Solution 1.1):

Co-adaptation refers to the phenomenon in deep learning where multiple components of a model become highly dependent on each other to achieve good performance, to the point where it becomes difficult to isolate the contributions of individual components. This can lead to overfitting and poor generalization when the model is applied to new data.

In deep learning, a model typically consists of many layers, each of which contains a large number of learnable parameters. These parameters are updated during training to optimize the model's performance on a training set. However, because the layers are highly interconnected, the optimization process can result in some layers becoming dependent on the output of others. This dependency can lead to a situation where the performance of the entire model is heavily dependent on the behavior of a few key components.

Internal covariance-shift, on the other hand, refers to a problem that can arise in machine learning algorithms when the statistical distribution of the input data changes between training and testing. This can happen, for example, if the training data is drawn from one population or environment, and the testing data is drawn from a different population or environment.

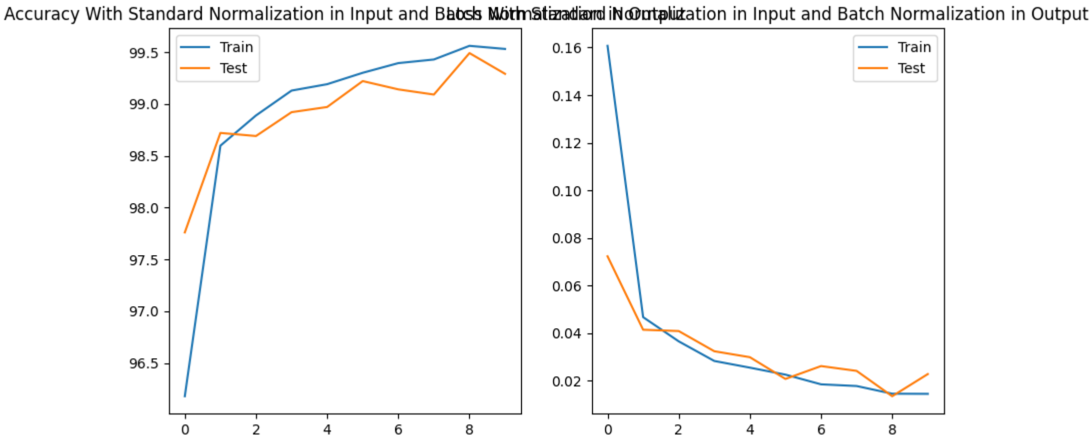
The problem with internal covariance-shift is that the model may not be able to generalize well to the testing data, because the statistical patterns that it learned from the training data no longer hold. This can lead to poor performance and low accuracy.

One way to mitigate the effects of internal covariance-shift is to use techniques such as domain adaptation or transfer learning, which aim to adapt the model to the new testing data by adjusting its internal representations or learning new representations from a related task. These techniques can help the model to generalize better to new environments and improve its accuracy.

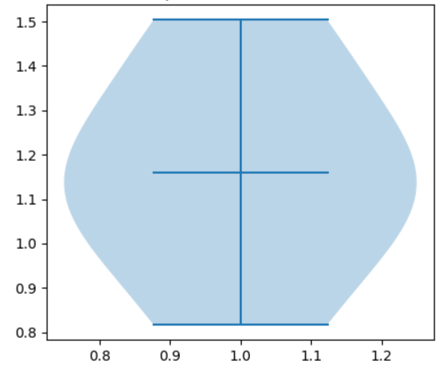
In [ ]:
```python
#Solution 1.2):

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt

# Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transforms.Compose([
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.131,), (0.302,))
                                ]), download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transforms.Compose([
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.131,), (0.302,))
                                ]), download=True)

# Define the data loaders
batch_size = 64
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Define the LeNet-5 model with batch normalization for all layers
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.bn1 = nn.BatchNorm2d(6)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.bn2 = nn.BatchNorm2d(16)
        self.fc1 = nn.Linear(16*4*4, 120)
        self.bn3 = nn.BatchNorm1d(120)
        self.fc2 = nn.Linear(120, 84)
        self.bn4 = nn.BatchNorm1d(84)
        self.fc3 = nn.Linear(84, 10)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        x = self.bn1(self.conv1(x))
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.bn2(self.conv2(x))
        x = self.relu(x)
        x = self.maxpool(x)
        x = x.view(-1, 16*4*4)
        x = self.bn3(self.fc1(x))
        x = self.relu(x)
        x = self.bn4(self.fc2(x))
        x = self.relu(x)
        x = self.fc3(x)
        return x

# Initialize the model
model = LeNet5()

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

# Define the batch size and number of epochs
batch_size = 64
n_epochs = 10

# Create data loaders for the train and test datasets
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Train the model
train_loss_bn = []
train_acc_bn = []
test_loss_bn = []
test_acc_bn = []

for epoch in range(n_epochs):
    model.train()
    train_loss = 0.0
    train_total = 0
    train_correct = 0
    for i, (inputs, labels) in enumerate(train_loader):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * inputs.size(0)
        _, predicted = torch.max(outputs.data, 1)
        train_total += labels.size(0)
        train_correct += (predicted == labels).sum().item()
```

```
        # Print training statistics
        if (i+1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch+1, n_epochs, i+1, len(train_loader), loss.item()))

    train_loss /= len(train_loader.dataset)
    train_acc = 100 * train_correct / train_total
    train_loss_bn.append(train_loss)
    train_acc_bn.append(train_acc)

    model.eval()
    test_loss = 0.0
    test_total = 0
    test_correct = 0
    for i, (inputs, labels) in enumerate(test_loader):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        test_loss += loss.item() * inputs.size(0)
        _, predicted = torch.max(outputs.data, 1)
        test_total += labels.size(0)
        test_correct += (predicted == labels).sum().item()

    test_loss /= len(test_loader.dataset)
    test_acc = 100 * test_correct / test_total
    test_loss_bn.append(test_loss)
    test_acc_bn.append(test_acc)

    print('Epoch [{}/{}], Train Loss: {:.4f}, Train Acc: {:.2f}, Test Loss: {:.4f}, Test Acc: {:.2f}'.format(epoch+1, n_epochs, train_loss, train_acc, test_loss, test_acc))

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(train_acc_bn, label='Train')
plt.plot(test_acc_bn, label='Test')
plt.title('Accuracy With Standard Normalization in Input and Batch Normalization in Output')
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(train_loss_bn, label='Train')
plt.plot(test_loss_bn, label='Test')
plt.title('Loss With Standard Normalization in Input and Batch Normalization in Output')
plt.legend()
plt.show()

bn_params = []
for name, module in model.named_modules():
    if isinstance(module, nn.BatchNorm2d) or isinstance(module, nn.BatchNorm1d):
        bn_params.append(module.weight.data.cpu().numpy())

fig, axs = plt.subplots(len(bn_params), figsize=(5, 20))
for i, p in enumerate(bn_params):
    axs[i].violinplot(dataset=p, showmeans=True)
    axs[i].set_title(f'Standard Normalization in Input and Batch Normalization in Output Layer {i+1}')
plt.show()
```

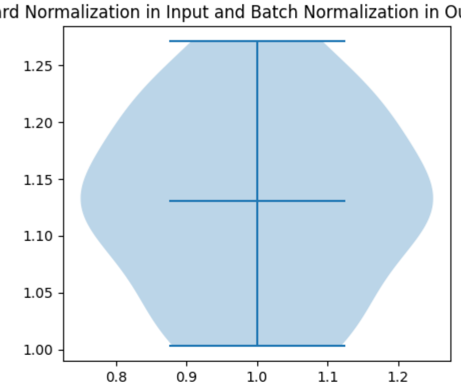
```
Epoch [1/10], Step [100/938], Loss: 0.2941
Epoch [1/10], Step [200/938], Loss: 0.0614
Epoch [1/10], Step [300/938], Loss: 0.0811
Epoch [1/10], Step [400/938], Loss: 0.2154
Epoch [1/10], Step [500/938], Loss: 0.1622
Epoch [1/10], Step [600/938], Loss: 0.0694
Epoch [1/10], Step [700/938], Loss: 0.0488
Epoch [1/10], Step [800/938], Loss: 0.2468
Epoch [1/10], Step [900/938], Loss: 0.0083
Epoch [1/10], Train Loss: 0.1607, Train Acc: 96.18, Test Loss: 0.0723, Test Acc: 97.76
Epoch [2/10], Step [100/938], Loss: 0.0553
Epoch [2/10], Step [200/938], Loss: 0.0417
Epoch [2/10], Step [300/938], Loss: 0.0132
Epoch [2/10], Step [400/938], Loss: 0.0263
Epoch [2/10], Step [500/938], Loss: 0.0233
Epoch [2/10], Step [600/938], Loss: 0.0176
Epoch [2/10], Step [700/938], Loss: 0.0196
Epoch [2/10], Step [800/938], Loss: 0.0874
Epoch [2/10], Step [900/938], Loss: 0.0115
Epoch [2/10], Train Loss: 0.0467, Train Acc: 98.60, Test Loss: 0.0414, Test Acc: 98.72
Epoch [3/10], Step [100/938], Loss: 0.0031
Epoch [3/10], Step [200/938], Loss: 0.0406
Epoch [3/10], Step [300/938], Loss: 0.0283
Epoch [3/10], Step [400/938], Loss: 0.1055
Epoch [3/10], Step [500/938], Loss: 0.0666
Epoch [3/10], Step [600/938], Loss: 0.0175
Epoch [3/10], Step [700/938], Loss: 0.0096
Epoch [3/10], Step [800/938], Loss: 0.0414
Epoch [3/10], Step [900/938], Loss: 0.0332
Epoch [3/10], Train Loss: 0.0365, Train Acc: 98.89, Test Loss: 0.0408, Test Acc: 98.69
Epoch [4/10], Step [100/938], Loss: 0.0264
Epoch [4/10], Step [200/938], Loss: 0.0016
Epoch [4/10], Step [300/938], Loss: 0.0021
Epoch [4/10], Step [400/938], Loss: 0.0271
Epoch [4/10], Step [500/938], Loss: 0.0073
Epoch [4/10], Step [600/938], Loss: 0.0045
Epoch [4/10], Step [700/938], Loss: 0.0188
Epoch [4/10], Step [800/938], Loss: 0.0101
Epoch [4/10], Step [900/938], Loss: 0.0118
Epoch [4/10], Train Loss: 0.0283, Train Acc: 99.13, Test Loss: 0.0324, Test Acc: 98.92
Epoch [5/10], Step [100/938], Loss: 0.0198
Epoch [5/10], Step [200/938], Loss: 0.0045
Epoch [5/10], Step [300/938], Loss: 0.0471
Epoch [5/10], Step [400/938], Loss: 0.0041
Epoch [5/10], Step [500/938], Loss: 0.0202
Epoch [5/10], Step [600/938], Loss: 0.0427
Epoch [5/10], Step [700/938], Loss: 0.0245
Epoch [5/10], Step [800/938], Loss: 0.0189
Epoch [5/10], Step [900/938], Loss: 0.0468
Epoch [5/10], Train Loss: 0.0255, Train Acc: 99.19, Test Loss: 0.0299, Test Acc: 98.97
Epoch [6/10], Step [100/938], Loss: 0.0024
Epoch [6/10], Step [200/938], Loss: 0.0768
Epoch [6/10], Step [300/938], Loss: 0.0051
Epoch [6/10], Step [400/938], Loss: 0.0006
Epoch [6/10], Step [500/938], Loss: 0.0019
Epoch [6/10], Step [600/938], Loss: 0.0059
Epoch [6/10], Step [700/938], Loss: 0.0028
Epoch [6/10], Step [800/938], Loss: 0.0056
Epoch [6/10], Step [900/938], Loss: 0.0037
Epoch [6/10], Train Loss: 0.0225, Train Acc: 99.30, Test Loss: 0.0207, Test Acc: 99.22
Epoch [7/10], Step [100/938], Loss: 0.0108
Epoch [7/10], Step [200/938], Loss: 0.0187
Epoch [7/10], Step [300/938], Loss: 0.0002
Epoch [7/10], Step [400/938], Loss: 0.0217
Epoch [7/10], Step [500/938], Loss: 0.0030
Epoch [7/10], Step [600/938], Loss: 0.0230
Epoch [7/10], Step [700/938], Loss: 0.0342
Epoch [7/10], Step [800/938], Loss: 0.0002
Epoch [7/10], Step [900/938], Loss: 0.0041
Epoch [7/10], Train Loss: 0.0185, Train Acc: 99.39, Test Loss: 0.0261, Test Acc: 99.14
Epoch [8/10], Step [100/938], Loss: 0.0020
Epoch [8/10], Step [200/938], Loss: 0.0065
Epoch [8/10], Step [300/938], Loss: 0.0011
Epoch [8/10], Step [400/938], Loss: 0.0233
Epoch [8/10], Step [500/938], Loss: 0.0014
Epoch [8/10], Step [600/938], Loss: 0.0024
Epoch [8/10], Step [700/938], Loss: 0.0035
Epoch [8/10], Step [800/938], Loss: 0.0128
Epoch [8/10], Step [900/938], Loss: 0.0003
Epoch [8/10], Train Loss: 0.0177, Train Acc: 99.43, Test Loss: 0.0241, Test Acc: 99.09
Epoch [9/10], Step [100/938], Loss: 0.0116
Epoch [9/10], Step [200/938], Loss: 0.0097
Epoch [9/10], Step [300/938], Loss: 0.0204
Epoch [9/10], Step [400/938], Loss: 0.0099
Epoch [9/10], Step [500/938], Loss: 0.0126
Epoch [9/10], Step [600/938], Loss: 0.0351
Epoch [9/10], Step [700/938], Loss: 0.0014
Epoch [9/10], Step [800/938], Loss: 0.0042
Epoch [9/10], Step [900/938], Loss: 0.0138
Epoch [9/10], Train Loss: 0.0145, Train Acc: 99.56, Test Loss: 0.0135, Test Acc: 99.49
Epoch [10/10], Step [100/938], Loss: 0.0005
Epoch [10/10], Step [200/938], Loss: 0.0053
Epoch [10/10], Step [300/938], Loss: 0.0290
Epoch [10/10], Step [400/938], Loss: 0.0130
Epoch [10/10], Step [500/938], Loss: 0.0003
Epoch [10/10], Step [600/938], Loss: 0.0009
Epoch [10/10], Step [700/938], Loss: 0.0013
Epoch [10/10], Step [800/938], Loss: 0.0015
Epoch [10/10], Step [900/938], Loss: 0.0050
Epoch [10/10], Train Loss: 0.0145, Train Acc: 99.53, Test Loss: 0.0228, Test Acc: 99.29
```


Accuracy With Standard Normalization in Input and Batch Normalization in Output
Train
Test
99.5
99.0
98.5
98.0
97.5
97.0
96.5
0
2
4
6
8
Loss With Standard Normalization in Input and Batch Normalization in Output
Train
Test
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0
2
4
6
8

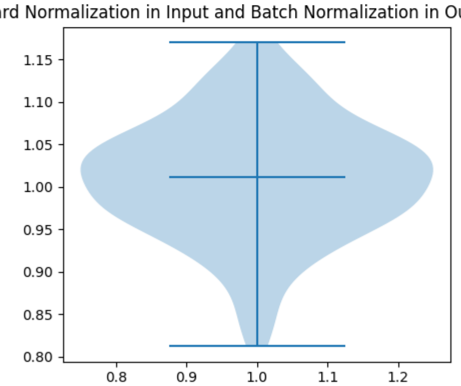
Standard Normalization in Input and Batch Normalization in Output Layer 1



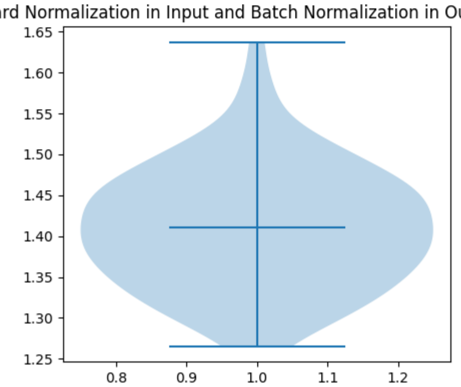
Standard Normalization in Input and Batch Normalization in Output Layer 2



Standard Normalization in Input and Batch Normalization in Output Layer 3



Standard Normalization in Input and Batch Normalization in Output Layer 4

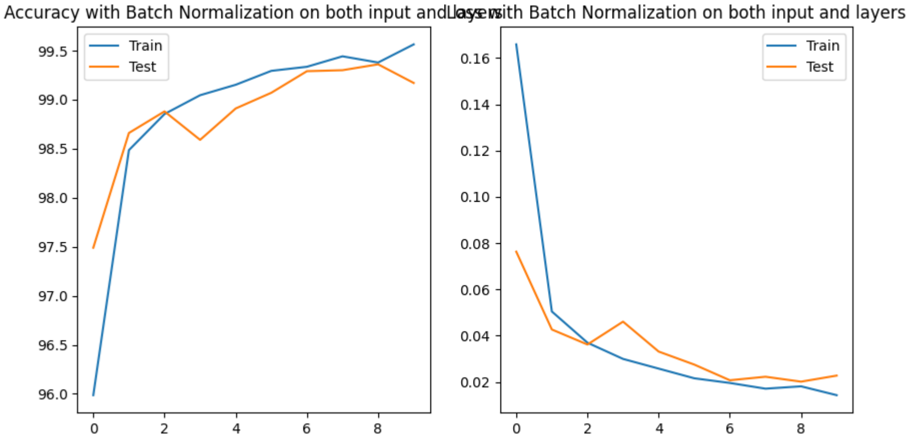


In [ ]:
```
#Solution 1.3):

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt

# Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transforms.ToTensor())

# Define the LeNet-5 model with batch normalization for all layers
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
```

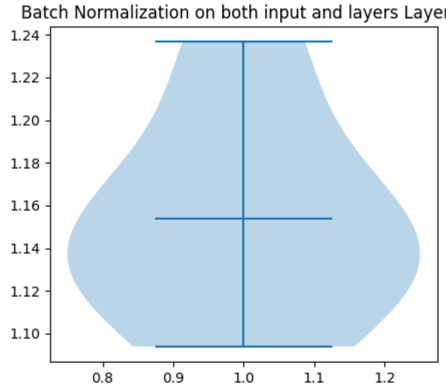
```
        self.bn1 = nn.BatchNorm2d(6)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.bn2 = nn.BatchNorm2d(16)
        self.fc1 = nn.Linear(16*4*4, 120)
        self.bn3 = nn.BatchNorm1d(120)
        self.fc2 = nn.Linear(120, 84)
        self.bn4 = nn.BatchNorm1d(84)
        self.fc3 = nn.Linear(84, 10)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        x = self.bn1(self.conv1(x))
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.bn2(self.conv2(x))
        x = self.relu(x)
        x = self.maxpool(x)
        x = x.view(-1, 16*4*4)
        x = self.bn3(self.fc1(x))
        x = self.relu(x)
        x = self.bn4(self.fc2(x))
        x = self.relu(x)
        x = self.fc3(x)
        return x

# Initialize the model
model = LeNet5()

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

# Define the batch size and number of epochs
batch_size = 64
n_epochs = 10

# Create data loaders for the train and test datasets
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Train the model
train_loss_bn = []
train_acc_bn = []
test_loss_bn = []
test_acc_bn = []

for epoch in range(n_epochs):
    model.train()
    train_loss = 0.0
    train_total = 0
    train_correct = 0
    for i, (inputs, labels) in enumerate(train_loader):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * inputs.size(0)
        _, predicted = torch.max(outputs.data, 1)
        train_total += labels.size(0)
        train_correct += (predicted == labels).sum().item()

        # Print training statistics
        if (i+1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch+1, n_epochs, i+1, len(train_loader), loss.item()))

    train_loss /= len(train_loader.dataset)
    train_acc = 100 * train_correct / train_total
    train_loss_bn.append(train_loss)
    train_acc_bn.append(train_acc)

    model.eval()
    test_loss = 0.0
    test_total = 0
    test_correct = 0
    for i, (inputs, labels) in enumerate(test_loader):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        test_loss += loss.item() * inputs.size(0)
        _, predicted = torch.max(outputs.data, 1)
        test_total += labels.size(0)
        test_correct += (predicted == labels).sum().item()

    test_loss /= len(test_loader.dataset)
    test_acc = 100 * test_correct / test_total
    test_loss_bn.append(test_loss)
    test_acc_bn.append(test_acc)

    print('Epoch [{}/{}], Train Loss: {:.4f}, Train Acc: {:.2f}, Test Loss: {:.4f}, Test Acc: {:.2f}'.format(epoch+1, n_epochs, train_loss, train_acc, test_loss, test_acc))

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(train_acc_bn, label='Train')
plt.plot(test_acc_bn, label='Test')
plt.title('Accuracy with Batch Normalization on both input and layers')
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(train_loss_bn, label='Train')
plt.plot(test_loss_bn, label='Test')
plt.title('Loss with Batch Normalization on both input and layers')
plt.legend()
plt.show()

bn_params = []
for name, module in model.named_modules():
    if isinstance(module, nn.BatchNorm2d) or isinstance(module, nn.BatchNorm1d):
        bn_params.append(module.weight.data.cpu().numpy())

fig, axs = plt.subplots(len(bn_params), figsize=(5, 20))
for i, p in enumerate(bn_params):
    axs[i].violinplot(dataset=p, showmeans=True)
```

```
    axs[i].set_title(f'Batch Normalization on both input and layers Layer {i+1}')
plt.show()
```

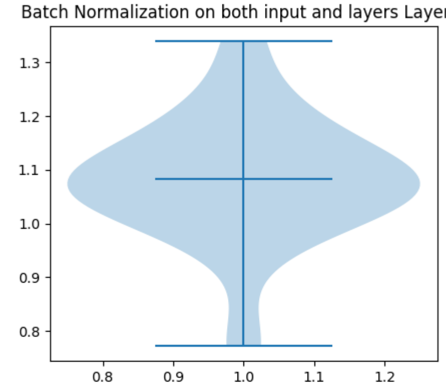
```
Epoch [1/10], Step [100/938], Loss: 0.3049
Epoch [1/10], Step [200/938], Loss: 0.2731
Epoch [1/10], Step [300/938], Loss: 0.0550
Epoch [1/10], Step [400/938], Loss: 0.0842
Epoch [1/10], Step [500/938], Loss: 0.0500
Epoch [1/10], Step [600/938], Loss: 0.0676
Epoch [1/10], Step [700/938], Loss: 0.0552
Epoch [1/10], Step [800/938], Loss: 0.0773
Epoch [1/10], Step [900/938], Loss: 0.0472
Epoch [1/10], Train Loss: 0.1659, Train Acc: 95.99, Test Loss: 0.0763, Test Acc: 97.49
Epoch [2/10], Step [100/938], Loss: 0.0335
Epoch [2/10], Step [200/938], Loss: 0.0354
Epoch [2/10], Step [300/938], Loss: 0.1013
Epoch [2/10], Step [400/938], Loss: 0.0679
Epoch [2/10], Step [500/938], Loss: 0.1258
Epoch [2/10], Step [600/938], Loss: 0.0113
Epoch [2/10], Step [700/938], Loss: 0.0941
Epoch [2/10], Step [800/938], Loss: 0.0423
Epoch [2/10], Step [900/938], Loss: 0.0226
Epoch [2/10], Train Loss: 0.0504, Train Acc: 98.48, Test Loss: 0.0426, Test Acc: 98.66
Epoch [3/10], Step [100/938], Loss: 0.0018
Epoch [3/10], Step [200/938], Loss: 0.1074
Epoch [3/10], Step [300/938], Loss: 0.0893
Epoch [3/10], Step [400/938], Loss: 0.0158
Epoch [3/10], Step [500/938], Loss: 0.0123
Epoch [3/10], Step [600/938], Loss: 0.0143
Epoch [3/10], Step [700/938], Loss: 0.0180
Epoch [3/10], Step [800/938], Loss: 0.0126
Epoch [3/10], Step [900/938], Loss: 0.0770
Epoch [3/10], Train Loss: 0.0369, Train Acc: 98.85, Test Loss: 0.0361, Test Acc: 98.88
Epoch [4/10], Step [100/938], Loss: 0.0076
Epoch [4/10], Step [200/938], Loss: 0.0528
Epoch [4/10], Step [300/938], Loss: 0.0206
Epoch [4/10], Step [400/938], Loss: 0.0393
Epoch [4/10], Step [500/938], Loss: 0.0433
Epoch [4/10], Step [600/938], Loss: 0.0172
Epoch [4/10], Step [700/938], Loss: 0.0065
Epoch [4/10], Step [800/938], Loss: 0.0089
Epoch [4/10], Step [900/938], Loss: 0.0316
Epoch [4/10], Train Loss: 0.0299, Train Acc: 99.05, Test Loss: 0.0460, Test Acc: 98.59
Epoch [5/10], Step [100/938], Loss: 0.0105
Epoch [5/10], Step [200/938], Loss: 0.0522
Epoch [5/10], Step [300/938], Loss: 0.0034
Epoch [5/10], Step [400/938], Loss: 0.0140
Epoch [5/10], Step [500/938], Loss: 0.0014
Epoch [5/10], Step [600/938], Loss: 0.0105
Epoch [5/10], Step [700/938], Loss: 0.0103
Epoch [5/10], Step [800/938], Loss: 0.1375
Epoch [5/10], Step [900/938], Loss: 0.0071
Epoch [5/10], Train Loss: 0.0257, Train Acc: 99.15, Test Loss: 0.0331, Test Acc: 98.91
Epoch [6/10], Step [100/938], Loss: 0.0278
Epoch [6/10], Step [200/938], Loss: 0.0595
Epoch [6/10], Step [300/938], Loss: 0.0068
Epoch [6/10], Step [400/938], Loss: 0.0056
Epoch [6/10], Step [500/938], Loss: 0.0286
Epoch [6/10], Step [600/938], Loss: 0.0067
Epoch [6/10], Step [700/938], Loss: 0.0701
Epoch [6/10], Step [800/938], Loss: 0.0054
Epoch [6/10], Step [900/938], Loss: 0.0988
Epoch [6/10], Train Loss: 0.0215, Train Acc: 99.29, Test Loss: 0.0275, Test Acc: 99.07
Epoch [7/10], Step [100/938], Loss: 0.0186
Epoch [7/10], Step [200/938], Loss: 0.0207
Epoch [7/10], Step [300/938], Loss: 0.0572
Epoch [7/10], Step [400/938], Loss: 0.0639
Epoch [7/10], Step [500/938], Loss: 0.0049
Epoch [7/10], Step [600/938], Loss: 0.0061
Epoch [7/10], Step [700/938], Loss: 0.0242
Epoch [7/10], Step [800/938], Loss: 0.0039
Epoch [7/10], Step [900/938], Loss: 0.0621
Epoch [7/10], Train Loss: 0.0195, Train Acc: 99.33, Test Loss: 0.0207, Test Acc: 99.29
Epoch [8/10], Step [100/938], Loss: 0.0008
Epoch [8/10], Step [200/938], Loss: 0.0040
Epoch [8/10], Step [300/938], Loss: 0.0059
Epoch [8/10], Step [400/938], Loss: 0.0494
Epoch [8/10], Step [500/938], Loss: 0.0005
Epoch [8/10], Step [600/938], Loss: 0.0012
Epoch [8/10], Step [700/938], Loss: 0.0065
Epoch [8/10], Step [800/938], Loss: 0.0041
Epoch [8/10], Step [900/938], Loss: 0.0165
Epoch [8/10], Train Loss: 0.0171, Train Acc: 99.44, Test Loss: 0.0222, Test Acc: 99.30
Epoch [9/10], Step [100/938], Loss: 0.0376
Epoch [9/10], Step [200/938], Loss: 0.0024
Epoch [9/10], Step [300/938], Loss: 0.0026
Epoch [9/10], Step [400/938], Loss: 0.0004
Epoch [9/10], Step [500/938], Loss: 0.0051
Epoch [9/10], Step [600/938], Loss: 0.0184
Epoch [9/10], Step [700/938], Loss: 0.0035
Epoch [9/10], Step [800/938], Loss: 0.0016
Epoch [9/10], Step [900/938], Loss: 0.0552
Epoch [9/10], Train Loss: 0.0181, Train Acc: 99.38, Test Loss: 0.0201, Test Acc: 99.36
Epoch [10/10], Step [100/938], Loss: 0.0005
Epoch [10/10], Step [200/938], Loss: 0.0011
Epoch [10/10], Step [300/938], Loss: 0.0054
Epoch [10/10], Step [400/938], Loss: 0.0445
Epoch [10/10], Step [500/938], Loss: 0.0022
Epoch [10/10], Step [600/938], Loss: 0.0043
Epoch [10/10], Step [700/938], Loss: 0.0094
Epoch [10/10], Step [800/938], Loss: 0.0021
Epoch [10/10], Step [900/938], Loss: 0.0012
Epoch [10/10], Train Loss: 0.0143, Train Acc: 99.56, Test Loss: 0.0227, Test Acc: 99.17
```


Accuracy with Batch Normalization on both input and layers
Train
Test
Loss with Batch Normalization on both input and layers
Train
Test

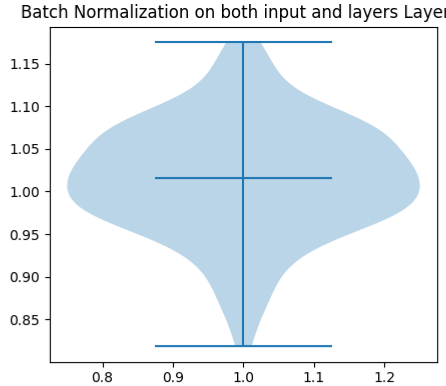
Batch Normalization on both input and layers Layer 1



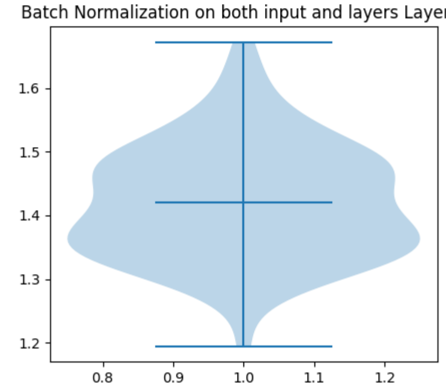
Batch Normalization on both input and layers Layer 2



Batch Normalization on both input and layers Layer 3



Batch Normalization on both input and layers Layer 4



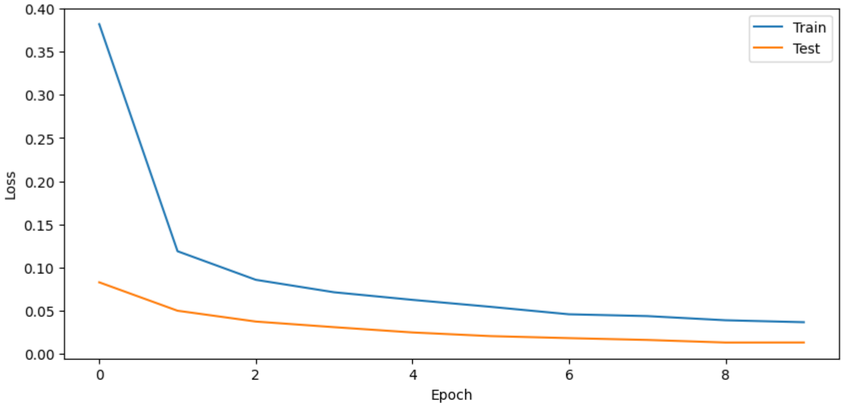
In [ ]:
```
#Solution 1.4):

#Drop out

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt

# Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transforms.ToTensor())

# Define the LeNet-5 model with dropout for all layers
class LeNet5(nn.Module):
    def __init__(self):
```

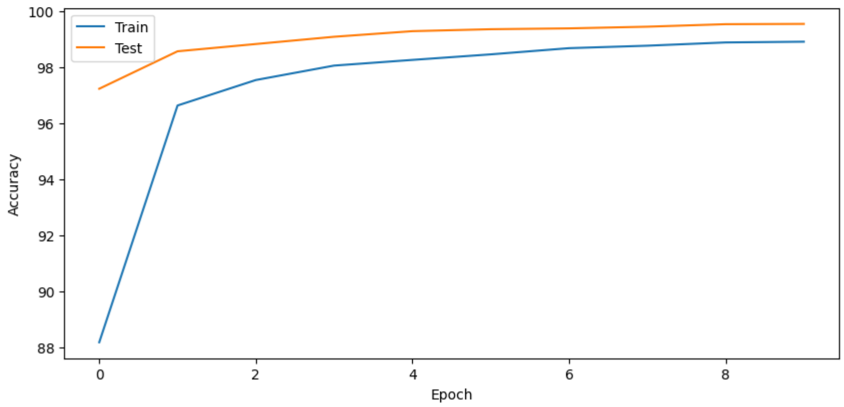
```
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.fc1 = nn.Linear(16*4*4, 120)
        self.dropout1 = nn.Dropout(p=0.2)
        self.fc2 = nn.Linear(120, 84)
        self.dropout2 = nn.Dropout(p=0.5)
        self.fc3 = nn.Linear(84, 10)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = x.view(-1, 16*4*4)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout1(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.dropout2(x)
        x = self.fc3(x)
        return x

# Initialize the model
model = LeNet5()

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

# Define the batch size and number of epochs
batch_size = 64
n_epochs = 10

# Create data loaders for the train and test datasets
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Train the model
train_loss_bn = []
train_acc_bn = []
test_loss_bn = []
test_acc_bn = []

for epoch in range(n_epochs):
    model.train()
    train_loss = 0.0
    train_total = 0
    train_correct = 0
    for i, (inputs, labels) in enumerate(train_loader):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * inputs.size(0)
        _, predicted = torch.max(outputs.data, 1)
        train_total += labels.size(0)
        train_correct += (predicted == labels).sum().item()

        # Print training statistics
        if (i+1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch+1, n_epochs, i+1, len(train_loader), loss.item()))

    train_loss /= len(train_loader.dataset)
    train_acc = 100 * train_correct / train_total
    train_loss_bn.append(train_loss)
    train_acc_bn.append(train_acc)

    model.eval()
    test_loss = 0.0
    test_total = 0
    test_correct = 0
    for i, (inputs, labels) in enumerate(test_loader):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        test_loss += loss.item() * inputs.size(0)
        _, predicted = torch.max(outputs.data, 1)
        test_total += labels.size(0)
        test_correct += (predicted == labels).sum().item()

    test_loss /= len(test_loader.dataset)
    test_acc = 100 * test_correct / test_total
    test_loss_bn.append(test_loss)
    test_acc_bn.append(test_acc)

    print('Epoch [{}/{}], Train Loss: {:.4f}, Train Acc: {:.2f}, Test Loss: {:.4f}, Test Acc: {:.2f}'.format(epoch+1, n_epochs, train_loss, train_acc, test_loss, test_acc))

# Plot the train/test loss and accuracy
fig, axs = plt.subplots(2, figsize=(10, 10))
axs[0].plot(train_loss_bn, label='Train')
axs[0].plot(test_loss_bn, label='Test')
axs[0].set_xlabel('Epoch')
axs[0].set_ylabel('Loss')
axs[0].legend()
axs[1].plot(train_acc_bn, label='Train')
axs[1].plot(test_acc_bn, label='Test')
axs[1].set_xlabel('Epoch')
axs[1].set_ylabel('Accuracy')
axs[1].legend()

# Plot the distribution of learned dropout parameters for each layer
fig, axs = plt.subplots(1, 4, figsize=(20, 5))
dropout_params = []
for i, module in enumerate(model.modules()):
    if isinstance(module, nn.Dropout):
        axs[i].violinplot(module.p.cpu().detach().numpy())
```

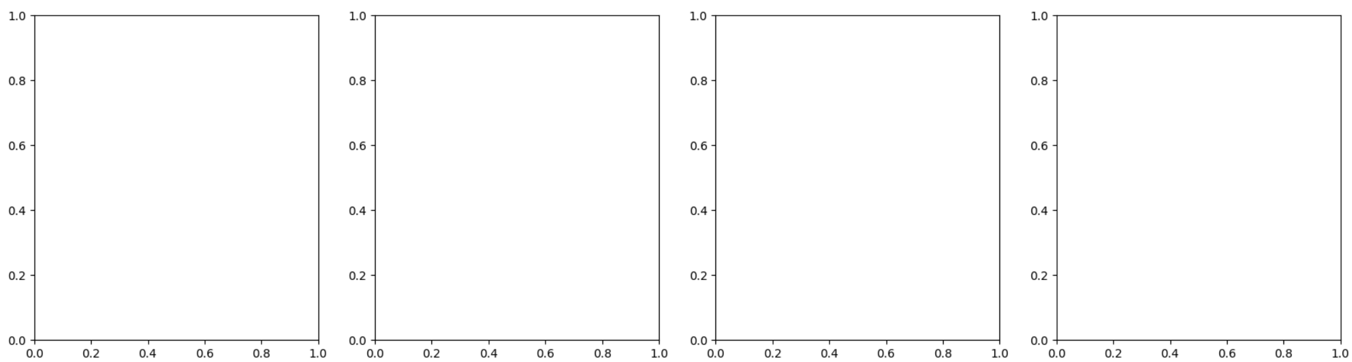
```
        axs[i].set_xticks([])
        axs[i].set_title(f'Layer {i+1}')
        dropout_params.append(module.p.cpu().detach().numpy())
plt.show()
```

```
Epoch [1/10], Step [100/938], Loss: 0.7948
Epoch [1/10], Step [200/938], Loss: 0.4855
Epoch [1/10], Step [300/938], Loss: 0.1772
Epoch [1/10], Step [400/938], Loss: 0.2756
Epoch [1/10], Step [500/938], Loss: 0.0792
Epoch [1/10], Step [600/938], Loss: 0.1046
Epoch [1/10], Step [700/938], Loss: 0.2825
Epoch [1/10], Step [800/938], Loss: 0.0977
Epoch [1/10], Step [900/938], Loss: 0.1902
Epoch [1/10], Train Loss: 0.3818, Train Acc: 88.17, Test Loss: 0.0831, Test Acc: 97.24
Epoch [2/10], Step [100/938], Loss: 0.1191
Epoch [2/10], Step [200/938], Loss: 0.0554
Epoch [2/10], Step [300/938], Loss: 0.1783
Epoch [2/10], Step [400/938], Loss: 0.0300
Epoch [2/10], Step [500/938], Loss: 0.1074
Epoch [2/10], Step [600/938], Loss: 0.0890
Epoch [2/10], Step [700/938], Loss: 0.0720
Epoch [2/10], Step [800/938], Loss: 0.0665
Epoch [2/10], Step [900/938], Loss: 0.0707
Epoch [2/10], Train Loss: 0.1191, Train Acc: 96.64, Test Loss: 0.0502, Test Acc: 98.58
Epoch [3/10], Step [100/938], Loss: 0.1632
Epoch [3/10], Step [200/938], Loss: 0.2067
Epoch [3/10], Step [300/938], Loss: 0.0597
Epoch [3/10], Step [400/938], Loss: 0.0557
Epoch [3/10], Step [500/938], Loss: 0.0580
Epoch [3/10], Step [600/938], Loss: 0.0668
Epoch [3/10], Step [700/938], Loss: 0.0263
Epoch [3/10], Step [800/938], Loss: 0.1739
Epoch [3/10], Step [900/938], Loss: 0.0503
Epoch [3/10], Train Loss: 0.0860, Train Acc: 97.55, Test Loss: 0.0377, Test Acc: 98.84
Epoch [4/10], Step [100/938], Loss: 0.0473
Epoch [4/10], Step [200/938], Loss: 0.0680
Epoch [4/10], Step [300/938], Loss: 0.1888
Epoch [4/10], Step [400/938], Loss: 0.0284
Epoch [4/10], Step [500/938], Loss: 0.0325
Epoch [4/10], Step [600/938], Loss: 0.0340
Epoch [4/10], Step [700/938], Loss: 0.0809
Epoch [4/10], Step [800/938], Loss: 0.0621
Epoch [4/10], Step [900/938], Loss: 0.0681
Epoch [4/10], Train Loss: 0.0716, Train Acc: 98.07, Test Loss: 0.0312, Test Acc: 99.10
Epoch [5/10], Step [100/938], Loss: 0.0624
Epoch [5/10], Step [200/938], Loss: 0.1523
Epoch [5/10], Step [300/938], Loss: 0.1516
Epoch [5/10], Step [400/938], Loss: 0.0276
Epoch [5/10], Step [500/938], Loss: 0.0258
Epoch [5/10], Step [600/938], Loss: 0.0453
Epoch [5/10], Step [700/938], Loss: 0.2150
Epoch [5/10], Step [800/938], Loss: 0.0410
Epoch [5/10], Step [900/938], Loss: 0.0585
Epoch [5/10], Train Loss: 0.0628, Train Acc: 98.27, Test Loss: 0.0251, Test Acc: 99.30
Epoch [6/10], Step [100/938], Loss: 0.0099
Epoch [6/10], Step [200/938], Loss: 0.0283
Epoch [6/10], Step [300/938], Loss: 0.0566
Epoch [6/10], Step [400/938], Loss: 0.0504
Epoch [6/10], Step [500/938], Loss: 0.0686
Epoch [6/10], Step [600/938], Loss: 0.0408
Epoch [6/10], Step [700/938], Loss: 0.0369
Epoch [6/10], Step [800/938], Loss: 0.0395
Epoch [6/10], Step [900/938], Loss: 0.0323
Epoch [6/10], Train Loss: 0.0547, Train Acc: 98.47, Test Loss: 0.0208, Test Acc: 99.37
Epoch [7/10], Step [100/938], Loss: 0.0985
Epoch [7/10], Step [200/938], Loss: 0.1369
Epoch [7/10], Step [300/938], Loss: 0.0536
Epoch [7/10], Step [400/938], Loss: 0.0841
Epoch [7/10], Step [500/938], Loss: 0.0151
Epoch [7/10], Step [600/938], Loss: 0.0088
Epoch [7/10], Step [700/938], Loss: 0.0855
Epoch [7/10], Step [800/938], Loss: 0.0126
Epoch [7/10], Step [900/938], Loss: 0.0082
Epoch [7/10], Train Loss: 0.0461, Train Acc: 98.69, Test Loss: 0.0185, Test Acc: 99.40
Epoch [8/10], Step [100/938], Loss: 0.0016
Epoch [8/10], Step [200/938], Loss: 0.0212
Epoch [8/10], Step [300/938], Loss: 0.0315
Epoch [8/10], Step [400/938], Loss: 0.0169
Epoch [8/10], Step [500/938], Loss: 0.2024
Epoch [8/10], Step [600/938], Loss: 0.1852
Epoch [8/10], Step [700/938], Loss: 0.0311
Epoch [8/10], Step [800/938], Loss: 0.0656
Epoch [8/10], Step [900/938], Loss: 0.0361
Epoch [8/10], Train Loss: 0.0440, Train Acc: 98.78, Test Loss: 0.0164, Test Acc: 99.46
Epoch [9/10], Step [100/938], Loss: 0.0286
Epoch [9/10], Step [200/938], Loss: 0.0319
Epoch [9/10], Step [300/938], Loss: 0.2100
Epoch [9/10], Step [400/938], Loss: 0.1601
Epoch [9/10], Step [500/938], Loss: 0.0027
Epoch [9/10], Step [600/938], Loss: 0.0210
Epoch [9/10], Step [700/938], Loss: 0.0756
Epoch [9/10], Step [800/938], Loss: 0.0059
Epoch [9/10], Step [900/938], Loss: 0.0035
Epoch [9/10], Train Loss: 0.0392, Train Acc: 98.90, Test Loss: 0.0134, Test Acc: 99.55
Epoch [10/10], Step [100/938], Loss: 0.0305
Epoch [10/10], Step [200/938], Loss: 0.1178
Epoch [10/10], Step [300/938], Loss: 0.0148
Epoch [10/10], Step [400/938], Loss: 0.0070
Epoch [10/10], Step [500/938], Loss: 0.0105
Epoch [10/10], Step [600/938], Loss: 0.0290
Epoch [10/10], Step [700/938], Loss: 0.1500
Epoch [10/10], Step [800/938], Loss: 0.0116
Epoch [10/10], Step [900/938], Loss: 0.0887
Epoch [10/10], Train Loss: 0.0369, Train Acc: 98.92, Test Loss: 0.0134, Test Acc: 99.56
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/home/karanvora/Documents/New York University/Classes/Semester 2/Introdution to High-Performance Machine Learning/Assignments/Assignment 3/kv2154_Assignment3_Problem1.ipynb
Cell 4 in 1
    <a href='vscode-notebook-cell:/home/karanvora/Documents/New%20York%20University/Classes/Semester%202/Introdution%20to%20High-Performance%20Machine%20Learning/Assignments/Assignment%203/kv2154_Assignment3_Problem1.ipynb#W3sZmlsZQ%3D%3D?line=131'>132</a> for i, module in enumerate(model.modules()):
    <a href='vscode-notebook-cell:/home/karanvora/Documents/New%20York%20University/Classes/Semester%202/Introdution%20to%20High-Performance%20Machine%20Learning/Assignments/Assignment%203/kv2154_Assignment3_Problem1.ipynb#W3sZmlsZQ%3D%3D?line=132'>133</a>     if isinstance(module, nn.Dropout):
--> <a href='vscode-notebook-cell:/home/karanvora/Documents/New%20York%20University/Classes/Semester%202/Introdution%20to%20High-Performance%20Machine%20Learning/Assignments/Assignment%203/kv2154_Assignment3_Problem1.ipynb#W3sZmlsZQ%3D%3D?line=133'>134</a>         axs[i].violinplot(module.p.cpu().detach().numpy())
    <a href='vscode-notebook-cell:/home/karanvora/Documents/New%20York%20University/Classes/Semester%202/Introdution%20to%20High-Performance%20Machine%20Learning/Assignments/Assignment%203/kv2154_Assignment3_Problem1.ipynb#W3sZmlsZQ%3D%3D?line=134'>135</a>         axs[i].set_xticks([])
    <a href='vscode-notebook-cell:/home/karanvora/Documents/New%20York%20University/Classes/Semester%202/Introdution%20to%20High-Performance%20Machine%20Learning/Assignments/Assignment%203/kv2154_Assignment3_Problem1.ipynb#W3sZmlsZQ%3D%3D?line=135'>136</a>         axs[i].set_title(f'Layer {i+1}')

IndexError: index 4 is out of bounds for axis 0 with size 4
```





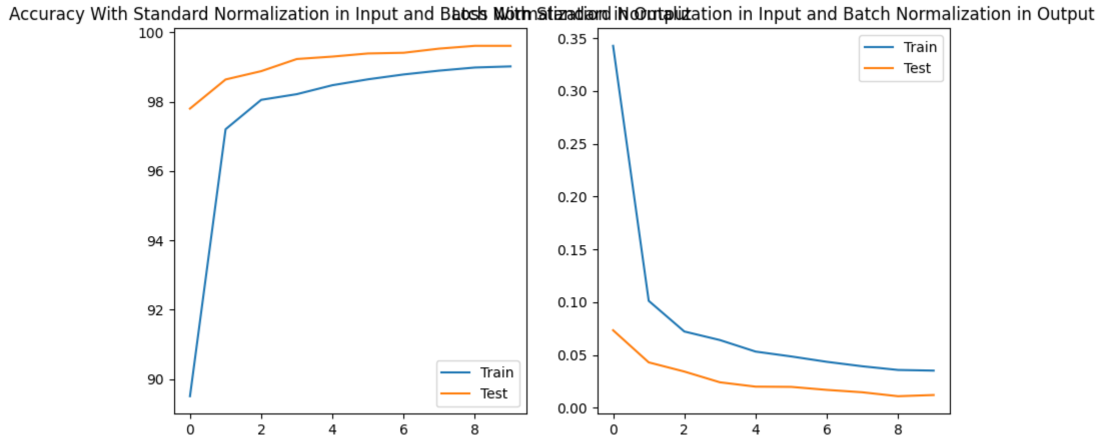


In [ ]:

```python
#Solution 1.5

#Solution 1.2):

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt

# Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transforms.Compose([
                                        transforms.ToTensor(),
                                        transforms.Normalize((0.5,), (0.5,))
                                    ]), download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transforms.Compose([
                                        transforms.ToTensor(),
                                        transforms.Normalize((0.5,), (0.5,))
                                    ]), download=True)

# Define the data loaders
batch_size = 64
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

```python
# Define the LeNet-5 model with batch normalization for all layers
class LeNet5BN(nn.Module):
    def __init__(self, num_classes=10):
        super(LeNet5BN, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1)
        self.bn1 = nn.BatchNorm2d(6)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1)
        self.bn2 = nn.BatchNorm2d(16)
        self.fc1 = nn.Linear(16*4*4, 120)
        self.bn3 = nn.BatchNorm1d(120)
        self.fc2 = nn.Linear(120, 84)
        self.bn4 = nn.BatchNorm1d(84)
        self.fc3 = nn.Linear(84, num_classes)
        self.dropout = nn.Dropout(p=0.5)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(self.bn1(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = self.conv2(x)
        x = F.relu(self.bn2(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = x.view(-1, 16*4*4)
        x = F.relu(self.bn3(self.fc1(x)))
        x = F.relu(self.bn4(self.fc2(x)))
        x = self.dropout(x)
        x = self.fc3(x)
        return x

# Initialize the model
model = LeNet5()

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

# Define the batch size and number of epochs
batch_size = 64
n_epochs = 10

# Create data loaders for the train and test datasets
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Train the model
train_loss_bn = []
train_acc_bn = []
test_loss_bn = []
test_acc_bn = []

for epoch in range(n_epochs):
    model.train()
    train_loss = 0.0
    train_total = 0
    train_correct = 0
    for i, (inputs, labels) in enumerate(train_loader):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * inputs.size(0)
        _, predicted = torch.max(outputs.data, 1)
        train_total += labels.size(0)
        train_correct += (predicted == labels).sum().item()

        # Print training statistics
        if (i+1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch+1, n_epochs, i+1, len(train_loader), loss.item()))

    train_loss /= len(train_loader.dataset)
    train_acc = 100 * train_correct / train_total
    train_loss_bn.append(train_loss)
    train_acc_bn.append(train_acc)

    model.eval()
    test_loss = 0.0
    test_total = 0
    test_correct = 0
    for i, (inputs, labels) in enumerate(test_loader):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        test_loss += loss.item() * inputs.size(0)
        _, predicted = torch.max(outputs.data, 1)
        test_total += labels.size(0)
        test_correct += (predicted == labels).sum().item()

    test_loss /= len(test_loader.dataset)
    test_acc = 100 * test_correct / test_total
    test_loss_bn.append(test_loss)
    test_acc_bn.append(test_acc)

    print('Epoch [{}/{}], Train Loss: {:.4f}, Train Acc: {:.2f}, Test Loss: {:.4f}, Test Acc: {:.2f}'.format(epoch+1, n_epochs, train_loss, train_acc, test_loss, test_acc))

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(train_acc_bn, label='Train')
plt.plot(test_acc_bn, label='Test')
plt.title('Accuracy With Standard Normalization in Input and Batch Normalization in Output')
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(train_loss_bn, label='Train')
plt.plot(test_loss_bn, label='Test')
plt.title('Loss With Standard Normalization in Input and Batch Normalization in Output')
plt.legend()
plt.show()
```

```
Epoch [1/10], Step [100/938], Loss: 0.6428
Epoch [1/10], Step [200/938], Loss: 0.1904
Epoch [1/10], Step [300/938], Loss: 0.2616
Epoch [1/10], Step [400/938], Loss: 0.2062
Epoch [1/10], Step [500/938], Loss: 0.0988
Epoch [1/10], Step [600/938], Loss: 0.1413
Epoch [1/10], Step [700/938], Loss: 0.0973
Epoch [1/10], Step [800/938], Loss: 0.1326
Epoch [1/10], Step [900/938], Loss: 0.2702
Epoch [1/10], Train Loss: 0.3428, Train Acc: 89.51, Test Loss: 0.0735, Test Acc: 97.80
Epoch [2/10], Step [100/938], Loss: 0.1540
Epoch [2/10], Step [200/938], Loss: 0.4550
Epoch [2/10], Step [300/938], Loss: 0.1096
Epoch [2/10], Step [400/938], Loss: 0.0733
Epoch [2/10], Step [500/938], Loss: 0.0988
Epoch [2/10], Step [600/938], Loss: 0.0900
Epoch [2/10], Step [700/938], Loss: 0.0786
Epoch [2/10], Step [800/938], Loss: 0.0221
Epoch [2/10], Step [900/938], Loss: 0.0706
Epoch [2/10], Train Loss: 0.1013, Train Acc: 97.20, Test Loss: 0.0431, Test Acc: 98.64
Epoch [3/10], Step [100/938], Loss: 0.0635
Epoch [3/10], Step [200/938], Loss: 0.1027
Epoch [3/10], Step [300/938], Loss: 0.0484
Epoch [3/10], Step [400/938], Loss: 0.1167
Epoch [3/10], Step [500/938], Loss: 0.0220
Epoch [3/10], Step [600/938], Loss: 0.0942
Epoch [3/10], Step [700/938], Loss: 0.1368
Epoch [3/10], Step [800/938], Loss: 0.0892
Epoch [3/10], Step [900/938], Loss: 0.0178
Epoch [3/10], Train Loss: 0.0723, Train Acc: 98.05, Test Loss: 0.0344, Test Acc: 98.88
Epoch [4/10], Step [100/938], Loss: 0.0315
Epoch [4/10], Step [200/938], Loss: 0.0512
Epoch [4/10], Step [300/938], Loss: 0.0458
Epoch [4/10], Step [400/938], Loss: 0.0847
Epoch [4/10], Step [500/938], Loss: 0.0328
Epoch [4/10], Step [600/938], Loss: 0.0369
Epoch [4/10], Step [700/938], Loss: 0.0266
Epoch [4/10], Step [800/938], Loss: 0.0252
Epoch [4/10], Step [900/938], Loss: 0.0774
Epoch [4/10], Train Loss: 0.0642, Train Acc: 98.22, Test Loss: 0.0242, Test Acc: 99.23
Epoch [5/10], Step [100/938], Loss: 0.0196
Epoch [5/10], Step [200/938], Loss: 0.0713
Epoch [5/10], Step [300/938], Loss: 0.0727
Epoch [5/10], Step [400/938], Loss: 0.0074
Epoch [5/10], Step [500/938], Loss: 0.2092
Epoch [5/10], Step [600/938], Loss: 0.0414
Epoch [5/10], Step [700/938], Loss: 0.0326
Epoch [5/10], Step [800/938], Loss: 0.0227
Epoch [5/10], Step [900/938], Loss: 0.0463
Epoch [5/10], Train Loss: 0.0533, Train Acc: 98.47, Test Loss: 0.0201, Test Acc: 99.30
Epoch [6/10], Step [100/938], Loss: 0.0145
Epoch [6/10], Step [200/938], Loss: 0.0320
Epoch [6/10], Step [300/938], Loss: 0.0150
Epoch [6/10], Step [400/938], Loss: 0.0228
Epoch [6/10], Step [500/938], Loss: 0.1365
Epoch [6/10], Step [600/938], Loss: 0.2312
Epoch [6/10], Step [700/938], Loss: 0.0144
Epoch [6/10], Step [800/938], Loss: 0.0183
Epoch [6/10], Step [900/938], Loss: 0.1622
Epoch [6/10], Train Loss: 0.0487, Train Acc: 98.64, Test Loss: 0.0199, Test Acc: 99.39
Epoch [7/10], Step [100/938], Loss: 0.0121
Epoch [7/10], Step [200/938], Loss: 0.0111
Epoch [7/10], Step [300/938], Loss: 0.0261
Epoch [7/10], Step [400/938], Loss: 0.0068
Epoch [7/10], Step [500/938], Loss: 0.0059
Epoch [7/10], Step [600/938], Loss: 0.0685
Epoch [7/10], Step [700/938], Loss: 0.0133
Epoch [7/10], Step [800/938], Loss: 0.0636
Epoch [7/10], Step [900/938], Loss: 0.0264
Epoch [7/10], Train Loss: 0.0436, Train Acc: 98.78, Test Loss: 0.0170, Test Acc: 99.41
Epoch [8/10], Step [100/938], Loss: 0.0266
Epoch [8/10], Step [200/938], Loss: 0.0381
Epoch [8/10], Step [300/938], Loss: 0.0589
Epoch [8/10], Step [400/938], Loss: 0.0236
Epoch [8/10], Step [500/938], Loss: 0.0520
Epoch [8/10], Step [600/938], Loss: 0.0138
Epoch [8/10], Step [700/938], Loss: 0.0013
Epoch [8/10], Step [800/938], Loss: 0.0370
Epoch [8/10], Step [900/938], Loss: 0.0008
Epoch [8/10], Train Loss: 0.0394, Train Acc: 98.89, Test Loss: 0.0148, Test Acc: 99.53
Epoch [9/10], Step [100/938], Loss: 0.0046
Epoch [9/10], Step [200/938], Loss: 0.0331
Epoch [9/10], Step [300/938], Loss: 0.0077
Epoch [9/10], Step [400/938], Loss: 0.0112
Epoch [9/10], Step [500/938], Loss: 0.0601
Epoch [9/10], Step [600/938], Loss: 0.0107
Epoch [9/10], Step [700/938], Loss: 0.0278
Epoch [9/10], Step [800/938], Loss: 0.0028
Epoch [9/10], Step [900/938], Loss: 0.0166
Epoch [9/10], Train Loss: 0.0359, Train Acc: 98.98, Test Loss: 0.0110, Test Acc: 99.61
Epoch [10/10], Step [100/938], Loss: 0.0008
Epoch [10/10], Step [200/938], Loss: 0.0011
Epoch [10/10], Step [300/938], Loss: 0.0248
Epoch [10/10], Step [400/938], Loss: 0.0269
Epoch [10/10], Step [500/938], Loss: 0.0175
Epoch [10/10], Step [600/938], Loss: 0.0076
Epoch [10/10], Step [700/938], Loss: 0.0062
Epoch [10/10], Step [800/938], Loss: 0.0579
Epoch [10/10], Step [900/938], Loss: 0.1801
Epoch [10/10], Train Loss: 0.0353, Train Acc: 99.02, Test Loss: 0.0122, Test Acc: 99.61
```



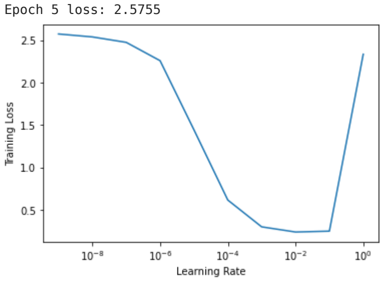

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/home/karanvora/Documents/New York University/Classes/Semester 2/Introdution to High-Performance Machine Learning/Assignments/Assignment 3/kv2154_Assignment3_Problem1.ipynb
Cell 5 in 1
    <a href='vscode-notebook-cell:/home/karanvora/Documents/New%20York%20University/Classes/Semester%202/Introdution%20to%20High-Performance%20Machine%20Learning/Assignment
s/Assignment%203/kv2154_Assignment3_Problem1.ipynb#W6sZmlsZQ%3D%3D?line=139'>140</a>     if isinstance(module, nn.BatchNorm2d) or isinstance(module, nn.BatchNorm1d):
    <a href='vscode-notebook-cell:/home/karanvora/Documents/New%20York%20University/Classes/Semester%202/Introdution%20to%20High-Performance%20Machine%20Learning/Assignment
s/Assignment%203/kv2154_Assignment3_Problem1.ipynb#W6sZmlsZQ%3D%3D?line=140'>141</a>         bn_params.append(module.weight.data.cpu().numpy())
--> <a href='vscode-notebook-cell:/home/karanvora/Documents/New%20York%20University/Classes/Semester%202/Introdution%20to%20High-Performance%20Machine%20Learning/Assignment
s/Assignment%203/kv2154_Assignment3_Problem1.ipynb#W6sZmlsZQ%3D%3D?line=142'>143</a> fig, axs = plt.subplots(len(bn_params), figsize=(5, 20))
    <a href='vscode-notebook-cell:/home/karanvora/Documents/New%20York%20University/Classes/Semester%202/Introdution%20to%20High-Performance%20Machine%20Learning/Assignment
s/Assignment%203/kv2154_Assignment3_Problem1.ipynb#W6sZmlsZQ%3D%3D?line=143'>144</a> for i, p in enumerate(bn_params):
    <a href='vscode-notebook-cell:/home/karanvora/Documents/New%20York%20University/Classes/Semester%202/Introdution%20to%20High-Performance%20Machine%20Learning/Assignment
s/Assignment%203/kv2154_Assignment3_Problem1.ipynb#W6sZmlsZQ%3D%3D?line=144'>145</a>     axs[i].violinplot(dataset=p, showmeans=True)

File ~/.local/lib/python3.10/site-packages/matplotlib/pyplot.py:1454, in subplots(nrows, ncols, sharex, sharey, squeeze, subplot_kw, gridspec_kw, **fig_kw)
   1321 """
   1322 Create a figure and a set of subplots.
   1323 
   (...)
   1451 
   1452 """
   1453 fig = figure(**fig_kw)
-> 1454 axs = fig.subplots(nrows=nrows, ncols=ncols, sharex=sharex, sharey=sharey,
   1455                    squeeze=squeeze, subplot_kw=subplot_kw,
   1456                    gridspec_kw=gridspec_kw)
   1457 return fig, axs

File ~/.local/lib/python3.10/site-packages/matplotlib/figure.py:896, in FigureBase.subplots(self, nrows, ncols, sharex, sharey, squeeze, subplot_kw, gridspec_kw)
    894 if gridspec_kw is None:
    895     gridspec_kw = {}
--> 896 gs = self.add_gridspec(nrows, ncols, figure=self, **gridspec_kw)
    897 axs = gs.subplots(sharex=sharex, sharey=sharey, squeeze=squeeze,
    898                   subplot_kw=subplot_kw)
    899 return axs

File ~/.local/lib/python3.10/site-packages/matplotlib/figure.py:1447, in FigureBase.add_gridspec(self, nrows, ncols, **kwargs)
   1408 """
   1409 Return a `.GridSpec` that has this figure as a parent.  This allows
   1410 complex layout of Axes in the figure.
   (...)
   1443 
   1444 """
   1446 _ = kwargs.pop('figure', None)  # pop in case user has added this...
-> 1447 gs = GridSpec(nrows=nrows, ncols=ncols, figure=self, **kwargs)
   1448 self._gridspecs.append(gs)
   1449 return gs

File ~/.local/lib/python3.10/site-packages/matplotlib/gridspec.py:385, in GridSpec.__init__(self, nrows, ncols, figure, left, bottom, right, top, wspace, hspace, width_rati
os, height_ratios)
    382 self.hspace = hspace
    383 self.figure = figure
--> 385 super().__init__(nrows, ncols,
    386                  width_ratios=width_ratios,
    387                  height_ratios=height_ratios)

File ~/.local/lib/python3.10/site-packages/matplotlib/gridspec.py:49, in GridSpecBase.__init__(self, nrows, ncols, height_ratios, width_ratios)
     34 """
     35 Parameters
     36 ----------
   (...)
     46     If not given, all rows will have the same height.
     47 """
     48 if not isinstance(nrows, Integral) or nrows <= 0:
---> 49     raise ValueError(
     50         f"Number of rows must be a positive integer, not {nrows!r}")
     51 if not isinstance(ncols, Integral) or ncols <= 0:
     52     raise ValueError(
     53         f"Number of columns must be a positive integer, not {ncols!r}")

ValueError: Number of rows must be a positive integer, not 0
<Figure size 500x2000 with 0 Axes>
```

In [1]:
```
# Solution 2.1):

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np


# Define the batch size
batch_size = 64

# Define the transformations to be applied to the images
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)),
    transforms.Resize((32,32))
])

# Load the fashionMNIST dataset
train_set = datasets.FashionMNIST('./data', train=True, download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True)

# ConvModule: a convolutional module in the above picture, consists a 2d convolutional layer, a 2d batchnorm layer, and a ReLU activation.
class ConvModule(nn.Module):
    def __init__(self, in_channels: int, out_channels: int, kernel_size, stride, padding='same'):
        super(ConvModule, self).__init__()
        self.conv2d = nn.Conv2d(
            in_channels, out_channels, kernel_size, stride=stride, padding=padding)

        self.batchnorm = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.conv2d(x)
        x = self.batchnorm(x)
        x = self.relu(x)

        return x

# InceptionModule: a inception module in the above picture, consists a convolution module with 1x1 filter,
# a convolution module with 3x3 filter, then concatenate these two outputs.
class InceptionModule(nn.Module):
    def __init__(self, in_channels, ch1x1, ch3x3):
        super(InceptionModule, self).__init__()

        self.conv1x1 = ConvModule(in_channels, ch1x1, (1, 1), 1)
        self.conv3x3 = ConvModule(in_channels, ch3x3, (3, 3), 1)

    def forward(self, x):
        out1 = self.conv1x1(x)
        out2 = self.conv3x3(x)
        x = torch.cat((out1, out2), 1)
        return x

# DownsampleModule: a downsample module in the above picture, consists a convolution module with 3x3 filter,
# a 2d maxpool layer, then concatenate these two outputs.
class DownsampleModule(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DownsampleModule, self).__init__()

        self.conv3x3 = ConvModule(in_channels, out_channels, (3, 3), (2, 2), padding='valid')
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2)

    def forward(self, x):
        out1 = self.conv3x3(x)
        out2 = self.maxpool(x)

        #return out1
        x = torch.cat((out1, out2), 1)

        return x


# MiniGoogLeNet: the MiniGoogLeNet model. Input: input_channels * 32 * 32.
# When input_channels is 1, the input is a grayscale image. When input_channels is 3, the input is a RGB image.
# Output: a tensor with the shape of [-1, classes], where classes it the number of classes.

class MiniGoogLeNet(nn.Module):
    def __init__(self, classes, input_channels):
        super(MiniGoogLeNet, self).__init__()

        self.conv1 = ConvModule(input_channels, 96, kernel_size=(3, 3), stride=1) # input_channel is 3 if you want to deal with RGB image, 1 for grey scale image
        self.inception1 = InceptionModule(96, 32, 32)
        self.inception2 = InceptionModule(32+32, 32, 48)
        self.downsample1 = DownsampleModule(32+48, 80)

        self.inception3 = InceptionModule(80+80, 112, 48)
        self.inception4 = InceptionModule(112+48, 96, 64)
        self.inception5 = InceptionModule(96+64, 80, 80)
        self.inception6 = InceptionModule(80+80, 48, 96)
        self.downsample2 = DownsampleModule(48+96, 96)

        self.inception7 = InceptionModule(96+96, 176, 160)
        self.inception8 = InceptionModule(176+160, 176, 160)
        self.avgpool2d = nn.AvgPool2d(kernel_size=7)
        self.dropout = nn.Dropout2d(0.5)

        self.fc = nn.Linear(240, classes)
        #self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        x = self.conv1(x)
        #print(x.shape)
        x = self.inception1(x)
        x = self.inception2(x)
        x = self.downsample1(x)

        x = self.inception3(x)
        x = self.inception4(x)
        x = self.inception5(x)
```

```
        x = self.inception6(x)
        x = self.downsample2(x)

        x = self.avgpool2d(x)
        x = self.dropout(x)

        x = torch.flatten(x, 1)
        x = self.fc(x)
        #x = self.softmax(x), no need for softmax because PyTorch Cross Entropy Loss implemented softmax

        return x

# Define the learning rate candidates
learning_rate_candidates = [10**(-i) for i in range(10)]

# Define the learning rate candidates
learning_rate_candidates = [10**(-i) for i in range(10)]

# Train the model for 5 epochs for each learning rate candidate and store the results
losses = []
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
for lr in learning_rate_candidates:
    print(f'Training with learning rate {lr}')
    model = MiniGoogLeNet(10, 1).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
    for epoch in range(5):
        running_loss = 0.0
        for i, data in enumerate(train_loader, 0):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        epoch_loss = running_loss / len(train_loader)
        print(f'Epoch {epoch+1} loss: {epoch_loss:.4f}')
    losses.append(epoch_loss)

# Plot the losses as a function of the learning rate
plt.plot(learning_rate_candidates, losses)
plt.xscale('log')
plt.xlabel('Learning Rate')
plt.ylabel('Training Loss')
plt.show()

min_loss_index = losses.index(min(losses))
max_loss_index = losses.index(max(losses))
lrmin = learning_rate_candidates[min_loss_index]
lrmax = learning_rate_candidates[max_loss_index]
print(f'lrmin: {lrmin}')
print(f'lrmax: {lrmax}')
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz
  0%|          | 0/26421880 [00:00<?, ?it/s]
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
  0%|          | 0/29515 [00:00<?, ?it/s]
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
  0%|          | 0/4422102 [00:00<?, ?it/s]
Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
  0%|          | 0/5148 [00:00<?, ?it/s]
```

```
Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Training with learning rate 1
Epoch 1 loss: 2.4475
Epoch 2 loss: 2.3377
Epoch 3 loss: 2.3393
Epoch 4 loss: 2.3403
Epoch 5 loss: 2.3368
Training with learning rate 0.1
Epoch 1 loss: 0.6276
Epoch 2 loss: 0.3635
Epoch 3 loss: 0.3105
Epoch 4 loss: 0.2753
Epoch 5 loss: 0.2508
Training with learning rate 0.01
Epoch 1 loss: 0.5694
Epoch 2 loss: 0.3545
Epoch 3 loss: 0.2952
Epoch 4 loss: 0.2661
Epoch 5 loss: 0.2406
Training with learning rate 0.001
Epoch 1 loss: 0.8012
Epoch 2 loss: 0.4481
Epoch 3 loss: 0.3725
Epoch 4 loss: 0.3299
Epoch 5 loss: 0.3014
Training with learning rate 0.0001
Epoch 1 loss: 1.5608
Epoch 2 loss: 0.9811
Epoch 3 loss: 0.7810
Epoch 4 loss: 0.6795
Epoch 5 loss: 0.6175
Training with learning rate 1e-05
Epoch 1 loss: 2.2330
Epoch 2 loss: 1.8995
Epoch 3 loss: 1.6914
Epoch 4 loss: 1.5528
Epoch 5 loss: 1.4458
Training with learning rate 1e-06
Epoch 1 loss: 2.5570
Epoch 2 loss: 2.4561
Epoch 3 loss: 2.3776
Epoch 4 loss: 2.3096
Epoch 5 loss: 2.2606
Training with learning rate 1e-07
Epoch 1 loss: 2.5137
Epoch 2 loss: 2.5013
Epoch 3 loss: 2.5007
Epoch 4 loss: 2.4887
Epoch 5 loss: 2.4767
Training with learning rate 1e-08
Epoch 1 loss: 2.5444
Epoch 2 loss: 2.5491
Epoch 3 loss: 2.5499
Epoch 4 loss: 2.5433
Epoch 5 loss: 2.5408
Training with learning rate 1e-09
Epoch 1 loss: 2.5761
Epoch 2 loss: 2.5769
Epoch 3 loss: 2.5747
Epoch 4 loss: 2.5764
Epoch 5 loss: 2.5755
```



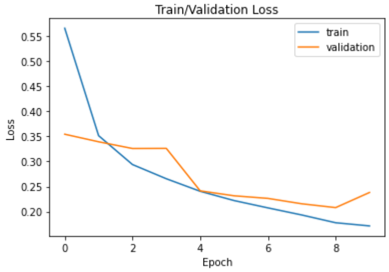

```
lrmin: 0.01
lrmax: 1e-09
```

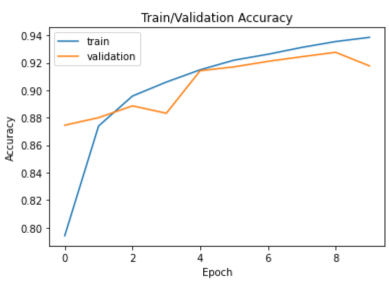
In [3]:
```
#Solution 2.2):

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np


# Define the batch size
batch_size = 64

# Define the transformations to be applied to the images
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)),
    transforms.Resize((32,32))
])

# Load the fashionMNIST dataset
train_set = datasets.FashionMNIST('./data', train=True, download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True)

test_set = datasets.FashionMNIST('./data', train=False, download=True, transform=transform)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, shuffle=True)

# ConvModule: a convolutional module in the above picture, consists a 2d convolutional layer, a 2d batchnorm layer, and a ReLU activation.
class ConvModule(nn.Module):
    def __init__(self, in_channels: int, out_channels: int, kernel_size, stride, padding='same'):
        super(ConvModule, self).__init__()
        self.conv2d = nn.Conv2d(
            in_channels, out_channels, kernel_size, stride=stride, padding=padding)
```

```
        self.batchnorm = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.conv2d(x)
        x = self.batchnorm(x)
        x = self.relu(x)

        return x

# InceptionModule: a inception module in the above picture, consists a convolution module with 1x1 filter,
# a convolution module with 3x3 filter, then concatenate these two outputs.
class InceptionModule(nn.Module):
    def __init__(self, in_channels, ch1x1, ch3x3):
        super(InceptionModule, self).__init__()

        self.conv1x1 = ConvModule(in_channels, ch1x1, (1, 1), 1)
        self.conv3x3 = ConvModule(in_channels, ch3x3, (3, 3), 1)

    def forward(self, x):
        out1 = self.conv1x1(x)
        out2 = self.conv3x3(x)
        x = torch.cat((out1, out2), 1)
        return x

# DownsampleModule: a downsample module in the above picture, consists a convolution module with 3x3 filter,
# a 2d maxpool layer, then concatenate these two outputs.
class DownsampleModule(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DownsampleModule, self).__init__()

        self.conv3x3 = ConvModule(in_channels, out_channels, (3, 3), (2, 2), padding='valid')
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2)

    def forward(self, x):
        out1 = self.conv3x3(x)
        out2 = self.maxpool(x)

        #return out1
        x = torch.cat((out1, out2), 1)

        return x


# MiniGoogLeNet: the MiniGoogLeNet model. Input: input_channels * 32 * 32.
# When input_channels is 1, the input is a grayscale image. When input_channels is 3, the input is a RGB image.
# Output: a tensor with the shape of [-1, classes], where classes it the number of classes.

class MiniGoogLeNet(nn.Module):
    def __init__(self, classes, input_channels):
        super(MiniGoogLeNet, self).__init__()

        self.conv1 = ConvModule(input_channels, 96, kernel_size=(3, 3), stride=1) # input_channel is 3 if you want to deal with RGB image, 1 for grey scale image
        self.inception1 = InceptionModule(96, 32, 32)
        self.inception2 = InceptionModule(32+32, 32, 48)
        self.downsample1 = DownsampleModule(32+48, 80)

        self.inception3 = InceptionModule(80+80, 112, 48)
        self.inception4 = InceptionModule(112+48, 96, 64)
        self.inception5 = InceptionModule(96+64, 80, 80)
        self.inception6 = InceptionModule(80+80, 48, 96)
        self.downsample2 = DownsampleModule(48+96, 96)

        self.inception7 = InceptionModule(96+96, 176, 160)
        self.inception8 = InceptionModule(176+160, 176, 160)
        self.avgpool2d = nn.AvgPool2d(kernel_size=7)
        self.dropout = nn.Dropout2d(0.5)

        self.fc = nn.Linear(240, classes)
        #self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        x = self.conv1(x)
        #print(x.shape)
        x = self.inception1(x)
        x = self.inception2(x)
        x = self.downsample1(x)

        x = self.inception3(x)
        x = self.inception4(x)
        x = self.inception5(x)
        x = self.inception6(x)
        x = self.downsample2(x)

        x = self.avgpool2d(x)
        x = self.dropout(x)

        x = torch.flatten(x, 1)
        x = self.fc(x)
        #x = self.softmax(x), no need for softmax because PyTorch Cross Entropy Loss implemented softmax

        return x

lrmin = 0.01
lrmax = 1e-09
step_size = 2000
mode = 'exp_range'

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = MiniGoogLeNet(10, 1).to(device)

# Create the cyclic learning rate scheduler
optimizer = optim.SGD(model.parameters(), lr=lrmax, momentum=0.9)
scheduler = optim.lr_scheduler.CyclicLR(optimizer, base_lr=lrmin,  max_lr=lrmax, step_size_up=step_size, mode=mode, gamma=0.99)
criterion = nn.CrossEntropyLoss()

# Train the model for 10 epochs using the cyclic learning rate policy
train_losses = []
train_accs = []
val_losses = []
val_accs = []

for epoch in range(10):
    model.train()
```

```
    running_loss = 0.0
    correct = 0
    total = 0
    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        scheduler.step()  # update the learning rate
        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    epoch_train_loss = running_loss / len(train_loader)
    epoch_train_acc = correct / total
    train_losses.append(epoch_train_loss)
    train_accs.append(epoch_train_acc)
    print(f'Train Epoch {epoch+1} loss: {epoch_train_loss:.4f} accuracy: {epoch_train_acc:.4f}')

    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    epoch_val_loss = running_loss / len(test_loader)
    epoch_val_acc = correct / total
    val_losses.append(epoch_val_loss)
    val_accs.append(epoch_val_acc)
    print(f'Validation Epoch {epoch+1} loss: {epoch_val_loss:.4f} accuracy: {epoch_val_acc:.4f}')

# Plot the train/validation loss curve
plt.plot(train_losses, label='train')
plt.plot(val_losses, label='validation')
plt.title('Train/Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plot the train/validation accuracy curve
plt.plot(train_accs, label='train')
plt.plot(val_accs, label='validation')
plt.title('Train/Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
Train Epoch 1 loss: 0.5649 accuracy: 0.7940
Validation Epoch 1 loss: 0.3541 accuracy: 0.8745
Train Epoch 2 loss: 0.3512 accuracy: 0.8739
Validation Epoch 2 loss: 0.3390 accuracy: 0.8800
Train Epoch 3 loss: 0.2937 accuracy: 0.8958
Validation Epoch 3 loss: 0.3256 accuracy: 0.8886
Train Epoch 4 loss: 0.2655 accuracy: 0.9059
Validation Epoch 4 loss: 0.3259 accuracy: 0.8832
Train Epoch 5 loss: 0.2405 accuracy: 0.9149
Validation Epoch 5 loss: 0.2414 accuracy: 0.9142
Train Epoch 6 loss: 0.2221 accuracy: 0.9219
Validation Epoch 6 loss: 0.2316 accuracy: 0.9170
Train Epoch 7 loss: 0.2075 accuracy: 0.9262
Validation Epoch 7 loss: 0.2264 accuracy: 0.9210
Train Epoch 8 loss: 0.1934 accuracy: 0.9312
Validation Epoch 8 loss: 0.2157 accuracy: 0.9244
Train Epoch 9 loss: 0.1779 accuracy: 0.9355
Validation Epoch 9 loss: 0.2080 accuracy: 0.9276
Train Epoch 10 loss: 0.1716 accuracy: 0.9385
Validation Epoch 10 loss: 0.2382 accuracy: 0.9177
```





In [ ]:
```python
# Solution 2.3 Test):

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torch.utils.data as data
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
import copy
import math
import traceback

torch.cuda.empty_cache()
# Define the transformations to be applied to the images
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)),
    transforms.Resize((32,32))
])

# Load the fashionMNIST dataset
train_set = datasets.FashionMNIST('./data', train=True, download=True, transform=transform)

# ConvModule: a convolutional module in the above picture, consists a 2d convolutional layer, a 2d batchnorm layer, and a ReLU activation.
class ConvModule(nn.Module):
    def __init__(self, in_channels: int, out_channels: int, kernel_size, stride, padding='same'):
        super(ConvModule, self).__init__()
        self.conv2d = nn.Conv2d(
            in_channels, out_channels, kernel_size, stride=stride, padding=padding)

        self.batchnorm = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.conv2d(x)
        x = self.batchnorm(x)
        x = self.relu(x)

        return x

# InceptionModule: a inception module in the above picture, consists a convolution module with 1x1 filter,
# a convolution module with 3x3 filter, then concatenate these two outputs.
class InceptionModule(nn.Module):
    def __init__(self, in_channels, ch1x1, ch3x3):
        super(InceptionModule, self).__init__()

        self.conv1x1 = ConvModule(in_channels, ch1x1, (1, 1), 1)
        self.conv3x3 = ConvModule(in_channels, ch3x3, (3, 3), 1)

    def forward(self, x):
        out1 = self.conv1x1(x)
        out2 = self.conv3x3(x)
        x = torch.cat((out1, out2), 1)
        return x

# DownsampleModule: a downsample module in the above picture, consists a convolution module with 3x3 filter,
# a 2d maxpool layer, then concatenate these two outputs.
class DownsampleModule(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DownsampleModule, self).__init__()

        self.conv3x3 = ConvModule(in_channels, out_channels, (3, 3), (2, 2), padding='valid')
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2)

    def forward(self, x):
        out1 = self.conv3x3(x)
        out2 = self.maxpool(x)

        #return out1
        x = torch.cat((out1, out2), 1)

        return x


# MiniGoogLeNet: the MiniGoogLeNet model. Input: input_channels * 32 * 32.
# When input_channels is 1, the input is a grayscale image. When input_channels is 3, the input is a RGB image.
# Output: a tensor with the shape of [-1, classes], where classes it the number of classes.

class MiniGoogLeNet(nn.Module):
    def __init__(self, classes, input_channels):
        super(MiniGoogLeNet, self).__init__()

        self.conv1 = ConvModule(input_channels, 96, kernel_size=(3, 3), stride=1) # input_channel is 3 if you want to deal with RGB image, 1 for grey scale image
        self.inception1 = InceptionModule(96, 32, 32)
        self.inception2 = InceptionModule(32+32, 32, 48)
        self.downsample1 = DownsampleModule(32+48, 80)

        self.inception3 = InceptionModule(80+80, 112, 48)
        self.inception4 = InceptionModule(112+48, 96, 64)
        self.inception5 = InceptionModule(96+64, 80, 80)
        self.inception6 = InceptionModule(80+80, 48, 96)
        self.downsample2 = DownsampleModule(48+96, 96)

        self.inception7 = InceptionModule(96+96, 176, 160)
        self.inception8 = InceptionModule(176+160, 176, 160)
        self.avgpool2d = nn.AvgPool2d(kernel_size=7)
        self.dropout = nn.Dropout2d(0.5)

        self.fc = nn.Linear(240, classes)
        #self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        x = self.conv1(x)
        #print(x.shape)
        x = self.inception1(x)
        x = self.inception2(x)
        x = self.downsample1(x)

        x = self.inception3(x)
        x = self.inception4(x)
        x = self.inception5(x)
```

```
        x = self.inception6(x)
        x = self.downsample2(x)

        x = self.avgpool2d(x)
        x = self.dropout(x)

        x = torch.flatten(x, 1)
        x = self.fc(x)
        #x = self.softmax(x), no need for softmax because PyTorch Cross Entropy Loss implemented softmax

        return x

lr = 1e-09
BatchSize = [32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384]
# BatchSize = [32, 64, 128, 256, 512, 1024, 2048]
VALID_RATIO = 0.9
batch_size_to_loss = {}

def calculate_accuracy(y_pred,y):
    top_pred=y_pred.argmax(1,keepdim= True)
    correct =top_pred.eq(y.view_as(top_pred)).sum()
    acc=correct.float()/y.shape[0]
    return acc

def train(model, iterator, optimizer, criterion, device):
    epoch_loss = 0
    epoch_acc = 0
    model.train()
    for (x, y) in iterator:
        x = x.to(device)
        y = y.to(device)
        optimizer.zero_grad()
        y_pred = model(x)
        loss = criterion(y_pred, y)
        acc = calculate_accuracy(y_pred, y)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        epoch_acc += acc.item()
    return epoch_loss / len(iterator), epoch_acc / len(iterator)

def evaluate(model, iterator, criterion, device):
    epoch_loss = 0
    epoch_acc = 0
    model.eval()
    with torch.no_grad():
        for (x, y) in iterator:
            x = x.to(device)
            y = y.to(device)
            y_pred= model(x)
            loss = criterion(y_pred, y)
            acc = calculate_accuracy(y_pred, y)
            epoch_loss += loss.item()
            epoch_acc += acc.item()
    return epoch_loss / len(iterator), epoch_acc / len(iterator)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
for batchSize in BatchSize:
    torch.cuda.empty_cache()
    print(f"Batch Size: {batchSize}")
    torch.cuda.empty_cache()
    transformations = transforms.Compose([transforms.Resize((32, 32)),
                                transforms.ToTensor(),
                               ])
    training = datasets.FashionMNIST(root='./data', train=True, download=True, transform=transformations)
    n_train_examples = int(len(training) * VALID_RATIO)
    n_valid_examples = len(training) - n_train_examples
    t, v = data.random_split(training, [n_train_examples, n_valid_examples])
    v = copy.deepcopy(v)
    train_set = torch.utils.data.DataLoader(t, batch_size=batchSize, shuffle=True)
    valid_set = torch.utils.data.DataLoader(v, batch_size=batchSize, shuffle=True)
    model = MiniGoogLeNet(classes=10, input_channels=1).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
    best_loss=float('inf')

    for epoch in range(5):
      print("Current ecoch: ", epoch)
      train_loss,train_acc = train(model,train_set,optimizer,criterion,device)
      val_loss,val_acc = evaluate(model,valid_set,criterion,device)
      print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
      print(f'\tValidation Loss: {val_loss:.3f} | Validation Acc: {val_acc*100:.2f}%')
      if val_loss < best_loss:
        best_loss=val_loss
    batch_size_to_loss[batchSize] = best_loss
```

```
Batch Size: 32
Current ecoch:  0
        Train Loss: 0.644 | Train Acc: 76.40%
        Validation Loss: 0.558 | Validation Acc: 81.04%
Current ecoch:  1
        Train Loss: 0.388 | Train Acc: 85.91%
        Validation Loss: 0.329 | Validation Acc: 87.84%
Current ecoch:  2
        Train Loss: 0.322 | Train Acc: 88.59%
        Validation Loss: 0.308 | Validation Acc: 89.12%
Current ecoch:  3
        Train Loss: 0.283 | Train Acc: 89.98%
        Validation Loss: 0.260 | Validation Acc: 90.75%
Current ecoch:  4
        Train Loss: 0.259 | Train Acc: 90.70%
        Validation Loss: 0.229 | Validation Acc: 91.95%
Current ecoch:  5
        Train Loss: 0.238 | Train Acc: 91.49%
        Validation Loss: 0.255 | Validation Acc: 90.64%
Current ecoch:  6
        Train Loss: 0.225 | Train Acc: 92.05%
        Validation Loss: 0.212 | Validation Acc: 92.53%
Current ecoch:  7
        Train Loss: 0.209 | Train Acc: 92.53%
        Validation Loss: 0.419 | Validation Acc: 86.61%
Current ecoch:  8
        Train Loss: 0.196 | Train Acc: 93.07%
        Validation Loss: 0.202 | Validation Acc: 92.67%
Current ecoch:  9
        Train Loss: 0.185 | Train Acc: 93.38%
        Validation Loss: 0.261 | Validation Acc: 90.92%
Batch Size: 64
Current ecoch:  0
        Train Loss: 0.631 | Train Acc: 77.26%
        Validation Loss: 0.414 | Validation Acc: 84.62%
Current ecoch:  1
        Train Loss: 0.385 | Train Acc: 86.22%
        Validation Loss: 0.345 | Validation Acc: 86.93%
Current ecoch:  2
        Train Loss: 0.321 | Train Acc: 88.59%
        Validation Loss: 0.286 | Validation Acc: 89.76%
Current ecoch:  3
        Train Loss: 0.281 | Train Acc: 89.96%
        Validation Loss: 0.275 | Validation Acc: 90.00%
Current ecoch:  4
        Train Loss: 0.255 | Train Acc: 91.05%
        Validation Loss: 0.247 | Validation Acc: 91.04%
Current ecoch:  5
        Train Loss: 0.240 | Train Acc: 91.55%
        Validation Loss: 0.243 | Validation Acc: 91.33%
Current ecoch:  6
        Train Loss: 0.224 | Train Acc: 92.03%
        Validation Loss: 0.236 | Validation Acc: 91.47%
Current ecoch:  7
        Train Loss: 0.208 | Train Acc: 92.65%
        Validation Loss: 0.201 | Validation Acc: 92.85%
Current ecoch:  8
        Train Loss: 0.197 | Train Acc: 92.94%
        Validation Loss: 0.201 | Validation Acc: 92.53%
Current ecoch:  9
        Train Loss: 0.188 | Train Acc: 93.26%
        Validation Loss: 0.214 | Validation Acc: 92.45%
Batch Size: 128
Current ecoch:  0
        Train Loss: 0.643 | Train Acc: 76.19%
        Validation Loss: 0.574 | Validation Acc: 79.48%
Current ecoch:  1
        Train Loss: 0.385 | Train Acc: 86.25%
        Validation Loss: 0.383 | Validation Acc: 85.82%
Current ecoch:  2
        Train Loss: 0.318 | Train Acc: 88.60%
        Validation Loss: 0.314 | Validation Acc: 88.75%
Current ecoch:  3
        Train Loss: 0.287 | Train Acc: 89.81%
        Validation Loss: 0.260 | Validation Acc: 90.48%
Current ecoch:  4
        Train Loss: 0.254 | Train Acc: 90.99%
        Validation Loss: 0.248 | Validation Acc: 90.61%
Current ecoch:  5
        Train Loss: 0.235 | Train Acc: 91.75%
        Validation Loss: 0.218 | Validation Acc: 91.70%
Current ecoch:  6
        Train Loss: 0.220 | Train Acc: 92.19%
        Validation Loss: 0.224 | Validation Acc: 92.03%
Current ecoch:  7
        Train Loss: 0.208 | Train Acc: 92.58%
        Validation Loss: 0.224 | Validation Acc: 92.08%
Current ecoch:  8
        Train Loss: 0.196 | Train Acc: 92.95%
        Validation Loss: 0.219 | Validation Acc: 92.51%
Current ecoch:  9
        Train Loss: 0.185 | Train Acc: 93.46%
        Validation Loss: 0.198 | Validation Acc: 93.00%
Batch Size: 256
Current ecoch:  0
        Train Loss: 0.628 | Train Acc: 77.12%
        Validation Loss: 0.399 | Validation Acc: 84.94%
Current ecoch:  1
        Train Loss: 0.375 | Train Acc: 86.57%
        Validation Loss: 0.306 | Validation Acc: 88.48%
Current ecoch:  2
        Train Loss: 0.317 | Train Acc: 88.63%
        Validation Loss: 0.282 | Validation Acc: 90.28%
Current ecoch:  3
        Train Loss: 0.282 | Train Acc: 89.87%
        Validation Loss: 0.270 | Validation Acc: 90.19%
Current ecoch:  4
        Train Loss: 0.256 | Train Acc: 90.93%
        Validation Loss: 0.242 | Validation Acc: 91.37%
Current ecoch:  5
        Train Loss: 0.239 | Train Acc: 91.48%
        Validation Loss: 0.222 | Validation Acc: 92.02%
Current ecoch:  6
        Train Loss: 0.221 | Train Acc: 92.12%
        Validation Loss: 0.242 | Validation Acc: 90.77%
```

```
Current ecoch:  7
        Train Loss: 0.208 | Train Acc: 92.57%
        Validation Loss: 0.215 | Validation Acc: 92.21%
Current ecoch:  8
        Train Loss: 0.196 | Train Acc: 93.04%
        Validation Loss: 0.205 | Validation Acc: 92.68%
Current ecoch:  9
        Train Loss: 0.182 | Train Acc: 93.49%
        Validation Loss: 0.220 | Validation Acc: 92.36%
Batch Size: 512
Current ecoch:  0
        Train Loss: 0.632 | Train Acc: 77.02%
        Validation Loss: 0.421 | Validation Acc: 83.26%
Current ecoch:  1
        Train Loss: 0.386 | Train Acc: 86.16%
        Validation Loss: 0.318 | Validation Acc: 88.53%
Current ecoch:  2
        Train Loss: 0.327 | Train Acc: 88.31%
        Validation Loss: 0.271 | Validation Acc: 89.96%
Current ecoch:  3
        Train Loss: 0.288 | Train Acc: 89.61%
        Validation Loss: 0.311 | Validation Acc: 89.19%
Current ecoch:  4
        Train Loss: 0.264 | Train Acc: 90.53%
        Validation Loss: 0.277 | Validation Acc: 89.63%
Current ecoch:  5
        Train Loss: 0.242 | Train Acc: 91.31%
        Validation Loss: 0.237 | Validation Acc: 91.72%
Current ecoch:  6
        Train Loss: 0.224 | Train Acc: 92.03%
        Validation Loss: 0.211 | Validation Acc: 92.51%
Current ecoch:  7
        Train Loss: 0.208 | Train Acc: 92.57%
        Validation Loss: 0.214 | Validation Acc: 92.26%
Current ecoch:  8
        Train Loss: 0.199 | Train Acc: 92.88%
        Validation Loss: 0.202 | Validation Acc: 92.73%
Current ecoch:  9
        Train Loss: 0.190 | Train Acc: 93.30%
        Validation Loss: 0.214 | Validation Acc: 92.20%
Batch Size: 1024
Current ecoch:  0
        Train Loss: 0.646 | Train Acc: 76.43%
        Validation Loss: 0.435 | Validation Acc: 83.52%
Current ecoch:  1
        Train Loss: 0.386 | Train Acc: 86.10%
        Validation Loss: 0.361 | Validation Acc: 87.22%
Current ecoch:  2
        Train Loss: 0.320 | Train Acc: 88.52%
        Validation Loss: 0.280 | Validation Acc: 89.54%
Current ecoch:  3
        Train Loss: 0.284 | Train Acc: 89.77%
        Validation Loss: 0.289 | Validation Acc: 90.05%
Current ecoch:  4
        Train Loss: 0.259 | Train Acc: 90.71%
        Validation Loss: 0.278 | Validation Acc: 89.47%
Current ecoch:  5
        Train Loss: 0.240 | Train Acc: 91.51%
        Validation Loss: 0.223 | Validation Acc: 91.93%
Current ecoch:  6
        Train Loss: 0.223 | Train Acc: 92.05%
        Validation Loss: 0.231 | Validation Acc: 91.32%
Current ecoch:  7
        Train Loss: 0.211 | Train Acc: 92.53%
        Validation Loss: 0.210 | Validation Acc: 92.07%
Current ecoch:  8
        Train Loss: 0.196 | Train Acc: 92.97%
        Validation Loss: 0.195 | Validation Acc: 92.60%
Current ecoch:  9
        Train Loss: 0.186 | Train Acc: 93.47%
        Validation Loss: 0.196 | Validation Acc: 93.04%
Batch Size: 2048
Current ecoch:  0
        Train Loss: 0.635 | Train Acc: 76.71%
        Validation Loss: 0.404 | Validation Acc: 85.68%
Current ecoch:  1
        Train Loss: 0.384 | Train Acc: 86.31%
        Validation Loss: 0.305 | Validation Acc: 88.94%
Current ecoch:  2
        Train Loss: 0.325 | Train Acc: 88.45%
        Validation Loss: 0.316 | Validation Acc: 89.17%
Current ecoch:  3
        Train Loss: 0.288 | Train Acc: 89.38%
        Validation Loss: 0.242 | Validation Acc: 91.32%
Current ecoch:  4
        Train Loss: 0.258 | Train Acc: 90.74%
        Validation Loss: 0.233 | Validation Acc: 91.49%
Current ecoch:  5
        Train Loss: 0.241 | Train Acc: 91.42%
        Validation Loss: 0.252 | Validation Acc: 90.20%
Current ecoch:  6
        Train Loss: 0.223 | Train Acc: 92.03%
        Validation Loss: 0.212 | Validation Acc: 92.38%
Current ecoch:  7
        Train Loss: 0.209 | Train Acc: 92.48%
        Validation Loss: 0.221 | Validation Acc: 92.08%
Current ecoch:  8
        Train Loss: 0.199 | Train Acc: 92.85%
        Validation Loss: 0.207 | Validation Acc: 92.54%
Current ecoch:  9
        Train Loss: 0.186 | Train Acc: 93.32%
        Validation Loss: 0.210 | Validation Acc: 92.48%
Batch Size: 4096
Current ecoch:  0
        Train Loss: 0.659 | Train Acc: 75.85%
        Validation Loss: 0.433 | Validation Acc: 83.73%
Current ecoch:  1
        Train Loss: 0.389 | Train Acc: 85.99%
        Validation Loss: 0.383 | Validation Acc: 85.73%
Current ecoch:  2
        Train Loss: 0.320 | Train Acc: 88.43%
        Validation Loss: 0.287 | Validation Acc: 89.52%
Current ecoch:  3
        Train Loss: 0.280 | Train Acc: 89.92%
        Validation Loss: 0.295 | Validation Acc: 89.50%
```

```
Current ecoch:  4
        Train Loss: 0.256 | Train Acc: 90.93%
        Validation Loss: 0.233 | Validation Acc: 91.78%
Current ecoch:  5
        Train Loss: 0.234 | Train Acc: 91.62%
        Validation Loss: 0.250 | Validation Acc: 91.10%
Current ecoch:  6
        Train Loss: 0.221 | Train Acc: 92.16%
        Validation Loss: 0.224 | Validation Acc: 92.10%
Current ecoch:  7
        Train Loss: 0.206 | Train Acc: 92.73%
        Validation Loss: 0.204 | Validation Acc: 92.49%
Current ecoch:  8
        Train Loss: 0.197 | Train Acc: 92.99%
        Validation Loss: 0.212 | Validation Acc: 91.90%
Current ecoch:  9
        Train Loss: 0.186 | Train Acc: 93.38%
        Validation Loss: 0.191 | Validation Acc: 93.12%
Batch Size: 8192
Current ecoch:  0
        Train Loss: 0.626 | Train Acc: 77.26%
        Validation Loss: 0.391 | Validation Acc: 86.04%
Current ecoch:  1
        Train Loss: 0.381 | Train Acc: 86.21%
        Validation Loss: 0.370 | Validation Acc: 87.18%
Current ecoch:  2
        Train Loss: 0.317 | Train Acc: 88.74%
        Validation Loss: 0.539 | Validation Acc: 80.90%
Current ecoch:  3
        Train Loss: 0.282 | Train Acc: 89.98%
        Validation Loss: 0.257 | Validation Acc: 91.40%
Current ecoch:  4
        Train Loss: 0.254 | Train Acc: 91.00%
        Validation Loss: 0.240 | Validation Acc: 91.36%
Current ecoch:  5
        Train Loss: 0.235 | Train Acc: 91.59%
        Validation Loss: 0.233 | Validation Acc: 91.46%
Current ecoch:  6
        Train Loss: 0.221 | Train Acc: 92.24%
        Validation Loss: 0.214 | Validation Acc: 92.25%
Current ecoch:  7
        Train Loss: 0.206 | Train Acc: 92.61%
        Validation Loss: 0.212 | Validation Acc: 92.13%
Current ecoch:  8
        Train Loss: 0.196 | Train Acc: 93.06%
        Validation Loss: 0.204 | Validation Acc: 92.68%
Current ecoch:  9
        Train Loss: 0.184 | Train Acc: 93.34%
        Validation Loss: 0.210 | Validation Acc: 92.40%
Batch Size: 16384
Current ecoch:  0
        Train Loss: 0.647 | Train Acc: 76.25%
        Validation Loss: 0.436 | Validation Acc: 83.96%
Current ecoch:  1
        Train Loss: 0.395 | Train Acc: 85.73%
        Validation Loss: 0.388 | Validation Acc: 86.21%
Current ecoch:  2
        Train Loss: 0.324 | Train Acc: 88.40%
        Validation Loss: 0.294 | Validation Acc: 89.35%
Current ecoch:  3
        Train Loss: 0.286 | Train Acc: 89.76%
        Validation Loss: 0.285 | Validation Acc: 89.26%
Current ecoch:  4
        Train Loss: 0.257 | Train Acc: 90.76%
        Validation Loss: 0.249 | Validation Acc: 91.47%
Current ecoch:  5
        Train Loss: 0.240 | Train Acc: 91.49%
        Validation Loss: 0.225 | Validation Acc: 91.82%
Current ecoch:  6
        Train Loss: 0.222 | Train Acc: 92.18%
        Validation Loss: 0.238 | Validation Acc: 91.35%
Current ecoch:  7
        Train Loss: 0.210 | Train Acc: 92.60%
        Validation Loss: 0.212 | Validation Acc: 92.44%
Current ecoch:  8
        Train Loss: 0.198 | Train Acc: 92.93%
        Validation Loss: 0.263 | Validation Acc: 89.94%
Current ecoch:  9
        Train Loss: 0.184 | Train Acc: 93.40%
        Validation Loss: 0.203 | Validation Acc: 92.71%
```

In [ ]:

```
def plot_losses_batch_version(losses):
    plt.plot(list(losses.keys()), list(losses.values()))
    plt.xlabel('Batch Size')
    plt.ylabel('Loss Value')
    plt.title('Loss vs Batch Size')
    plt.show()

plot_losses_batch_version(batch_size_to_loss)
```

Karan Vora (kv2154)
ECE-GY 9143 Introduction to High-Performance Machine Learning Assignment 3

Problem 3):

Solution 3.1):

AlexNet was one of the first CNN to win the ImageNet classification competition of 2012. It consists of 8 layers: 5 Convolutional and 3 Fully-Connected layers with a 1000 class classification as the output layer.

→ Convolutional layer 1:
Input: 227 x 227 image with 3 input channels
96 filters of size 11 x 11 with stride 4 and no padding
Number of parameters: (11 x 11 x 3 x 96) + 96 = 34944

→ Max-Pooling layer 1:
Input: 96 channels with 55 x 55 feature maps
Max-Pooling of size 3 x 3 with stride 2

→ Convolutional layer 2:
Input: 96 channels with 27 x 27 feature maps
256 filters of size 5 x 5 with stride 1 and padding 2
Number of parameters: (5 x 5 x 96 x 256) + 256 = 614656

→ Max-Pooling layer 2:
Input: 256 channels with 13 x 13 feature maps
Max-Pooling pooling of size 3 x 3 with stride 2

→ Convolutional layer 3:
Input: 256 channels with 13 x 13 feature maps
384 filters of size 3 x 3 with stride 1 and padding 1
Number of parameters: (3 x 3 x 256 x 384) + 384 = 885120

→ Convolutional layer 4:
Input: 384 channels with 13 x 13 feature maps
384 filters of size 3 x 3 with stride 1 and padding 1
Number of parameters: (3 x 3 x 384 x 384) + 384 = 1327488

→ Convolutional layer 5:
Input: 384 channels with 13 x 13 feature maps
256 filters of size 3 x 3 with stride 1 and padding 1
Number of parameters: (3 x 3 x 384 x 256) + 256 = 884992

→ Max-Pooling layer 3:
Input: 256 channels with 13 x 13 feature maps
Max-Pooling of size 3 x 3 and stride 2

→ Fully-Connected layer 1:

Input: 9216 (256 x 6 x 6) features
4096 Neurons
Number of parameters: (9216 x 4096) + 4096 = 37752832
→ Fully-Connected layer 2:
Input: 4096 features
4096 neurons
Number of parameters: (4096 x 4096) + 4096 = 16781312

→ Fully-Connected layer 3 (Output layer):
Input: 4096 features
1000 neurons, one for each class in ImageNet dataset
Number of parameters: (4096 x 1000) + 1000 = 4097000

Total number of parameters in AlexNet: 34944 + 614656 + 885120 + 1327488 + 884992 + 37752832 + 16781312 + 4097000 = 61100344

Solution 3.2):

Layer	Number of Activations (Memory)	Parameters (Compute)
Input	224x224x3 = 150K	0
CONV3-64	224x224x64 = 3.2M	(3x3x3)x64 = 1728
CONV3-64	224x224x64 = 3.2M	(3x3x3)x64 = 36864
POOL2	112x112x64 = 800K	0
CONV3-128	112x112x128 = 1.6M	(3x3x64)x128 = 73728
CONV3-128	112x112x128 = 1.6M	(3x3x128)x128 = 147456
POOL2	56x56x128 = 400K	0
CONV3-256	56x56x256 = 800K	(3x3x128)x256 = 294912
CONV3-256	56x56x256 = 800K	(3x3x256)x256 = 589824
CONV3-256	56x56x256 = 800K	(3x3x256)x256 = 589824
CONV3-256	56x56x256 = 800K	(3x3x256)x256 = 589824
POOL2	28x28x256 = 200K	0
CONV3-512	28x28x512 = 400K	(3x3x256)x512 = 1179648
CONV3-512	28x28x512 = 400K	(3x3x512)x512 = 2359296
CONV3-512	28x28x512 = 400K	(3x3x512)x512 = 2359296
CONV3-512	28x28x512 = 400K	(3x3x512)x512 = 2359296
POOL2	14x14x512 = 100K	0
CONV3-512	14x14x512 = 100K	(3x3x512)x512 = 2359296
CONV3-512	14x14x512 = 100K	(3x3x512)x512 = 2359296
CONV3-512	14x14x512 = 100K	(3x3x512)x512 = 2359296
CONV3-512	14x14x512 = 100K	(3x3x512)x512 = 2359296
POOL2	7x7x512 = 25K	0
FC	4096	7x7x512x4096 = 102760448
FC	4096	4096x4096 = 16777216
FC	1000	4096x1000 = 4096000
Total	17144296	143653144

Solution 3.3):

==> For Naive Inception Module,

→ For 1x1 Filter,
Number of Operations = 32 * 32 * 1 * 1 * 128 * 256 = 1048576

→ For 3x3 Filter,
Each 3 x 3 filter operates on all 256 channels of the input volume, which has dimensions of 32 x 32 x 256. The output volume for each filter will be of size 30 x 30 x 1, with the height and width reduced by 2 due to the filter size
Number of Operations = 30 * 30 * 3 * 3 * 192 * 256 = 11940096000

→ For 5x5 Filter,
Each 5 x 5 filter operates on all 256 channels of the input volume, which has dimensions of 32 x 32 x 256. The output volume for each filter will be of size 28 x 28 x 1, with the height and width reduced by 4 due to the filter size
Number of Operations = 28 * 28 * 5 * 5 * 96 * 256 = 5806893760

Total Number of Operations = 1048576 + 11940096000 + 5806893760 = 17748038336

==> For Inception Module with Dimension reduction

→ For 1x1 Filter,
Number of Operations = 32 * 32 * 1 * 1 * 128 * 256 = 1048576

Next,
→ For 1x1 Filter,
Number of Operations = 32 * 32 * 1 * 1 * 128 * 256 = 1048576
→ For the next layer, The output dimensions are 28 x 28 x 128, so for filter size of 3x3
Number of Operations = 30 * 30 * 3 * 3 * 128 * 192 = 199065600

Next,
→ For 1x1 Filter,
Number of Operations = 32 * 32 * 1 * 1 * 32 * 256 = 8388608
→ For the next layer, The output dimensions are 30 x 30 x 32, so for filter size of 5x5
Number of Operations = 28 * 28 * 5 * 5 * 96 * 32 = 60211200

Next,
→ We have a 3x3 max-pooling layer, so the input dimensions of 32 x 32 x 256 will be reduced to 30 x 30 x 256.
→ For next layer the output dimension is 30 x 30 x 64 so for filter size of 1x1
Number of Operations = 30 * 30 * 1 * 1 * 64 * 256 = 14745600

Total Number of Operations = 1048576 + 1048576 + 199065600 + 8388608 + 60211200 + 14745600 = 284508160

From the above mentioned calculation, it is clear that Dimensionality Reduction reduces the required number of operations to perform the inception module by a large factor

Solution 3.4):

Naive architectures for convolutional neural networks (CNNs) typically stack multiple convolutional layers with high numbers of filters to extract features from the input image. However, this approach can lead to two problems:

1. High computational cost: As the number of filters increases in each convolutional layer, the number of parameters and computations required also increases. This can make the model slow and computationally expensive.

2. Information loss: As the input volume passes through multiple convolutional layers, the spatial dimensions reduce while the depth increases. This can lead to a loss of information and may result in the network missing important features.

To address these problems, dimensionality reduction architectures, such as the inception module, have been proposed. Inception modules use multiple filter sizes in parallel to extract features from the input volume at different scales. By doing this, they can capture both fine-grained and coarse-grained features in the input volume.

Specifically, inception modules use 1x1, 3x3, and 5x5 filters in parallel and concatenate their outputs to form the final output of the module. The 1x1 filters are used to reduce the number of input channels and, hence, reduce the computational cost of the subsequent filters. This is known as a bottleneck layer. Additionally, max-pooling is applied before the 1x1 filters to reduce the spatial dimensions of the input volume, which further reduces the computational cost.

By using multiple filter sizes and dimensionality reduction techniques, inception modules can extract features from the input volume in a more efficient and effective way. The use of 1x1 filters for dimensionality reduction significantly reduces the number of computations required, while the use of multiple filter sizes helps capture both fine-grained and coarse-grained features.

The computational saving of the inception module depends on the specific architecture and input volume size, but it can be significant. In some cases, the use of dimensionality reduction architectures like inception modules can reduce the number of computations required by up to 10 times compared to naive architectures with the same number of parameters. This reduction in computational cost makes the model faster and more efficient, which is important for real-world applications with limited computing resources.