Handwritten Digit Recognition with LeNet

Experiment Description

使用`MNIST`手写数字体数据集进行训练和预测，实现测试集准确率达到98%及以上。本实验的目的：

- 掌握卷积神经网络基本原理，以`LeNet`为例
- 掌握主流框架的基本用法以及构建卷积神经网络的基本操作
- 了解如何使用`GPU`

Experiment Procedure

Preparation

Import libraries: pytorch, matplotlib, numpy, torchinfo. In MacOS, GPU acceleration is achieved through MPS of pytorch. Check if MPS is available.

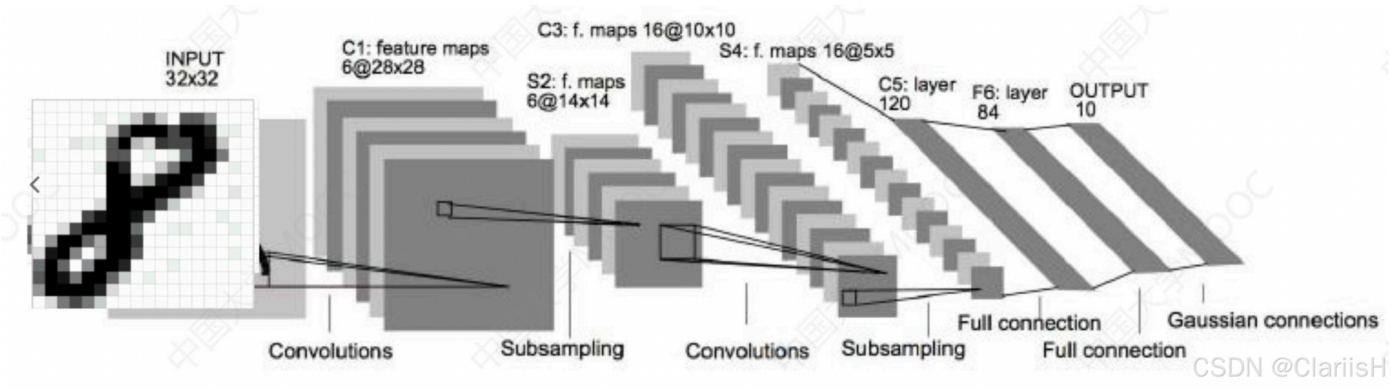
```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import torchvision
import numpy as np
import torch.nn.functional as F
from torchinfo import summary
import os

# 检查 MPS
device = torch.device("mps")
print(torch.backends.mps.is_available())
```

Download Dataset

- Set the download path as data file under the project: use os.path.join to create new *data* folder as data_path to store train data and test data.
- Load MNIST dataset:
  - Use torchvision.transforms.ToTensor() as transformation function. Use datasets to download MNIST dataset to data_path, and transform it into a tensor(tensor is like ndarray, without dimention limit, suitable for deep learning matrix operation)
  - Set `batch_size` = 32
- Dataloader: use data loader `train_dl` and `test_dl` to load dataset into model in `batch_size`
- Check the data of first batch: Use `next(iter(train_dl))` to check the first batch of data in `train_dl`, and return `Batch shape: torch.Size([32, 1, 28, 28])`, which means load 32 data per batch, one channel, the height is 28px and the width is 32px.
- Show sample picture: show the first 20 sample pictures.

```
# 下载路径
data_path = './data'

# load data
transform = torchvision.transforms.ToTensor()
train_ds = torchvision.datasets.MNIST(data_path, train=True, transform=transform,
download=True)
test_ds = torchvision.datasets.MNIST(data_path, train=False, transform=transform,
download=True)

batch_size = 32

# dataloader
train_dl = torch.utils.data.DataLoader(train_ds, batch_size=batch_size, shuffle=True)
test_dl = torch.utils.data.DataLoader(test_ds, batch_size=batch_size)

# see first batch data
imgs, labels = next(iter(train_dl))
print("Batch shape:", imgs.shape)

# see pictures
plt.figure(figsize=(20, 5))
for i, img in enumerate(imgs[:20]):
    npimg = np.squeeze(img.numpy())
    plt.subplot(2, 10, i + 1)
    plt.imshow(npimg, cmap=plt.cm.binary)
    plt.axis('off')
plt.show()
```

Build the Model Network



The structure of the network.

Define `num_classes` as 10. The network model inherit nn.Module of pytorch.

Initialize Network Structure

- First convolusion layer: input 1 channel, and output 6 channels
- Second convolusion layer: input 6 channels, and output 16 channels
- Full connection layer

    - fc1 input 16 5*5 features, and output 120 units.
    - fc2 input 120 units, and output 84 units
    - fc2 input 84 units, and output 10 categories(MNIST)

Forward: from input data x

- First layer of convolution + pooling + activation: After conv1 and relu, use F.max_pool2d(x, 2) for 2*2 max pooling
- Second layer of convolution + pooling + activation: After con2 and relu, use F.max_pool2d(x, 2) for 2*2 max pooling
- Flatten the data: torch.flatten(x, 1) flatten the data into one-dimension
- Full connection layer:
    - Pass the data into fc1, and the output function is relu.
    - Pass the data into fc2, and the output function is relu.
    - Pass the data into fc3, and get the final output.

```
# model definiton
num_classes = 10
import torch.nn.functional as F

class LeNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)

        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        # flatten
        x = torch.flatten(x, 1)

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)  # output
        return x
```

Load and Print

Use .to(device) to transfer model to GPU. Use summary(model) to print the parameters of the model.

```
model = LeNet().to(device)
summary(model)
```

Model Training

1. Set hyperparameters

   Use cross entropy as the loss function of this classification model, set the learning rate to 1e-2 (0.01), and use stochastic gradient descent algorithm as optimization to update the parameters.

```
loss_fn = nn.CrossEntropyLoss()
learn_rate = 1e-2
opt = torch.optim.SGD(model.parameters(), lr=learn_rate)
```

2. Training function: train()

Firstly, calculate size and num_batches to use for later calculation of accuracy and loss of train dataset. Then, load the data in batches to GPU. Use trained model to forward the input data X, and get the prediction value pred. After that, use loss_fn to calculate the error between pred(predicted label) the y(real label), clear gradient, and calculate the new gradient of the loss to all parameters of the model. Finally, update the parameters of the model.

Change the boolean value represents whether the predicted label equals real label into float, and calculate training accuracy: Add the loss of all batches, calculate the sum loss, and output accuracy and loss.

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    train_loss, train_acc = 0, 0
    for X, y in dataloader:
        X, y = X.to(device), y.to(device) # move data to GPU
        pred = model(X)
        loss = loss_fn(pred, y) # calculate loss between predicted and real labels
        optimizer.zero_grad() # empty gradient
        loss.backward() # calculate gradient of all parameters
        optimizer.step() # update parameters
        train_acc += (pred.argmax(1) == y).type(torch.float).sum().item()
        train_loss += loss.item()
    return train_acc / size, train_loss / num_batches
```

Test Function

With torch.no_grad(): Forbid gradient calculation, which can save memory and compute resourse, because testing phase does not need backpropagation and parameters update. Calculate size and num_batches to facilitate the future calculation of accuracy and loss. Then load imgs and target data into GPU. Firstly use the trained model to forward the input `imgs`, and get the prediction `target_pred`. Use loss_fn to calculate the loss between target_pred and real label target. Use test_loss to accumulate loss. Finally, change the boolean value represents whether the predicted label equals real label into float, and calculate teat accuracy.

```python
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, test_acc = 0, 0
    with torch.no_grad():
        for imgs, target in dataloader:
            imgs, target = imgs.to(device), target.to(device)
            target_pred = model(imgs)
            loss = loss_fn(target_pred, target)
            test_loss += loss.item()
            test_acc += (target_pred.argmax(1) == target).type(torch.float).sum().item()
    return test_acc / size, test_loss / num_batches
```

Training

Use two methods for training, with and without Batch Normalization and Dropout.

1. Set the epoch is 8, and train in batches, formatted output Epoch，Train_acc，Train_loss，Test_acc and Test_loss every batch.

```python
# train
epochs = 8
train_loss = []
train_acc = []
test_loss = []
test_acc = []

for epoch in range(epochs):
    # model.train() # Use Batch Normalization, Dropout
    # epoch_train_acc, epoch_train_loss = train(train_dl, model, loss_fn, opt)
    # model.eval()
    epoch_test_acc, epoch_test_loss = test(test_dl, model, loss_fn)

    train_acc.append(epoch_train_acc)
    train_loss.append(epoch_train_loss)
    test_acc.append(epoch_test_acc)
    test_loss.append(epoch_test_loss)

    template = 'Epoch:{:2d}, Train_acc:{:.1f}%, Train_loss:{:.3f}, Test_acc:{:.1f}%, 
Test_loss:{:.3f}'
    print(template.format(epoch + 1, epoch_train_acc * 100, epoch_train_loss, 
epoch_test_acc * 100, epoch_test_loss))

print('Done')
```

Result Presentation

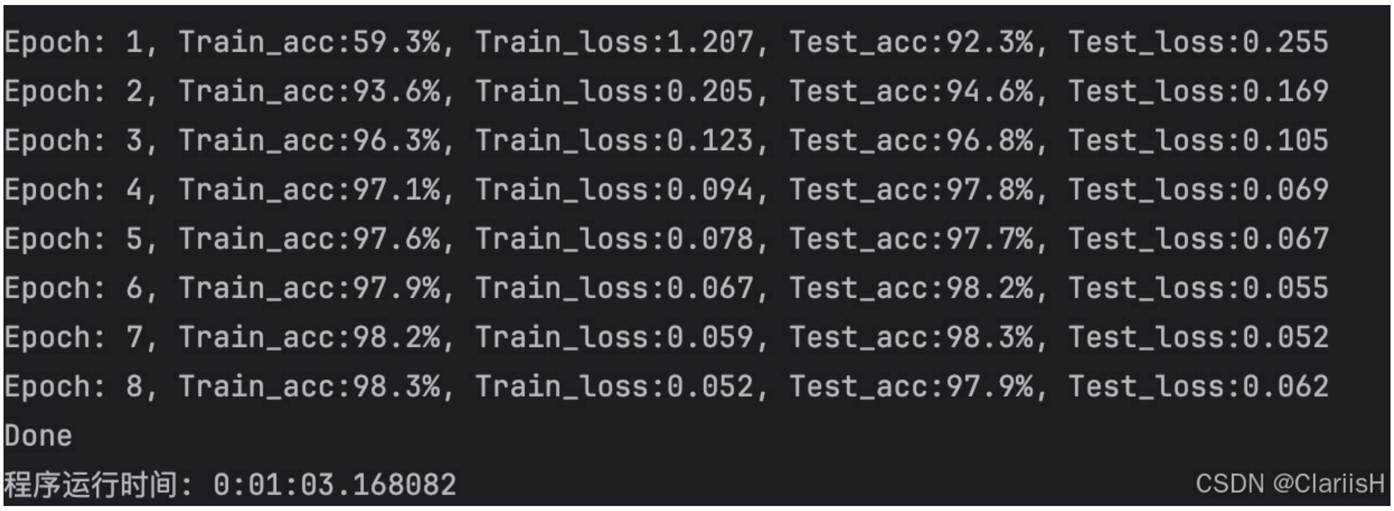
Plot Training/Test Accuracy plot and Training/Test Loss plot in the same line.

```python
plt.rcParams['axes.unicode_minus'] = False
plt.rcParams['figure.dpi'] = 100

epochs_range = range(epochs)
plt.figure(figsize=(12, 3))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, train_acc, label='Training Accuracy')
plt.plot(epochs_range, test_acc, label='Test Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, train_loss, label='Training Loss')
plt.plot(epochs_range, test_loss, label='Test Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

Analysis

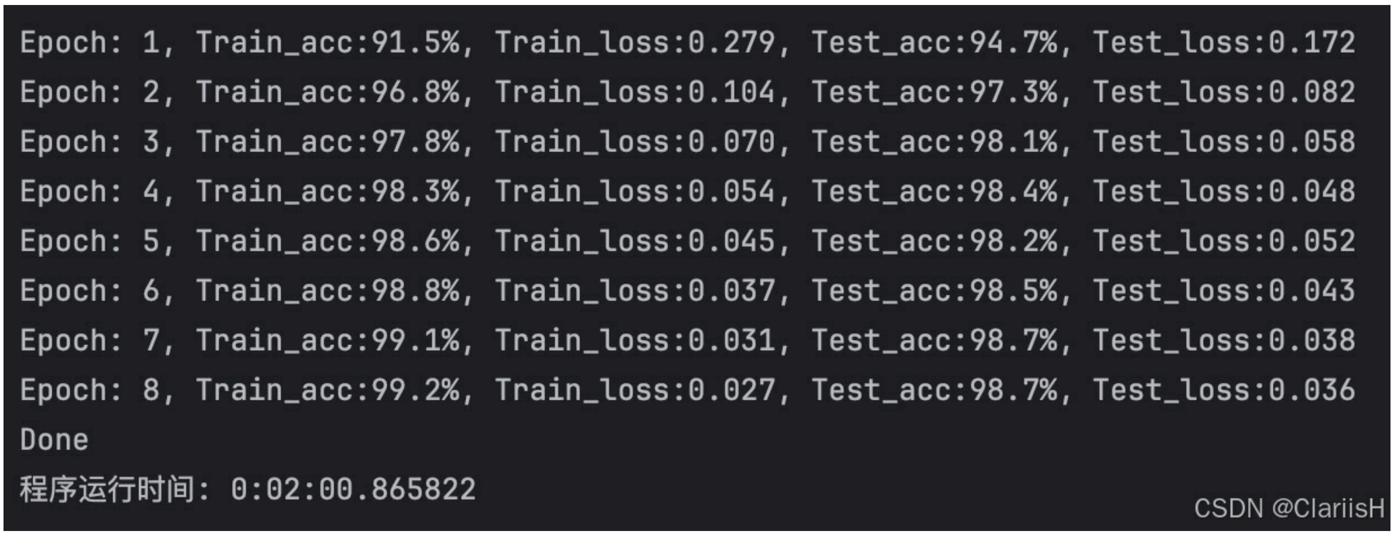
Print result

Without Batch Normalization and Dropout



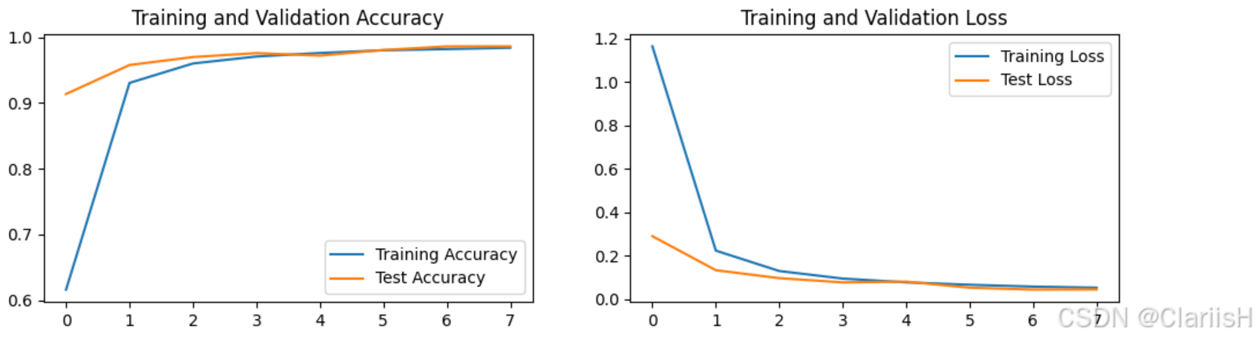
```
Epoch: 1, Train_acc:59.3%, Train_loss:1.207, Test_acc:92.3%, Test_loss:0.255
Epoch: 2, Train_acc:93.6%, Train_loss:0.205, Test_acc:94.6%, Test_loss:0.169
Epoch: 3, Train_acc:96.3%, Train_loss:0.123, Test_acc:96.8%, Test_loss:0.105
Epoch: 4, Train_acc:97.1%, Train_loss:0.094, Test_acc:97.8%, Test_loss:0.069
Epoch: 5, Train_acc:97.6%, Train_loss:0.078, Test_acc:97.7%, Test_loss:0.067
Epoch: 6, Train_acc:97.9%, Train_loss:0.067, Test_acc:98.2%, Test_loss:0.055
Epoch: 7, Train_acc:98.2%, Train_loss:0.059, Test_acc:98.3%, Test_loss:0.052
Epoch: 8, Train_acc:98.3%, Train_loss:0.052, Test_acc:97.9%, Test_loss:0.062
Done
程序运行时间: 0:01:03.168082
```

Using Batch Normalization and Dropout



```
Epoch: 1, Train_acc:91.5%, Train_loss:0.279, Test_acc:94.7%, Test_loss:0.172
Epoch: 2, Train_acc:96.8%, Train_loss:0.104, Test_acc:97.3%, Test_loss:0.082
Epoch: 3, Train_acc:97.8%, Train_loss:0.070, Test_acc:98.1%, Test_loss:0.058
Epoch: 4, Train_acc:98.3%, Train_loss:0.054, Test_acc:98.4%, Test_loss:0.048
Epoch: 5, Train_acc:98.6%, Train_loss:0.045, Test_acc:98.2%, Test_loss:0.052
Epoch: 6, Train_acc:98.8%, Train_loss:0.037, Test_acc:98.5%, Test_loss:0.043
Epoch: 7, Train_acc:99.1%, Train_loss:0.031, Test_acc:98.7%, Test_loss:0.038
Epoch: 8, Train_acc:99.2%, Train_loss:0.027, Test_acc:98.7%, Test_loss:0.036
Done
程序运行时间: 0:02:00.865822
```

Using , the Training/Test Accuracy and Training/Test Loss plots.



It can be seen from the result that, the train result is good, the the accuracy is all above 95%. In this case, the accuracy of training increases rapidly, and the models have good performance on both train dataset and test dataset. If Batch Normalization and Dropout are implemented, the model reaches 98% of accuracy in the third epoch; if not implemented, it reaches 98% of accuracy in the sixth epoch. For the final result, the accuracy of using Batch Normalization and Dropout is also higher, but with a longer training time. Thus, it can be seen that Batch Normalization and Dropout can increase the accuracy of handwritting recognization by LeNet, with a higher final accuracy, but it also consumes more training time.