learning Rate = {lr:.3f}")
    ax1.set_xlabel("Epoch")
    ax1.set_ylabel("MSE Loss", color='blue')
    ax1.tick_params(axis='y', labelcolor='blue')
    ax1.grid(True)

    # Create second axis for accuracy
    ax2 = ax1.twinx()
    ax2.plot(epochs, acc_dict[lr], marker='o', color='red', label='Accuracy')
    ax2.set_ylabel("Accuracy (%)", color='red')
    ax2.tick_params(axis='y', labelcolor='red')

    # Combine legends
    lines1, labels1 = ax1.get_legend_handles_labels()
    lines2, labels2 = ax2.get_legend_handles_labels()
    ax1.legend(lines1 + lines2, labels1 + labels2, loc='lower right')

for j in range(len(learning_rate_s), len(axes)):
    axes[j].axis('off')

plt.tight_layout(rect=[0,0,1,0.95])
plt.show()
```



[15]:

```
plt.figure()
plt.plot(learning_rate_s, final_test_acc_s, marker='o', linestyle='-')
plt.title("Test Accuracy vs. Learning Rate")
plt.xlabel("Learning Rate")
plt.ylabel("Test Accuracy (%)")
plt.grid(True)
plt.show()
```



Performance Report

Learning Rate	Final Training Accuracy	Test Accuracy
0.10	~68.73%	~69.52%
0.20	~87.62%	~88.49%
0.30	~89.15%	~89.96%
0.35	~89.55%	~90.43%

Error Rates: - At LR=0.10, Test Error 30.48%. - At LR=0.35, Test Error 9.57%.

Influence of Learning Rate: - **Low LR (e.g. 0.10)**: Slower convergence, lower final accuracy. - **Moderate to High LR (0.20–0.35)**: Faster and more effective training, leading to significantly higher accuracy.

Increasing the learning rate from 0.10 to around 0.30–0.35 steadily improves both training and test performance, allowing the model to reach near 90% accuracy. However, extremely high learning rates (not shown) could cause instability and hinder convergence.

[ ]: