1. (a) Assume that $z_i = W_1 x_i + b_1$, $a_i = \tanh(z_i)$, $\hat{y}_i = w_2^T a_i + b_2$.
   $\frac{\partial J}{\partial b_2} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_2} = \frac{1}{N} \sum_{i=1}^{N} \text{sign}(\hat{y}_i - y_i)$
   $\frac{\partial J}{\partial w_2} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2} = \frac{1}{N} \sum_{i=1}^{N} \text{sign}(\hat{y}_i - y_i) \cdot a_i = \frac{1}{N} \sum_{i=1}^{N} \text{sign}(\hat{y}_i - y_i) \cdot \tanh(W_1 x_i + b_1)$
   $\frac{\partial J}{\partial b_1} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a} \frac{\partial a}{\partial z} \frac{\partial z}{\partial b_1} = \frac{1}{N} \sum_{i=1}^{N} \text{sign}(\hat{y}_i - y_i) \cdot w_2 \cdot (1 - \tanh^2(z_i)) = \frac{1}{N} \sum_{i=1}^{N} \text{sign}(\hat{y}_i - y_i) \cdot w_2 \cdot (1 - \tanh^2(W_1 x_i + b_1))$
   $\frac{\partial J}{\partial W_1} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a} \frac{\partial a}{\partial z} \frac{\partial z}{\partial W_1} = \frac{1}{N} \sum_{i=1}^{N} \text{sign}(\hat{y}_i - y_i) \cdot w_2 \cdot (1 - \tanh^2(z_i)) x_i^T = \frac{1}{N} \sum_{i=1}^{N} \text{sign}(\hat{y}_i - y_i) \cdot w_2 \cdot (1 - \tanh^2(W_1 x_i + b_1)) x_i^T$
   where $\cdot$ is dot product.

   The code part starts on the next page.

1. (b)

[38]:

```
import torch

# Define network dimensions
input_size = 10
hidden_size = 20
output_size = 1

# Define network parameters
W1 = torch.randn(hidden_size, input_size, requires_grad=True)
b1 = torch.randn(hidden_size, 1, requires_grad=True)
W2 = torch.randn(hidden_size, output_size, requires_grad=True)
b2 = torch.randn(output_size, requires_grad=True)

# Define forward pass function
def forward(x):
    z = torch.matmul(W1, x) + b1
    a = torch.tanh(z)
    y_hat = torch.matmul(W2.T, a) + b2
    return y_hat

# Define loss function
def loss(y_hat, y):
    return torch.mean(torch.abs(y_hat - y))

# Generate random input and output data
x = torch.randn(100, input_size, 1)
y = torch.randn(100, output_size, 1)

# Compute forward pass and loss
y_hat = forward(x)
L = loss(y_hat, y)

# Compute gradients

L.backward()
grad_W1 = W1.grad
grad_b1 = b1.grad
grad_W2 = W2.grad
grad_b2 = b2.grad

# Compute gradients manually
dL_db2 = torch.mean(torch.sign(y_hat - y), dim=0)
dL_dW2 = torch.mean(torch.sign(y_hat - y) * torch.tanh(torch.matmul(W1, x) +␣
 ↪b1), dim=0)
```

```
dL_db1 = torch.mean(torch.sign(y_hat - y) * W2 * (1 - torch.tanh(torch.
 ↪matmul(W1, x) + b1) ** 2), dim=0)
dL_dW1 = torch.mean(torch.matmul(torch.sign(y_hat - y) * W2 * (1 - torch.
 ↪tanh(torch.matmul(W1, x) + b1) ** 2), x.permute(0, 2, 1)), dim=0)


# Compare gradients
print('W1 gradient difference:', torch.linalg.norm(grad_W1 - dL_dW1))
print('b1 gradient difference:', torch.linalg.norm(grad_b1 - dL_db1))
print('W2 gradient difference:', torch.linalg.norm(grad_W2 - dL_dW2))
print('b2 gradient difference:', torch.linalg.norm(grad_b2 - dL_db2))
```

```
W1 gradient difference: tensor(1.8306e-07, grad_fn=<LinalgVectorNormBackward0>)
b1 gradient difference: tensor(3.5550e-08, grad_fn=<LinalgVectorNormBackward0>)
W2 gradient difference: tensor(1.6843e-07, grad_fn=<LinalgVectorNormBackward0>)
b2 gradient difference: tensor(1.4901e-08, grad_fn=<LinalgVectorNormBackward0>)
```

1. (c)

[45]:

```
import torch
from torch.utils.data import DataLoader
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load the California Housing Prices dataset
data = fetch_california_housing()
X = data['data']
y = data['target']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,␣
 ↪random_state=42)

# Scale the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create a validation set from the training set
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
 ↪2, random_state=42)

# Define network dimensions
input_size = X_train.shape[1]
hidden_size = 20
output_size = 1
```

```
# Define network parameters
device = 'cuda'
W1 = torch.randn(hidden_size, input_size, requires_grad=True, device=device)
b1 = torch.randn(hidden_size, 1, requires_grad=True, device=device)
W2 = torch.randn(hidden_size, output_size, requires_grad=True, device=device)
b2 = torch.randn(output_size, requires_grad=True, device=device)
torch.nn.init.xavier_uniform_(W1)


torch.nn.init.xavier_uniform_(W2)


torch.nn.init.constant_(b1, 0.0)

torch.nn.init.constant_(b2, 0.0)

# Define batch size and number of epochs
batch_size = 64
num_epochs = 21

# Define optimizer and learning rate
optimizer = torch.optim.Adam([W1, b1, W2, b2], lr=0.001)

# Create data loaders
train_data = torch.utils.data.TensorDataset(torch.Tensor(X_train), torch.
 ↪Tensor(y_train))
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
val_data = torch.utils.data.TensorDataset(torch.Tensor(X_val), torch.
 ↪Tensor(y_val))
val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=False)
test_data = torch.utils.data.TensorDataset(torch.Tensor(X_test), torch.
 ↪Tensor(y_test))
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False)

# Train the network
for epoch in range(num_epochs):
    # Training loop
    train_loss = 0.0
    for i, (inputs, labels) in enumerate(train_loader):
        inputs = inputs.unsqueeze(-1)
        labels = labels.unsqueeze(-1).unsqueeze(-1)
        inputs = inputs.to(device)
        labels = labels.to(device)
        # Zero the gradients
        optimizer.zero_grad()
        # Forward pass
        outputs = forward(inputs)
        # Compute loss
        batch_loss = loss(outputs, labels.reshape(-1,1))
        train_loss += batch_loss.item() * len(inputs)
        # Backward pass
        batch_loss.backward()
        # Update parameters
        optimizer.step()
    train_loss /= len(train_loader.dataset)
    # Validation loop
    val_loss = 0.0
    for i, (inputs, labels) in enumerate(val_loader):
        inputs = inputs.unsqueeze(-1)
        labels = labels.unsqueeze(-1)
        inputs = inputs.to(device)
        labels = labels.to(device)
        # Forward pass
        outputs = forward(inputs)
        # Compute loss
        batch_loss = loss(outputs, labels.reshape(-1,1))
        val_loss += batch_loss.item() * len(inputs)
    val_loss /= len(val_loader.dataset)
    # Print loss
    print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f}, Val␣
 ↪Loss: {val_loss:.4f}')
test_loss = 0.0
for i, (inputs, labels) in enumerate(test_loader):
    inputs = inputs.unsqueeze(-1)
    labels = labels.unsqueeze(-1)
    inputs = inputs.to(device)
    labels = labels.to(device)
    # Forward pass
    outputs = forward(inputs)
    # Compute loss
    batch_loss = loss(outputs, labels.reshape(-1,1))
    test_loss += batch_loss.item() * len(inputs)
test_loss /= len(test_loader.dataset)
    # Print loss
print(f'Test Loss: {test_loss:.4f}')
```

```
Epoch [1/21], Train Loss: 1.5904, Val Loss: 1.1854
Epoch [2/21], Train Loss: 0.9699, Val Loss: 0.9250
Epoch [3/21], Train Loss: 0.8963, Val Loss: 0.9158
Epoch [4/21], Train Loss: 0.8895, Val Loss: 0.9126
Epoch [5/21], Train Loss: 0.8865, Val Loss: 0.9122
Epoch [6/21], Train Loss: 0.8845, Val Loss: 0.9104
Epoch [7/21], Train Loss: 0.8838, Val Loss: 0.9099
Epoch [8/21], Train Loss: 0.8835, Val Loss: 0.9105
Epoch [9/21], Train Loss: 0.8828, Val Loss: 0.9111
Epoch [10/21], Train Loss: 0.8829, Val Loss: 0.9109
Epoch [11/21], Train Loss: 0.8825, Val Loss: 0.9098
Epoch [12/21], Train Loss: 0.8821, Val Loss: 0.9093
Epoch [13/21], Train Loss: 0.8817, Val Loss: 0.9093
Epoch [14/21], Train Loss: 0.8817, Val Loss: 0.9093
Epoch [15/21], Train Loss: 0.8814, Val Loss: 0.9081
Epoch [16/21], Train Loss: 0.8810, Val Loss: 0.9104
Epoch [17/21], Train Loss: 0.8813, Val Loss: 0.9085
Epoch [18/21], Train Loss: 0.8807, Val Loss: 0.9085
Epoch [19/21], Train Loss: 0.8813, Val Loss: 0.9082
Epoch [20/21], Train Loss: 0.8806, Val Loss: 0.9079
Epoch [21/21], Train Loss: 0.8804, Val Loss: 0.9078
Test Loss: 0.8833
```

TABLE 1 – Time consumption of forward mode vs. backward mode, on 1000 test cases, when $D = K = 500$, $P = 1$.

mode	L=3		L=5		L=10	
	CPU	GPU	CPU	GPU	CPU	GPU
forward	0.24s	0.08s	0.37s	0.15s	0.76s	0.29s
backward	2.79s	0.29s	4.99s	0.51s	10.51s	1.10s

2. (c) I set $D = K = 500$, $P = 1$ and use 1000 test cases for visible time contrast. The results in table 1 show that on the CPU, forward mode Jacobian calculation consumes a dozen times more than backward mode Jacobian calculation, and on the GPU, forward mode Jacobian calculation consumes several times more than backward mode Jacobian calculation. Also, multiple increases slightly with L.

   (d) For clarity, we discuss the calculation in the case of $L = N, D = K = M$, $P = 1$. The forward mode has $N \times M^3 + M^2$, and the backward mode has $(N + 1) \times M^2$. Therefore, the larger the M is, the more obvious the difference in time consumption is between the two modes, such as $M = 500$ in the question (c).

The code part starts on the next page.

2. (a)

[200]:

```
%reset
import torch


def compute_jacobian_backward(x, weights):
    D = x.shape[0]
    K = weights['W1'].shape[0]
    P = weights['WF'].shape[0]
    length = len(weights)

    # forward pass
    forward_pass = []
    forward_pass.append(torch.matmul(weights['W1'], x))
    for i in range(2, length):
        forward_pass.append(torch.matmul(weights[f'W{i}'], torch.
 ↪tanh(forward_pass[-1])))

    # implement backward mode automatic differentiation
    result = torch.matmul(weights['WF'],
                          torch.diag(1 - torch.tanh(forward_pass[-1]) ** 2))
    for i in range(length-1, 1, -1):
        result = torch.matmul(torch.matmul(result, weights[f'W{i}']),
                              torch.diag(1 - torch.tanh(forward_pass[i-2])**2))
    result = torch.matmul(result, weights['W1'])

    return result

# Test the function
D = 2
K = 30
P = 10
L = 3
print('L =', L, 'D =', D, 'K =', K, 'P =', P)

# Set up random weights
weights = {}
weights['W1'] = torch.randn(K, D)
for i in range(2, L+1):
    weights[f'W{i}'] = torch.randn(K, K)
weights['WF'] = torch.randn(P, K)

# Set up random input
x = torch.randn(D)

# Compute Jacobian
```

```python
J = compute_jacobian_backward(x, weights)
print('manual jacobian: ', J)

def f(x):
    r = torch.matmul(weights['W1'], x)
    for i in range(2, len(weights)):
        r = torch.matmul(weights[f'W{i}'], torch.tanh(r))
    r = torch.matmul(weights['WF'], torch.tanh(r))
    return r

# Test against autograd
J_autograd = torch.autograd.functional.jacobian(f, x)
print('autograd jacobian: ', J_autograd)

# Check that the Jacobians match
assert torch.allclose(J, J_autograd, rtol=1e-4, atol=1e-4)
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
L = 3 D = 2 K = 30 P = 10
manual jacobian:  tensor([[ -1.9668,  -2.4276],
        [ 11.2458,  -1.6239],
        [  3.3068,   0.8205],
        [ -6.5943,   0.0701],
        [ -5.4183,   0.5918],
        [  7.5465,  -4.0497],
        [  6.6259,  -5.1651],
        [  8.9972,  -7.8698],
        [-15.7593,   7.5143],
        [-13.2292,   2.6224]])
autograd jacobian:  tensor([[ -1.9668,  -2.4276],
        [ 11.2458,  -1.6239],
        [  3.3068,   0.8205],
        [ -6.5943,   0.0701],
        [ -5.4183,   0.5918],
        [  7.5465,  -4.0497],
        [  6.6259,  -5.1651],
        [  8.9972,  -7.8698],
        [-15.7593,   7.5143],
        [-13.2292,   2.6224]])
```

2. (b)

[226]:

```python
def compute_jacobian_forward(x, weights):
    D = x.shape[0]
    K = weights['W1'].shape[0]
    P = weights['WF'].shape[0]
    length = len(weights)
    r_pass = torch.matmul(weights['W1'], x) # forward pass
    result = torch.matmul(torch.diag(1 - torch.tanh(r_pass) ** 2), weights['W1'])
    for i in range(2, length):
        r_pass = torch.matmul(weights[f'W{i}'], torch.tanh(r_pass))
        result = torch.matmul(torch.diag(1 - torch.tanh(r_pass) ** 2), torch.matmul(weights[f'W{i}'], result))
    result = torch.matmul(weights['WF'], result)

    return result


# Test the function
D = 2
K = 30
P = 10
L = 3
print('L =', L, 'D =', D, 'K =', K, 'P =', P)

# Set up random weights
weights = {}
weights['W1'] = torch.randn(K, D)
for i in range(2, L+1):
    weights[f'W{i}'] = torch.randn(K, K)
weights['WF'] = torch.randn(P, K)

for i in range(10):
    # Set up random input
    x = torch.randn(D)

    # Compute Jacobian backward
    J_backward = compute_jacobian_backward(x, weights)

    # Compute Jacobian forward
    J_forward = compute_jacobian_forward(x, weights)
    if i == 0:
        print('manual jacobian_backward: ', J_backward)
        print('manual jacobian_forward: ', J_forward)

    # Check that the Jacobians match
    assert torch.allclose(J_forward, J_backward, rtol=1e-4, atol=1e-4)
    print('All close!')
```

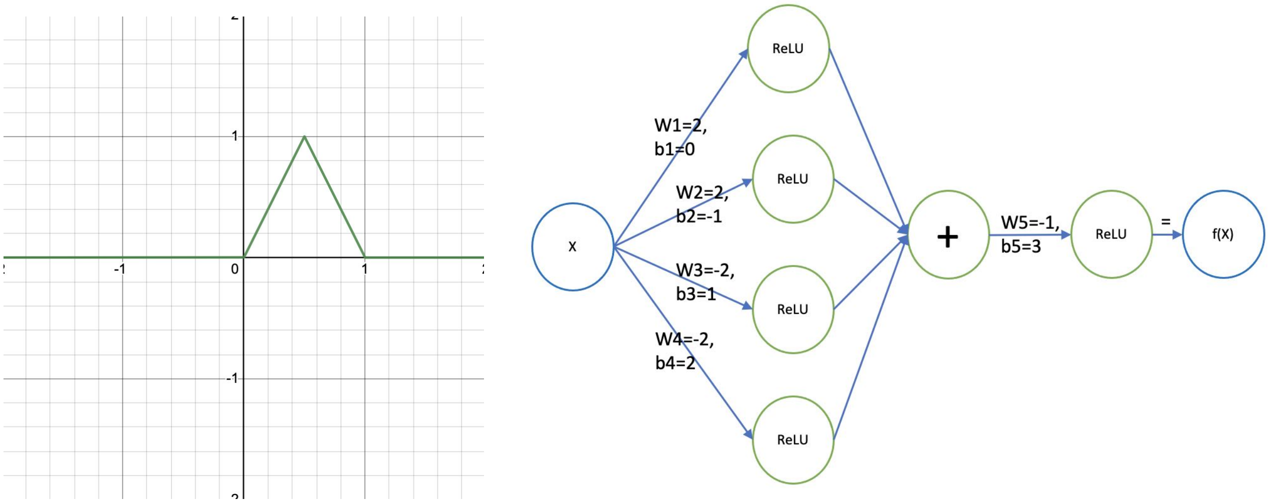
```
L = 3 D = 2 K = 30 P = 10
manual jacobian_backward:  tensor([[ 3.4693,  0.8789],
        [-0.5727, -1.8603],
        [-4.3573, -2.2833],
        [ 3.5770,  1.7826],
        [ 2.3216, -0.1311],
        [ 3.2716,  0.8445],
        [ 2.8224,  1.0990],
        [ 4.7019,  2.5888],
        [ 8.8580,  1.9520],
        [-0.8372,  0.0422]])
manual jacobian_forward:  tensor([[ 3.4693,  0.8789],
        [-0.5727, -1.8603],
        [-4.3573, -2.2833],
        [ 3.5770,  1.7826],
        [ 2.3216, -0.1311],
        [ 3.2716,  0.8445],
        [ 2.8224,  1.0990],
        [ 4.7019,  2.5888],
        [ 8.8580,  1.9520],
        [-0.8372,  0.0422]])
All close!
All close!
All close!
All close!
All close!
All close!
All close!
All close!
All close!
All close!
```

2. (c)

[286]:

```python
import time


D = 500
K = 500
P = 1
print(f'For clear comparison, I set D=K={K}, P={P}.')
for L in [3, 5, 10]:
    for device in ['cpu', 'cuda']:
        # Set up random weights
        weights = {}
        weights['W1'] = torch.randn((K, D), device=device)
        for i in range(2, L+1):
            weights[f'W{i}'] = torch.randn((K, K),device=device)
        weights['WF'] = torch.randn((P, K), device=device)

        print(f'====== L={L}, device={device}:')
        starting_time = time.time()
        for i in range(1000):
            x = torch.randn(D, device=device)
            J_backward = compute_jacobian_backward(x, weights)
        time_point = time.time()
        print(f'backward mode jacobian: {time_point - starting_time}s')
        for i in range(1000):
            x = torch.randn(D, device=device)
            J_forward = compute_jacobian_forward(x, weights)
        print(f'forward mode jacobian: {time.time() - time_point}s')
```

```
For clear comparison, I set D=K=500, P=1.
====== L=3, device=cpu:
backward mode jacobian: 0.23776507377624512s
forward mode jacobian: 2.790817975997925s
====== L=3, device=cuda:
backward mode jacobian: 0.08162641525268555s
forward mode jacobian: 0.28896093368530273s
====== L=5, device=cpu:
backward mode jacobian: 0.3728818893432617s
forward mode jacobian: 4.991540431976318s
====== L=5, device=cuda:
backward mode jacobian: 0.1506357192993164s
forward mode jacobian: 0.5093932151794434s
====== L=10, device=cpu:
backward mode jacobian: 0.756688117980957s
forward mode jacobian: 10.513039827346802s
====== L=10, device=cuda:
backward mode jacobian: 0.29082632064819336s
forward mode jacobian: 1.1020920276641846s
```



(a) (b)

Figure 1 – Problem 3(a)

3. (a) Figure 1(a) is the graphical representation of the function in the problem. In figure 1(b), we use the first layer, consisting of 4 neures : $ReLU(2x), ReLU(2x-1), ReLU(-2x+1), ReLU(-2x+2)$, to fit the shape of function in $0 < x < 1$ and the salient points at $x = 0, 1/2, 1$. Then we use $ReLU(-x+3)$ on the sum of outputs from the 4 neures, to form the whole function $f(x) = ReLU(-ReLU(2x) - ReLU(2x-1) - ReLU(-2x+1) - ReLU(-2x+2)+3)$.

Proof : as show in figure 1(b), when $0 \leq x \leq 1/2$,

$$\begin{aligned} f(x) &= ReLU(-(2x - 2x + 1 - 2x + 2) + 3) \\ &= ReLU(-(-2x+3)+3) \\ &= ReLU(2x) \\ &= 2x. \end{aligned}$$

When $1/2 \leq x \leq 1$,

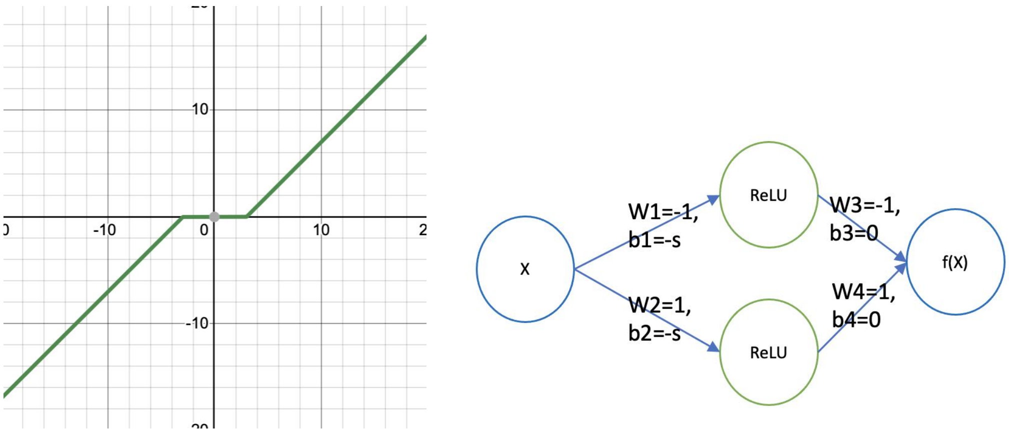
$$\begin{aligned} f(x) &= ReLU(-(2x2x - 1 - 2x + 2) + 3) \\ &= ReLU(-(2x+1)+3) \\ &= ReLU(-2x+2) \\ &= -2x+2. \end{aligned}$$

When $x < 0$,

$$\begin{aligned} f(x) &= ReLU(-(-2x + 1 - 2x + 2) + 3) \\ &= ReLU(-(-4x+3)+3) \\ &= ReLU(4x) \\ &= 0. \end{aligned}$$

When $x > 1$,

$$\begin{aligned} f(x) &= ReLU(-(2x + 2x - 1) + 3) \\ &= ReLU(-(4x - 1) + 3) \\ &= ReLU(-4x + 4) \\ &= 0. \end{aligned}$$



(a)

(b)

Figure 2 – Problem 3(b)

(b) Figure 2(a) is the graphical representation of the function in the problem. The formula of figure 2(b) is $f(x) = -ReLU(-x - s) + ReLU(x - s)$.

Proof : when $x < -s$,

$$f(x) = -ReLU(-x - s) + 0 = x + s.$$

When $-s \leq x \leq s$,

$$f(x) = 0.$$

When $x > s$,

$$f(x) = ReLU(x - s) = x - s.$$

4. (e) Deeper networks have more complex activation, they can also suffer from the vanishing gradient problem, where gradients become very small as they are backpropagated through many layers. Initialization can also have a significant impact on activations, especially for deep networks. Poor initialization can lead to activations that are too small or too large, causing the gradients to vanish or explode during backpropagation, thus networks fail to go to a convergence.

The code part starts on the next page.

4. (a)(b)

[120]:

```
%reset
import torch
import torch.nn as nn
import torch.nn.init as init


class MyModel(nn.Module):
    def __init__(self, depth, d):
        super(MyModel, self).__init__()
        self.depth = depth
        self.layers = nn.ModuleList([nn.Linear(784, 50)])
        self.layers.extend([nn.Linear(50, 50) for i in range(depth-2)])
        self.layers.extend([nn.Linear(50, 10)])
        self.activation = nn.Tanh()

        self.d = d
        self.initialize_weights()

    def initialize_weights(self):
        if self.d == 'Xavier':
            for layer in self.layers:
                self.d = np.sqrt(6/(layer.in_features + layer.out_features))
                init.uniform_(layer.weight, -self.d, self.d)
                init.zeros_(layer.bias)
        else:
            for layer in self.layers:
                init.uniform_(layer.weight, -self.d, self.d)
                init.zeros_(layer.bias)

    def forward(self, x):
        x = x.view(-1, 784)
        for i in range(self.depth):
            x = self.activation(self.layers[i](x))
        return x
```

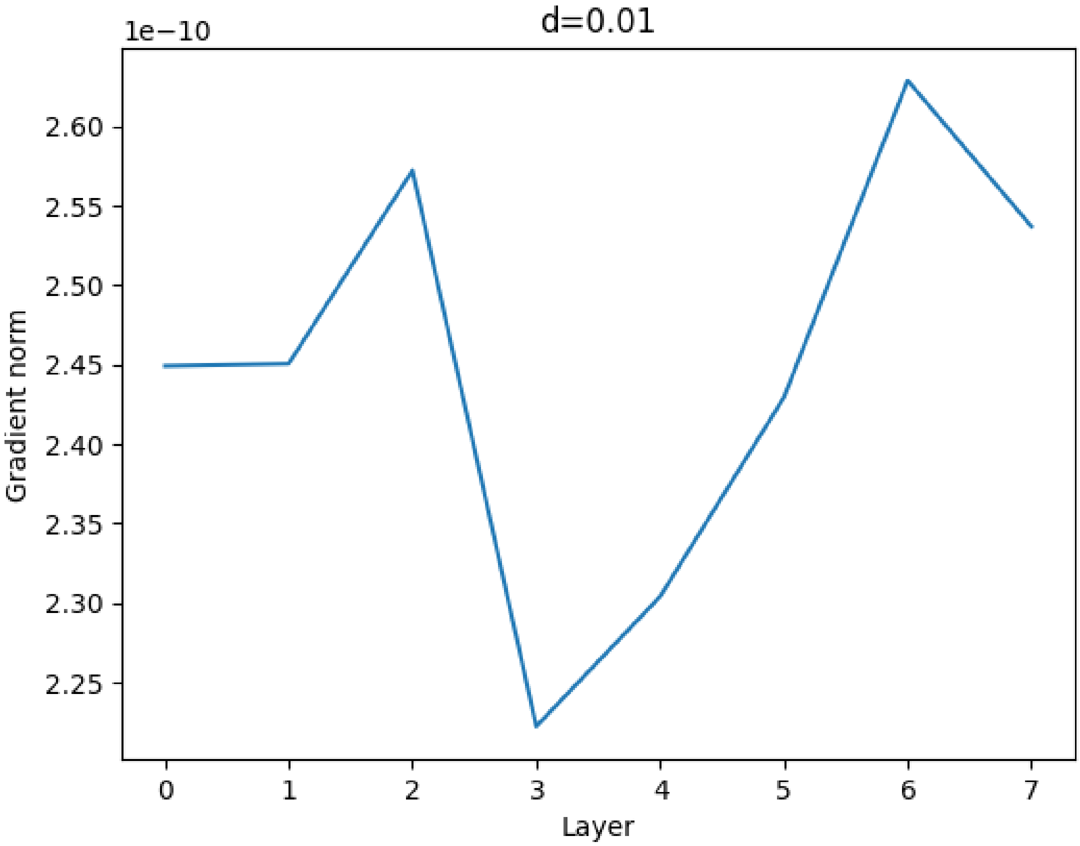
```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

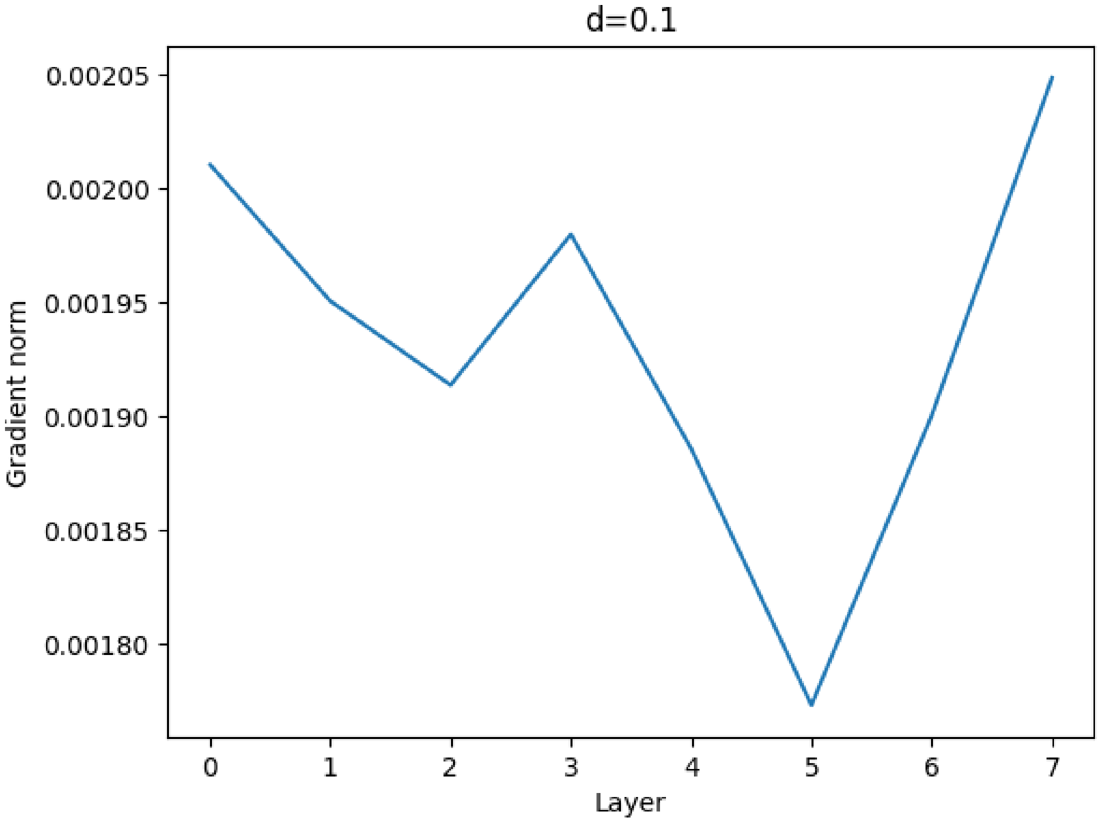
4. (c)

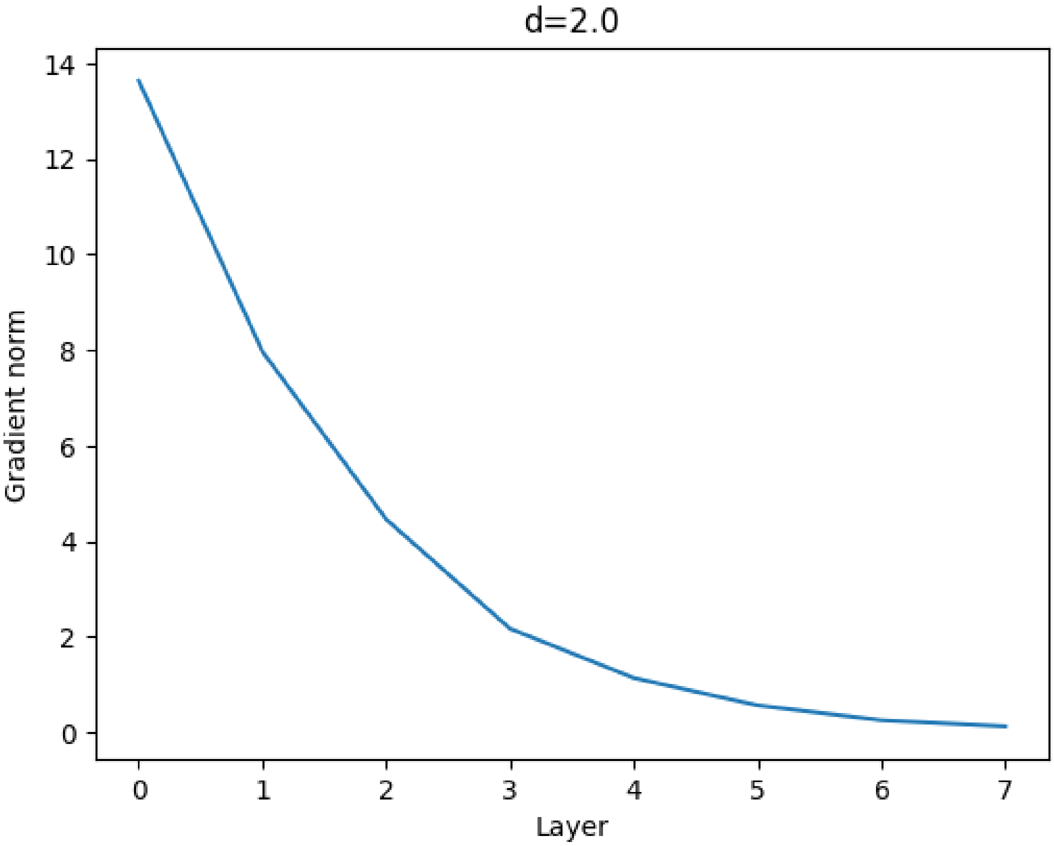
[126]:

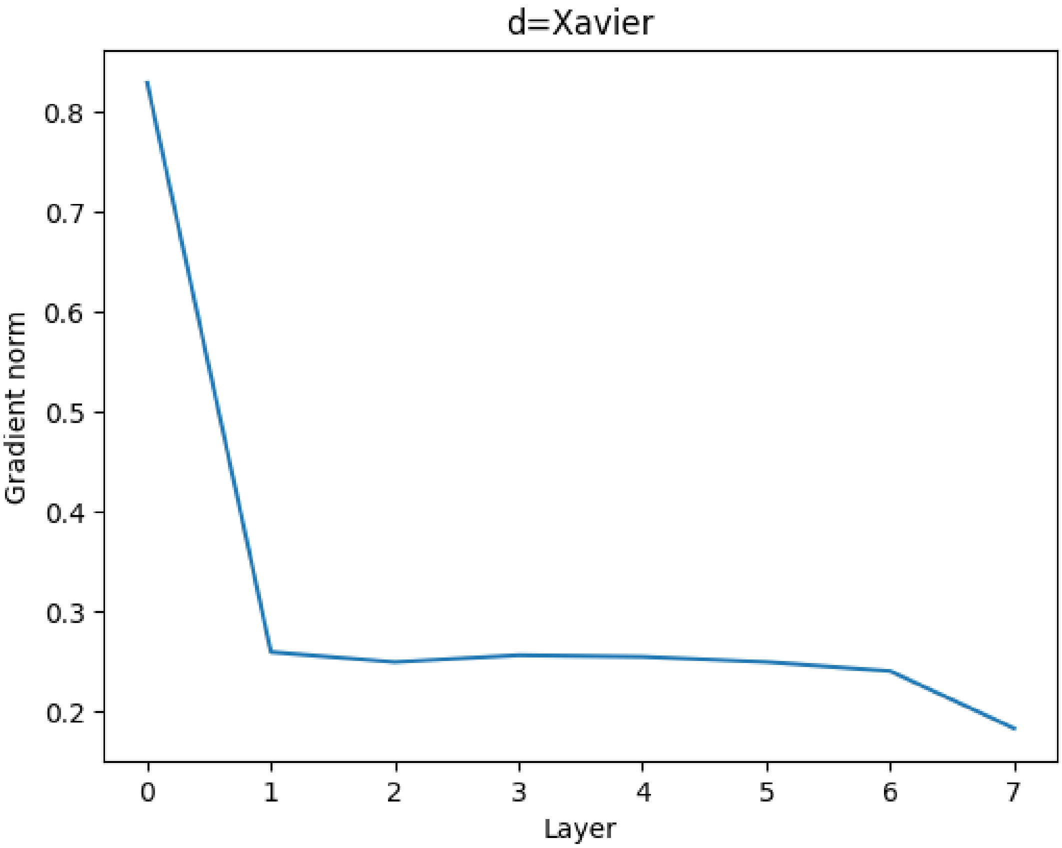
```
import numpy as np
import matplotlib.pyplot as plt
from torchvision import datasets, transforms


device = 'cuda'
```

```python
# Define the loss function
criterion = nn.CrossEntropyLoss()

# Define the input tensor and labels for the minibatch
train_dataset = datasets.MNIST('./data', train=True, download=True, transform=transforms.ToTensor())
train_loader = torch.utils.data.DataLoader(dataset1,
                                          batch_size=256,
                                          shuffle=True,
                                          drop_last=True)

# Get one batch of data
for (input_data, labels) in train_loader:
    input_data = input_data.to(device)
    labels = labels.to(device)
    break

# Create an instance of the model for each initialization scheme
depth = 8
d_values = [0.01, 0.1, 2.0, 'Xavier']
# d_values = [0.01, 0.1, 'Xavier']
nets = [MyModel(depth, d) for d in d_values]

# Forward pass through each network, retaining gradients for the layer outputs
for (index, net) in enumerate(nets):
    net.to(device)
    net.train()
    with torch.set_grad_enabled(True):

        outputs = net(input_data)
        loss = criterion(outputs, labels)
        loss.backward()
        gradients = []
        for i in range(depth):
            gradients.append(torch.norm(net.layers[i].weight.grad))  # Compute the gradient norm at each layer


    # Visualize the gradient norms at each layer
    plt.plot(np.arange(depth), [g.item() for g in gradients])
    plt.title(f"d={d_values[index]}")
    plt.xlabel("Layer")
    plt.ylabel("Gradient norm")
    plt.show()
```


d=0.01
1e−10
Gradient norm
2.60
2.55
2.50
2.45
2.40
2.35
2.30
2.25
0
1
2
3
4
5
6
7
Layer


d=0.1
Gradient norm
0.00205
0.00200
0.00195
0.00190
0.00185
0.00180
Layer
0
1
2
3
4
5
6
7


d=2.0
14
12
10
8
6
4
2
0
Gradient norm
0
1
2
3
4
5
6
7
Layer

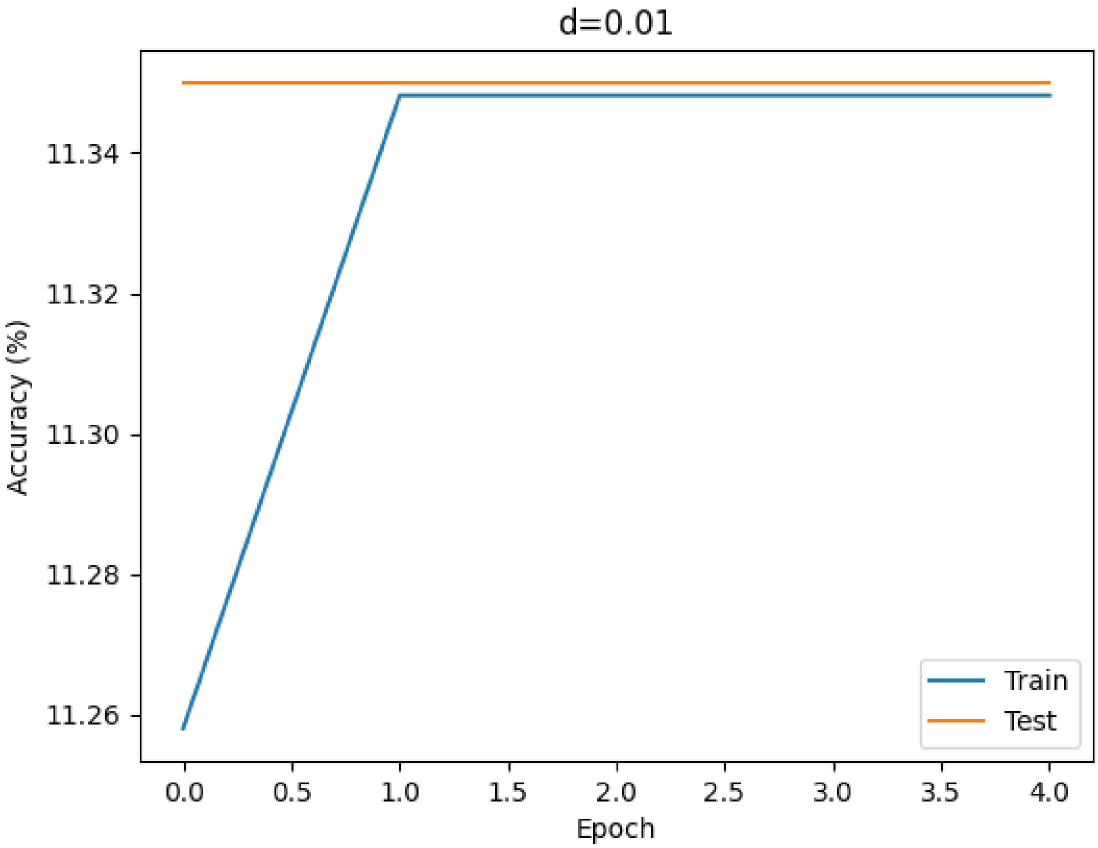


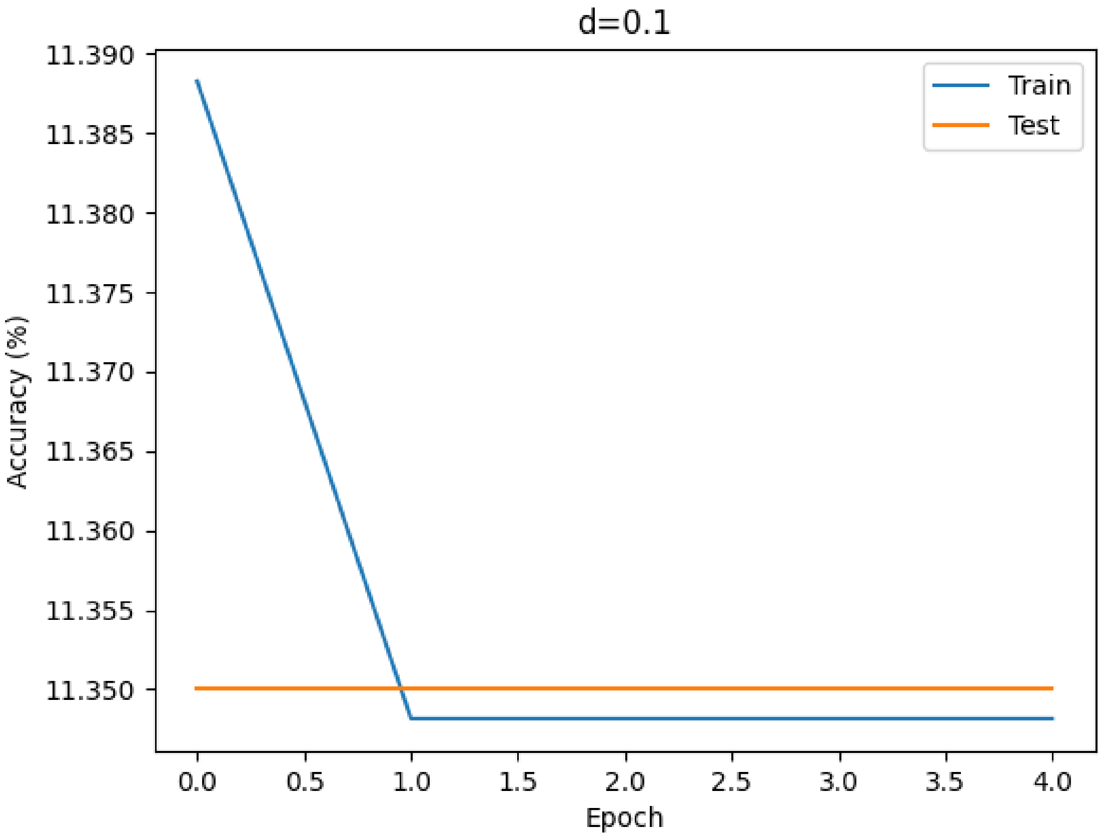
4. (d)

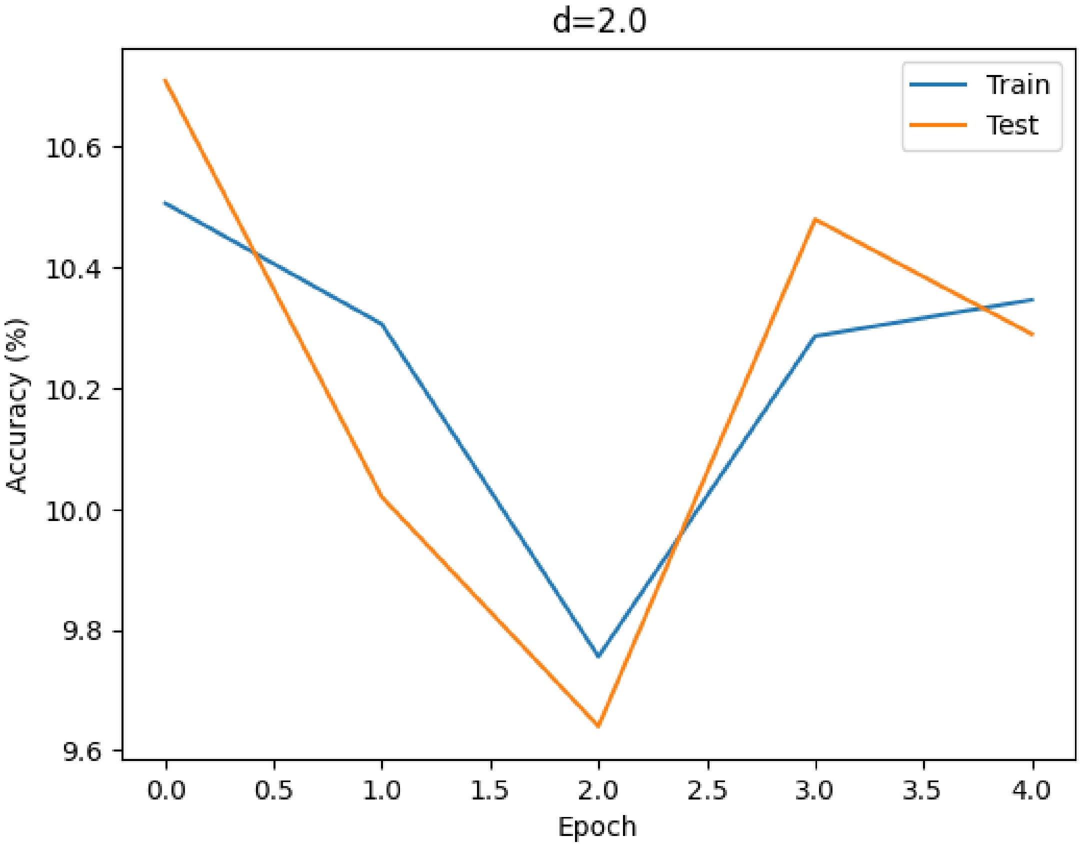
```
[143]: import torch.optim as optim


       # Define the training and testing datasets
       test_dataset = datasets.MNIST('./data', train=False, download=True,␣
        ↪transform=transforms.ToTensor())

       # Define the data loaders
       batch_size = 128
       train_loader = torch.utils.data.DataLoader(train_dataset,
                                                  batch_size=batch_size,
                                                  shuffle=True,
                                                  drop_last=True)
       train_loader = torch.utils.data.DataLoader(test_dataset,
                                                  batch_size=batch_size,
                                                  shuffle=False,
                                                  drop_last=True)
```

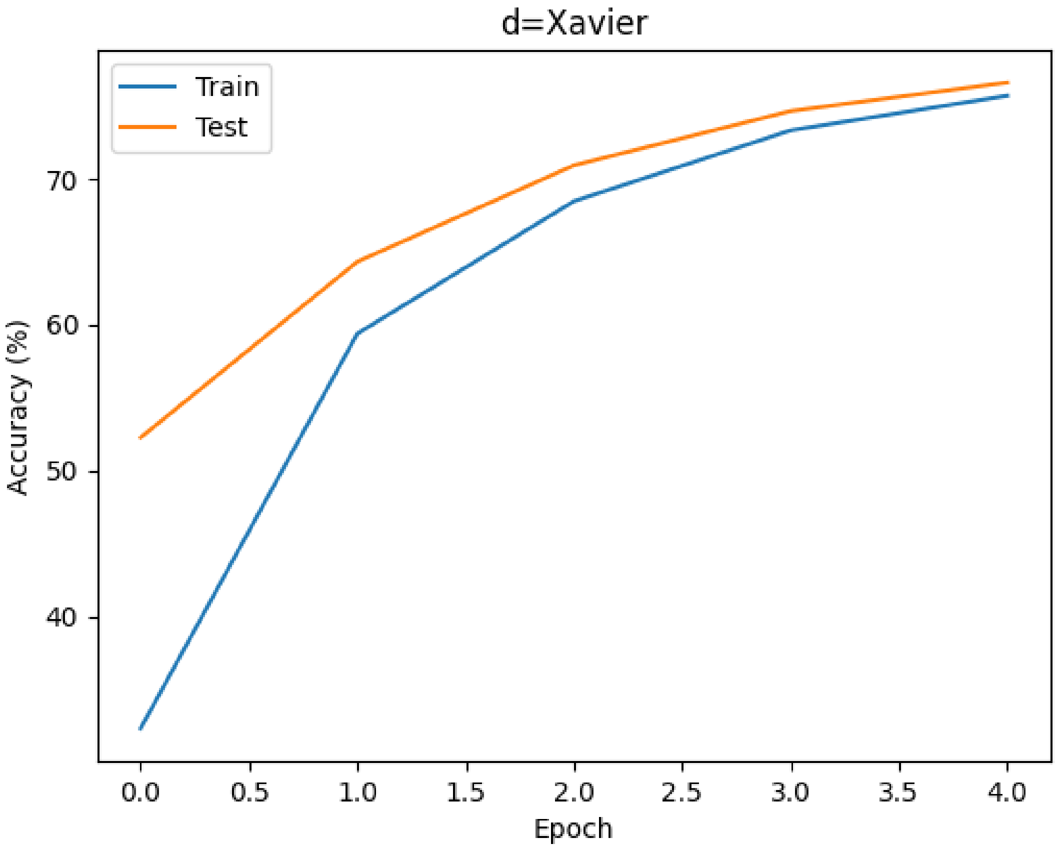
```
# Define the model and optimizer for each initialization scheme
depth = 8
d_values = [0.01, 0.1, 2.0, 'Xavier']
nets = [MyModel(depth, d) for d in d_values]
optimizers = [optim.SGD(net.parameters(), lr=0.01) for net in nets]

# Train each model for 5 epochs and record the training and testing accuracy␣
 ↪after each epoch
n_epochs = 5
train_accs = [[] for _ in range(len(nets))]
test_accs = [[] for _ in range(len(nets))]
for i, net in enumerate(nets):
    net.to(device)
    net.train()
    for epoch in range(n_epochs):
        optimizer = optimizers[i]
        train_correct = 0
        train_total = 0
        for input_data, labels in train_loader:
            input_data = input_data.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()
            outputs = net(input_data)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # Compute the training accuracy
            _, predicted = torch.max(outputs.data, 1)
            train_total += labels.size(0)
            train_correct += (predicted == labels).sum().item()

        train_acc = 100 * train_correct / train_total
        train_accs[i].append(train_acc)

        # Evaluate the model on the test set
        net.eval()
        test_correct = 0
        test_total = 0
        with torch.no_grad():
            for input_data, labels in test_loader:
                input_data = input_data.to(device)
                labels = labels.to(device)
                outputs = net(input_data)
                _, predicted = torch.max(outputs.data, 1)
                test_total += labels.size(0)
                test_correct += (predicted == labels).sum().item()
        test_acc = 100 * test_correct / test_total
        test_accs[i].append(test_acc)

        print(f"Initialization with {d_values[i]}, epoch {epoch+1}, train acc:␣
↪{train_acc:.2f}%, test acc: {test_acc:.2f}%")

# Plot the training and testing accuracy curves for each initialization scheme
for i in range(len(nets)):
    plt.plot(np.arange(n_epochs), train_accs[i], label="Train")
    plt.plot(np.arange(n_epochs), test_accs[i], label="Test")
    plt.legend()
    plt.title(f"d={d_values[i]}")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy (%)")
    plt.show()
```

```
Initialization with 0.01, epoch 1, train acc: 11.26%, test acc: 11.35%
Initialization with 0.01, epoch 2, train acc: 11.35%, test acc: 11.35%
Initialization with 0.01, epoch 3, train acc: 11.35%, test acc: 11.35%
Initialization with 0.01, epoch 4, train acc: 11.35%, test acc: 11.35%
Initialization with 0.01, epoch 5, train acc: 11.35%, test acc: 11.35%
Initialization with 0.1, epoch 1, train acc: 11.39%, test acc: 11.35%
Initialization with 0.1, epoch 2, train acc: 11.35%, test acc: 11.35%
Initialization with 0.1, epoch 3, train acc: 11.35%, test acc: 11.35%
Initialization with 0.1, epoch 4, train acc: 11.35%, test acc: 11.35%
Initialization with 0.1, epoch 5, train acc: 11.35%, test acc: 11.35%
Initialization with 2.0, epoch 1, train acc: 10.51%, test acc: 10.71%
Initialization with 2.0, epoch 2, train acc: 10.31%, test acc: 10.02%
Initialization with 2.0, epoch 3, train acc: 9.76%, test acc: 9.64%
Initialization with 2.0, epoch 4, train acc: 10.29%, test acc: 10.48%
Initialization with 2.0, epoch 5, train acc: 10.35%, test acc: 10.29%
Initialization with Xavier, epoch 1, train acc: 32.30%, test acc: 52.24%
Initialization with Xavier, epoch 2, train acc: 59.36%, test acc: 64.30%
Initialization with Xavier, epoch 3, train acc: 68.45%, test acc: 70.91%
Initialization with Xavier, epoch 4, train acc: 73.31%, test acc: 74.63%
Initialization with Xavier, epoch 5, train acc: 75.68%, test acc: 76.59%
```


d=0.01
Accuracy (%)
11.34
11.32
11.30
11.28
11.26
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epoch
Train
Test


d=0.1
Train
Test
Accuracy (%)
11.390
11.385
11.380
11.375
11.370
11.365
11.360
11.355
11.350
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epoch


d=2.0
Train
Test
10.6
10.4
10.2
10.0
9.8
9.6
Accuracy (%)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epoch


d=Xavier
Train
Test
70
60
50
40
Accuracy (%)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epoch

5. (a) The squared loss function $L(X,w,y)$ can be written as :

$$\begin{aligned} L(X,w,y) &= \frac{1}{2}\|Xw-y\|^2 \\ &= \frac{1}{2}(Xw-y)^T(Xw-y) \\ &= \frac{1}{2}(w^TX^TXw - w^TX^Ty - y^TXw + y^Ty) \end{aligned}$$

Taking the derivative of this expression with respect to w, we get :

$$\begin{aligned} \nabla_w L(X,w,y) &= \frac{1}{2}(2X^TXw - w^TX^Ty - y^TXw + y^Ty) \\ &= \frac{1}{2}(2X^TXw - X^Ty - (y^TX)^T) \\ &= X^TXw - X^Ty \end{aligned}$$

Therefore, the gradient of the squared loss function with respect to w is $X^TXw - X^Ty$. To find the minimizer of this loss function, we set the gradient equal to zero because this is a convex optimization problem, and solve for w :

$$\begin{aligned} X^TXw - X^Ty &= 0 \\ X^TXw &= X^Ty \\ w &= (X^TX)^{-1}X^Ty \end{aligned}$$

Therefore, the minimizer of the squared loss function is $w = (X^TX)^{-1}X^Ty$.

(b) For the first iterate, we have w_1 :

$$\begin{aligned} w_1 &= w_0 - \alpha\nabla_w L(X,w_0,y) \\ &= w_0 - \alpha(X^TXw_0 - X^Ty) \\ &= (I - \alpha X^TX)w_0 + \alpha X^Ty \end{aligned}$$

For the second iterate, we have w_2 :

$$\begin{aligned} w_2 &= w_1 - \alpha\nabla_w L(X,w_1,y) \\ &= w_1 - \alpha(X^TXw_1 - X^Ty) \\ &= w_1 - \alpha X^TXw_1 + \alpha X^Ty \\ &= w_0 - \alpha(X^TXw_0 - X^Ty) - \alpha X^TX(w_0 - \alpha(X^TXw_0 - X^Ty)) + \alpha X^Ty \\ &= w_0 - \alpha X^TXw_0 + \alpha X^Ty - \alpha X^TXw_0 + \alpha^2X^TXX^TXw_0 - \alpha^2X^TXX^Ty + \alpha X^Ty \\ &= (I - 2\alpha X^TX + \alpha^2X^TXX^TX)w_0 + \alpha X^Ty - \alpha^2X^TXX^Ty + \alpha X^Ty \\ &= (I - \alpha X^TX)^2w_0 + (I - \alpha X^TX)\alpha X^Ty + \alpha X^Ty \end{aligned}$$

(c) From results of (b), we can get :

$$\begin{aligned}
w_1 &= w_0 - \alpha(X^TXw_0 - X^Ty) \\
w_2 &= w_1 - \alpha(X^TXw_1 - X^Ty) \\
&= (I - \alpha X^TX)w_1 + \alpha X^Ty \\
&= (I - \alpha X^TX)^2 w_0 + (I - \alpha X^TX)\alpha X^Ty + \alpha X^Ty \\
w_3 &= w_2 - \alpha(X^TXw_2 - X^Ty) \\
&= (I - \alpha X^TX)w_2 + \alpha X^Ty \\
&= (I - \alpha X^TX)^3 w_0 + (I - \alpha X^TX)^2\alpha X^Ty + (I - \alpha X^TX)\alpha X^Ty + \alpha X^Ty
\end{aligned}$$

Let's start a mathematical induction : we assume that the update rule for w_t is given by :

$$w_t = (I - \alpha X^TX)^t w_0 + \sum_{i=0}^{t-1}(I - \alpha X^TX)^i(\alpha X^Ty)$$

Now, the rule is true for $t = 1$, then we can show that this holds true for $t + 1$:

$$\begin{aligned}
w_{t+1} &= w_t - \alpha(X^TXw_t - X^Ty) \\
&= (I - \alpha X^TX)w_t + \alpha X^Ty \\
&= (I - \alpha X^TX)[(I - \alpha X^TX)^t w_0 + \sum_{i=0}^{t-1}(I - \alpha X^TX)^i(\alpha X^Ty)] + \alpha X^Ty \\
&= (I - \alpha X^TX)^{t+1} w_0 + \sum_{i=0}^{t}(I - \alpha X^TX)^i(\alpha X^Ty)
\end{aligned}$$

Thus, the assumed update rule is proved, we get the desired w_k :

$$w_k = (I - \alpha X^TX)^k w_0 + \sum_{i=0}^{k-1}(I - \alpha X^TX)^i(\alpha X^Ty)$$

(d) $(I - \alpha X^TX)$ is a real symmetric matrix so it is diagonalizable, which means $(I - \alpha X^TX)$ can be represented as PDP^{-1} where the diagonal entries of D are the eigenvalues of $(I - \alpha X^TX)$ in magnitude. We can assume that α is small enough such that its eigenvalues are all less than 1. We get :

$$\lim_{k\to\infty}(I - \alpha X^TX)^k = \lim_{k\to\infty}(PDP^{-1})^k = \lim_{k\to\infty} PD^kP^{-1} = \lim_{k\to\infty} P0P^{-1} = 0$$

Thus, we get :

$$\lim_{k\to\infty} w_k = \lim_{k\to\infty}\sum_{i=0}^{k-1}(I - \alpha X^TX)^i(\alpha X^Ty) = \left(\sum_{i=0}^{\infty}(I - \alpha X^TX)^i\right)(\alpha X^Ty)$$

The sum $\sum_{i=0}^{\infty}(I-\alpha X^TX)^i$ is a geometric series, so we get :

$$\begin{aligned}\lim_{k\to\infty} w_k &= \lim_{k\to\infty} \frac{I-(I-\alpha X^TX)^k}{I-(I-\alpha X^TX)}\alpha X^T y\\ &= \frac{I}{\alpha X^TX}\alpha X^T y\\ &= (X^TX)^{-1}X^Ty\end{aligned}$$