Prob 5.

```
import torch
import torch.nn as nn
from torch.optim import Optimizer
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import transforms
import matplotlib.pyplot as plt
from random import shuffle

'''
Step 1: (same step)
'''
# Use data with only 4 and 9 as labels: which is hardest to classify
label_1, label_2 = 4, 9

# MNIST training data
train_set = datasets.MNIST(root='./mnist_data/', train=True,
transform=transforms.ToTensor(), download=True)

# Use data with two labels
idx = (train_set.targets == label_1) + (train_set.targets == label_2)
train_set.data = train_set.data[idx]
train_set.targets = train_set.targets[idx]
train_set.targets[train_set.targets == label_1] = -1
train_set.targets[train_set.targets == label_2] = 1

# MNIST testing data
test_set = datasets.MNIST(root='./mnist_data/', train=False,
transform=transforms.ToTensor())

# Use data with two labels
idx = (test_set.targets == label_1) + (test_set.targets == label_2)
test_set.data = test_set.data[idx]
test_set.targets = test_set.targets[idx]
test_set.targets[test_set.targets == label_1] = -1
test_set.targets[test_set.targets == label_2] = 1


'''
Step 2: (same step)
'''
class LR(nn.Module) :
    '''
    Initialize model
        input_dim : dimension of given input data
    '''
    # MNIST data is 28x28 images
```

```
    def __init__(self, input_dim=28*28) :
        super().__init__()
        self.linear = nn.Linear(input_dim, 1, bias=False)

    ''' forward given input x '''
    def forward(self, x) :
        return self.linear(x.float().view(-1, 28*28))


'''
Step 3: (same step)
'''
model_logistic = LR()                                   # Define a
Neural Network Model
model_sum_of_square = LR()                              # Define a
Neural Network Model

def logistic_loss(output, target):
    return -torch.nn.functional.logsigmoid(target*output)
def sum_of_square_loss(output, target):
    return 0.5*(1-target)*((1-torch.sigmoid(-
output))**2+torch.sigmoid(output)**2) + 0.5*(1+target)*((1-
torch.sigmoid(output))**2+torch.sigmoid(-output)**2)

logistic_loss_function = logistic_loss
# Specify loss function
sum_of_square_loss_function = sum_of_square_loss
# Specify loss function
logistic_optimizer = torch.optim.SGD(model_logistic.parameters(),
lr=255*1e-4)   # specify SGD with learning rate
sum_of_square_optimizer =
torch.optim.SGD(model_sum_of_square.parameters(), lr=255*1e-4)   #
specify SGD with learning rate



'''
Step 4: Train model with SGD (LOOK HERE)
'''
train_loader = DataLoader(dataset=train_set, batch_size=1,
shuffle=True)

import time
start = time.time()
# Train the model for 3 epochs
for epoch in range(3) :
    for image, label in train_loader :
        # Clear previously computed gradient
        logistic_optimizer.zero_grad()
        sum_of_square_optimizer.zero_grad()
```

```
        # then compute gradient with forward and backward passes
        logistic_train_loss =
logistic_loss_function(model_logistic(image), label.float())
        sum_of_square_train_loss =
sum_of_square_loss_function(model_sum_of_square(image), label.float())

        logistic_train_loss.backward()
        sum_of_square_train_loss.backward()

        # perform SGD step (parameter update)
        logistic_optimizer.step()
        sum_of_square_optimizer.step()
end = time.time()
print(f"Time ellapsed in training is: {end-start}")


'''
Step 5: (same step)
'''
logistic_test_loss, logistic_correct = 0, 0
sum_of_square_test_loss, sum_of_square_correct = 0, 0

# Test data
test_loader = DataLoader(dataset=test_set, batch_size=1,
shuffle=False)
# no need to shuffle test data

# Evaluate accuracy using test data
for ind, (image, label) in enumerate(test_loader) :

    # Forward pass
    logistic_output = model_logistic(image)
    sum_of_square_output = model_sum_of_square(image)

    # Calculate cumulative loss
    logistic_test_loss += logistic_loss_function(logistic_output,
label.float()).item()
    sum_of_square_test_loss +=
sum_of_square_loss_function(sum_of_square_output,
label.float()).item()

    # Make a prediction
    if logistic_output.item() * label.item() >= 0 :
        logistic_correct += 1
    if sum_of_square_output.item() * label.item() >= 0 :
        sum_of_square_correct += 1


# Print out the results
```

```
print("logistic loss:")
print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\
n'.format(
        logistic_test_loss /len(test_loader), logistic_correct,
len(test_loader),
        100. * logistic_correct / len(test_loader)))
print("sum of square loss:")
print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\
n'.format(
        sum_of_square_test_loss /len(test_loader),
sum_of_square_correct, len(test_loader),
        100. * sum_of_square_correct / len(test_loader)))
```

```
Time ellapsed in training is: 34.90736937522888
logistic loss:
[Test set] Average loss: 0.0875, Accuracy: 1926/1991 (96.74%)

sum of square loss:
[Test set] Average loss: 0.0482, Accuracy: 1933/1991 (97.09%)
```

The sum of square loss function is defined as follows, and this case handles the loss function without using the if else statement. The two models are trained on the same dataset with same suffled SGD and are validated in the same testset. This ablation study on only using the loss function gives the result that the sum of square loss gives a little bit more accuracy gain than the logistic loss, but the difference is negligable.

Prob. 7

```
import torch
import torch.nn as nn
from torch.optim import Optimizer
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import transforms

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

'''
Step 1:
'''

# MNIST dataset
train_dataset = datasets.MNIST(root='./mnist_data/',
                               train=True,
                               transform=transforms.ToTensor(),
                               download=True)

test_dataset = datasets.MNIST(root='./mnist_data/',
                              train=False,
```

```
                              transform=transforms.ToTensor())


'''
Step 2: LeNet5
'''

# Modern LeNet uses this layer for C3
class C3_layer_full(nn.Module):
    def __init__(self):
        super(C3_layer_full, self).__init__()
        self.conv_layer = nn.Conv2d(6, 16, kernel_size=5)

    def forward(self, x):
        return self.conv_layer(x)

# Original LeNet uses this layer for C3
class C3_layer(nn.Module):
    def __init__(self):
        super(C3_layer, self).__init__()
        self.ch_in_3 = [[0, 1, 2],
                        [1, 2, 3],
                        [2, 3, 4],
                        [3, 4, 5],
                        [0, 4, 5],
                        [0, 1, 5]] # filter with 3 subset of input
channels
        self.ch_in_4 = [[0, 1, 2, 3],
                        [1, 2, 3, 4],
                        [2, 3, 4, 5],
                        [0, 3, 4, 5],
                        [0, 1, 4, 5],
                        [0, 1, 2, 5],
                        [0, 1, 3, 4],
                        [1, 2, 4, 5],
                        [0, 2, 3, 5]] # filter with 4 subset of input
channels
        self.ch_in_6 = [[0, 1, 2, 3, 4, 5]] # filter with all input
channels

        self.conv_layer_3 = nn.ModuleList([nn.Conv2d(3, 1,
kernel_size=5) for _ in range(6)])
        self.conv_layer_4 = nn.ModuleList([nn.Conv2d(4, 1,
kernel_size=5) for _ in range(9)])
        self.conv_layer_6 = nn.Conv2d(6, 1, kernel_size=5)




    def forward(self, x):
```

```
        # put implementation here
        conv_3_output = torch.cat([self.conv_layer_3[i]
(x[:,self.ch_in_3[i],:,:]) for i in range(6)], dim=1)
        conv_4_output = torch.cat([self.conv_layer_4[i]
(x[:,self.ch_in_4[i],:,:]) for i in range(9)], dim=1)
        conv_6_output = self.conv_layer_6(x[:,self.ch_in_6[0],:,:])

        return torch.cat([conv_3_output, conv_4_output,
conv_6_output], dim=1)

class LeNet(nn.Module) :
    def __init__(self) :
        super(LeNet, self).__init__()
        #padding=2 makes 28x28 image into 32x32
        self.C1_layer = nn.Sequential(
                nn.Conv2d(1, 6, kernel_size=5, padding=2),
                nn.Tanh()
                )
        self.P2_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C3_layer = nn.Sequential(
                #C3_layer_full(),
                C3_layer(),
                nn.Tanh()
                )
        self.P4_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C5_layer = nn.Sequential(
                nn.Linear(5*5*16, 120),
                nn.Tanh()
                )
        self.F6_layer = nn.Sequential(
                nn.Linear(120, 84),
                nn.Tanh()
                )
        self.F7_layer = nn.Linear(84, 10)
        self.tanh = nn.Tanh()

    def forward(self, x) :
        output = self.C1_layer(x)
        output = self.P2_layer(output)
        output = self.C3_layer(output)
        output = self.P4_layer(output)
        output = output.view(-1,5*5*16)
        output = self.C5_layer(output)
```

```
        output = self.F6_layer(output)
        output = self.F7_layer(output)
        return output
```

The C_3 layer that is actuall used in the original Lenet architecture is implemented as follows. The indices of the 6 channels that only perform the convolution operation on 3 channels are implemented as a list, and the same for the other 9 and 1 channels. The forward method is implemented as a list indexing using the initialized list and the input, and torch.cat is used to concat the following channels.

```
class LeNet(nn.Module) :
    def __init__(self) :
        super(LeNet, self).__init__()
        #padding=2 makes 28x28 image into 32x32
        self.C1_layer = nn.Sequential(
                nn.Conv2d(1, 6, kernel_size=5, padding=2),
                nn.Tanh()
                )
        self.P2_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C3_layer = nn.Sequential(
                #C3_layer_full(),
                C3_layer(),
                nn.Tanh()
                )
        self.P4_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C5_layer = nn.Sequential(
                nn.Linear(5*5*16, 120),
                nn.Tanh()
                )
        self.F6_layer = nn.Sequential(
                nn.Linear(120, 84),
                nn.Tanh()
                )
        self.F7_layer = nn.Linear(84, 10)
        self.tanh = nn.Tanh()

    def forward(self, x) :
        output = self.C1_layer(x)
        output = self.P2_layer(output)
        output = self.C3_layer(output)
        output = self.P4_layer(output)
        output = output.view(-1,5*5*16)
        output = self.C5_layer(output)
```

```
        output = self.F6_layer(output)
        output = self.F7_layer(output)
        return output

'''
Step 3
'''
model = LeNet().to(device)
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

# print total number of trainable parameters
param_ct = sum([p.numel() for p in model.parameters()])
print(f"Total number of trainable parameters: {param_ct}")

'''
Step 4
'''
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
batch_size=100, shuffle=True)

import time
start = time.time()
for epoch in range(10) :
    print("{}th epoch starting.".format(epoch))
    for images, labels in train_loader :
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        train_loss = loss_function(model(images), labels)
        train_loss.backward()

        optimizer.step()
end = time.time()
print("Time ellapsed in training is: {}".format(end - start))


'''
Step 5
'''
test_loss, correct, total = 0, 0, 0

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
batch_size=100, shuffle=False)

for images, labels in test_loader :
    images, labels = images.to(device), labels.to(device)

    output = model(images)
    test_loss += loss_function(output, labels).item()
```

```
    pred = output.max(1, keepdim=True)[1]
    correct += pred.eq(labels.view_as(pred)).sum().item()

    total += labels.size(0)

print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\
n'.format(
        test_loss /total, correct, total,
        100. * correct / total))
```

```
Total number of trainable parameters: 60806
0th epoch starting.
1th epoch starting.
2th epoch starting.
3th epoch starting.
4th epoch starting.
5th epoch starting.
6th epoch starting.
7th epoch starting.
8th epoch starting.
9th epoch starting.
Time ellapsed in training is: 256.9895806312561
[Test set] Average loss: 0.0004, Accuracy: 9841/10000 (98.41%)
```

```
class LeNet(nn.Module) :
    def __init__(self) :
        super(LeNet, self).__init__()
        #padding=2 makes 28x28 image into 32x32
        self.C1_layer = nn.Sequential(
                nn.Conv2d(1, 6, kernel_size=5, padding=2),
                nn.Tanh()
                )
        self.P2_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C3_layer = nn.Sequential(
                C3_layer_full(),
                # C3_layer(),
                nn.Tanh()
                )
        self.P4_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C5_layer = nn.Sequential(
                nn.Linear(5*5*16, 120),
                nn.Tanh()
```

```python
                )
        self.F6_layer = nn.Sequential(
                nn.Linear(120, 84),
                nn.Tanh()
                )
        self.F7_layer = nn.Linear(84, 10)
        self.tanh = nn.Tanh()

    def forward(self, x) :
        output = self.C1_layer(x)
        output = self.P2_layer(output)
        output = self.C3_layer(output)
        output = self.P4_layer(output)
        output = output.view(-1,5*5*16)
        output = self.C5_layer(output)
        output = self.F6_layer(output)
        output = self.F7_layer(output)
        return output

'''
Step 3
'''
model = LeNet().to(device)
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

# print total number of trainable parameters
param_ct = sum([p.numel() for p in model.parameters()])
print(f"Total number of trainable parameters: {param_ct}")

'''
Step 4
'''
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
batch_size=100, shuffle=True)

import time
start = time.time()
for epoch in range(10) :
    print("{}th epoch starting.".format(epoch))
    for images, labels in train_loader :
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        train_loss = loss_function(model(images), labels)
        train_loss.backward()

        optimizer.step()
end = time.time()
print("Time ellapsed in training is: {}".format(end - start))
```

```
'''
Step 5
'''
test_loss, correct, total = 0, 0, 0

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
batch_size=100, shuffle=False)

for images, labels in test_loader :
    images, labels = images.to(device), labels.to(device)

    output = model(images)
    test_loss += loss_function(output, labels).item()

    pred = output.max(1, keepdim=True)[1]
    correct += pred.eq(labels.view_as(pred)).sum().item()

    total += labels.size(0)

print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\
n'.format(
        test_loss /total, correct, total,
        100. * correct / total))
```

```
Total number of trainable parameters: 61706
0th epoch starting.
1th epoch starting.
2th epoch starting.
3th epoch starting.
4th epoch starting.
5th epoch starting.
6th epoch starting.
7th epoch starting.
8th epoch starting.
9th epoch starting.
Time ellapsed in training is: 70.5274019241333
[Test set] Average loss: 0.0004, Accuracy: 9867/10000 (98.67%)
```

The number parameters decrease from 61706 to 60806 which 900 parameters are decreased. This value is identical of the number of parameter loss in the C_3 channel, which can be obtained by $5 \times 5 \times \left(6 \times 16 - \left(3 \times 6 + 4 \times 9 + 6 \times 1\right)\right)$