```
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='~/Downloads/mnist_data/',
                               train=True,
                               transform=transforms.ToTensor(),)

test_dataset = datasets.MNIST(root='~/Downloads/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

        # put implementation here
        self.ch = self.ch_in_3 + self.ch_in_4 + [list(range(6))]

        self.Conv_layers = nn.ModuleList()
        for _ in range(6):
            self.Conv_layers.append(nn.Conv2d(3,1,kernel_size=5))
        for _ in range(9):
            self.Conv_layers.append(nn.Conv2d(4,1,kernel_size=5))
        self.Conv_layers.append(nn.Conv2d(6,1,kernel_size=5))

    def forward(self, x):
        # put implementation here
        output = []
        for i in range(16):
            output.append(self.Conv_layers[i](x[:,self.ch[i],:,:]))

        return torch.cat(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


'''
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))
```

Result

```
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: 464.1957411766052
[Test set] Average loss: 0.0004, Accuracy: 9841/10000 (98.41%)
```

Hand calculation

C3_layer : 6*(3*5*5) + 9*(4*5*5) + 1*(6*5*5) + 16(bias) = 1516
C3_layer_full : 6*5*5*16 + 16(bias) = 2416

주석 수정하여 C3_layer_full로 바꾸어 장착할 경우
Total number of trainable parameters가 61706으로 900증가하므로
hand calculation한 결과와 일치