ADA7-ThreeLayer.py

```python
# Implementation of a Three-layer neural network for MNIST classification
# with manual gradient calculation and manual optimization.

import torch
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

torch.manual_seed(0)
lr = 0.005
hidden_dim = 15
batch_size = 64
epochs = 20
plot = True

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]) # mnist mean and std
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=1000, shuffle=False)

class ThreeLayerNet:
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        self.W1 = torch.randn(input_size, hidden_size1) * 0.1
        self.b1 = torch.randn(hidden_size1) * 0.1
        self.W2 = torch.randn(hidden_size1, hidden_size2) * 0.1
        self.b2 = torch.randn(hidden_size2) * 0.1
        self.W3 = torch.randn(hidden_size2, output_size) * 0.1
        self.b3 = torch.randn(output_size) * 0.1

    def forward(self, x):
        # Input
        self.x = x
        # Hidden Layer 1
        self.z1 = x @ self.W1 + self.b1
        self.a1 = F.relu(self.z1)
        # Hidden Layer 2
        self.z2 = self.a1 @ self.W2 + self.b2
        self.a2 = F.relu(self.z2)
        # Output Layer
        self.z3 = self.a2 @ self.W3 + self.b3
        return self.z3


model = ThreeLayerNet(784, hidden_dim, hidden_dim, 10)

train_loss_values = []
test_loss_values = []
train_accuracy_values = []
test_accuracy_values = []

def train(epoch):
    for batch_idx, (data, target) in enumerate(train_loader):
```

```
        data = data.view(-1, 784)  # flatten the input

        # === FORWARD PASS ===
        output = model.forward(data)
        log_softmax = F.log_softmax(output, dim=1)
        loss = - torch.mean(log_softmax[range(len(target)), target]) # Equivalent to NLLLoss

        # === BACKWARD PASS ===
        # gradient of the loss w.r.t. output of model
        grad_z3 = F.softmax(output, dim=1)
        grad_z3[range(len(target)), target] -= 1
        grad_z3 /= len(target) # recall that loss is average over batch

        # gradient of the loss w.r.t. the output after the second hidden layer ReLU
        grad_a2 = grad_z3 @ model.W3.T

        # gradient of the loss w.r.t. the output before the second hidden layer ReLU
        grad_z2 = grad_a2.clone()
        grad_z2[model.z2 < 0] = 0

        # gradient of the loss w.r.t. the output after the first hidden layer ReLU
        grad_a1 = grad_z2 @ model.W2.T

        # gradient of the loss w.r.t. the output before the first hidden layer ReLU
        grad_z1 = grad_a1.clone()
        grad_z1[model.z1 < 0] = 0

        # gradient of the loss w.r.t. the model parameters
        model.W3.grad = model.a2.T @ grad_z3
        model.b3.grad = grad_z3.sum(axis=0)
        model.W2.grad = model.a1.T @ grad_z2
        model.b2.grad = grad_z2.sum(axis=0)
        model.W1.grad = model.x.T @ grad_z1
        model.b1.grad = grad_z1.sum(axis=0)

        # === PARAM UPDATES === # Vanilla gradient descent
        model.W3 -= lr * model.W3.grad
        model.b3 -= lr * model.b3.grad
        model.W2 -= lr * model.W2.grad
        model.b2 -= lr * model.b2.grad
        model.W1 -= lr * model.W1.grad
        model.b1 -= lr * model.b1.grad

        # === PRINT EVERY 200 ITERATIONS ===
        if batch_idx % 200 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))

@torch.no_grad()
def check(train_or_test, loader):
    loss, correct = 0, 0
    for data, target in loader:
        data = data.view(-1, 784) # flatten the input
        output = model.forward(data)
        log_softmax = F.log_softmax(output, dim=1)
        loss += F.nll_loss(log_softmax, target, reduction='sum').item() # sum up batch loss
        pred = output.argmax(dim=1, keepdim=True) # get the index of the max log-probability
```

```python
        correct += pred.eq(target.view_as(pred)).sum().item() # pred is batch_size x 1, target is batch_size

    loss /= len(loader.dataset) # note that reduction is 'sum' instead of 'mean' in F.nll_loss
    accuracy = 100. * correct / len(loader.dataset)
    print('{} set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'.format(
        train_or_test, loss, correct, len(loader.dataset), accuracy))
    return loss, accuracy

for epoch in range(1, 1+epochs):
    train(epoch)
    train_loss, train_acc = check(train_or_test='train', loader=train_loader)
    train_loss_values.append(train_loss)
    train_accuracy_values.append(train_acc)
    test_loss, test_acc = check(train_or_test='test', loader=test_loader)
    test_loss_values.append(test_loss)
    test_accuracy_values.append(test_acc)
    print('', end='\n')
    # Notice how we are doing a full forward pass again at the end of each epoch to compute the train in loss and accuracy,
    # but to save time, we could keep track of the loss (or number of correct predictions) for each mini-batch
    # and take an average at the end of the epoch instead.

if plot:
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(range(1, 1+epochs), train_loss_values, label='Training Loss')
    plt.plot(range(1, 1+epochs), test_loss_values, label='Test Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(range(1, 1+epochs), train_accuracy_values, label='Training Accuracy')
    plt.plot(range(1, 1+epochs), test_accuracy_values, label='Test Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.legend()

    plt.show()
```