LeNet

Net

```
import torch
from torch import nn

# define network
class LeNet(nn.Module):
    # initialize the network
    def __init__(self):
        super(LeNet, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5,
stride=1, padding=2)
        # (28-5+2*2)/1 + 1 = 28 人话就是输入加两个填充再减卷积层
        self.sigmoid1 = nn.Sigmoid()
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        # (28-2)/2 + 1 = 14
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5,
stride=1, padding=0)
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5,
stride=1, padding=0)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(in_features=120, out_features=84)
        self.output = nn.Linear(in_features=84, out_features=10)

    # forward pass
    def forward(self, x):
        x = self.conv1(x)
        x = self.sigmoid1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.conv3(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.output(x)
        return x

if __name__ == '__main__':
    x = torch.randn(1, 1, 28, 28)
    model = LeNet()
    y = model(x)
    print(y.shape)
```

Train

```
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from torchvision.transforms import ToPILImage
from net import LeNet
from torch.autograd import Variable
```

```
# Define the transforms for the data
date_transforms = transforms.Compose([
    transforms.ToTensor(),
])

# Load the dataset
train_dataset = datasets.MNIST('../Deeplearing_data', train=True,
transform=date_transforms, download=False)
train_dataloader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_dataset = datasets.MNIST('../Deeplearing_data', train=True,
transform=date_transforms, download=False)
test_dataloader = DataLoader(test_dataset, batch_size=128, shuffle=False)

# If you want to use GPU, uncomment the following line
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define the model
model = LeNet().to(device)

model.load_state_dict(torch.load('D:/LeNet/saved_model/best_model.pth'))

classes = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')

for i in range(10):
    img, label = test_dataset[i][0], train_dataset[i][1]
    img = Variable(torch.unsqueeze(img, 0)).to(device)
    with torch.no_grad():
        output = model(img)
        predicted, actual = classes[torch.argmax(output[0])], classes[label]
        print('Predicted: ', predicted, 'Actual: ', actual)
```

Test

```
import os
import torch
from torch import nn
from torch.optim import lr_scheduler
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from net import LeNet

# Define the transforms for the data
date_transforms = transforms.Compose([
    transforms.ToTensor(),
])

# Load the dataset
train_dataset = datasets.MNIST('../Deeplearing_data', train=True,
transform=date_transforms, download=False)
train_dataloader = DataLoader(train_dataset, batch_size=16, shuffle=True)
test_dataset = datasets.MNIST('../Deeplearing_data', train=False,
transform=date_transforms, download=False)
test_dataloader = DataLoader(test_dataset, batch_size=16, shuffle=False)

# If you want to use GPU, uncomment the following line
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
# Define the model
model = LeNet().to(device)

# Define the loss function and optimizer
loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3, momentum=0.9)

# Define the learning rate scheduler
scheduler = lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

# Train the model
def train(dataloader, model, loss_func, optimizer, device):
    model.train()
    total_loss, total_correct, total_samples = 0.0, 0, 0
    for batch_idx, (data, target) in enumerate(dataloader):
        data, target = data.to(device), target.to(device)
        output = model(data)
        loss = loss_func(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * data.size(0)
        _, pred = torch.max(output, 1)
        total_correct += (pred == target).sum().item()
        total_samples += data.size(0)
    average_loss = total_loss / total_samples
    accuracy = total_correct / total_samples
    print(f"train_loss: {average_loss:.4f}")
    print(f"train_acc: {accuracy:.4f}")

# Test the model
def test(dataloader, model, loss_func, device):
    model.eval()
    total_loss, total_correct, total_samples = 0.0, 0, 0
    with torch.no_grad():
        for batch_idx, (data, target) in enumerate(dataloader):
            data, target = data.to(device), target.to(device)
            output = model(data)
            loss = loss_func(output, target)
            total_loss += loss.item() * data.size(0)
            _, pred = torch.max(output, 1)
            total_correct += (pred == target).sum().item()
            total_samples += data.size(0)
    average_loss = total_loss / total_samples
    accuracy = total_correct / total_samples
    print(f"test_loss: {average_loss:.4f}")
    print(f"test_acc: {accuracy:.4f}")
    return accuracy

# Start training
epochs = 50
min_acc = 0
for epoch in range(epochs):
    print(f'epoch {epoch+1}\n----------------')
    train(train_dataloader, model, loss_func, optimizer, device)
    accuracy = test(test_dataloader, model, loss_func, device)
    scheduler.step()  # Adjust learning rate
    if accuracy > min_acc:
```

```
        folder = 'saved_model'
        if not os.path.exists(folder):
            os.mkdir(folder)
        min_acc = accuracy
        print('Saving best model')
        torch.save(model.state_dict(), 'saved_model/best_model.pth')
print('Training complete')
```