加载数据

```python
import torch
from torch import nn, optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

# 定义图像转换操作：转换为Tensor，并进行标准化
transform = transforms.Compose([
    transforms.ToTensor(),
    # 转换为[0,1] 的张量
    transforms.Normalize(mean=(0.5,), std=(0.5,))  # 标准化为[-1,1]
])

# 加载训练集和测试集
train_dataset = datasets.FashionMNIST(
    root='data',
    train=True,
    transform=transform,
    download=True
)

test_dataset = datasets.FashionMNIST(
    root='data',
    train=False,
    transform=transform,
    download=True
)

# 创建数据加载器
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# 获取训练集中第一个批次
images, labels = next(iter(train_loader))

# 输出第一个图像张量的尺寸
print(images[0].shape)
```

```
torch.Size([1, 28, 28])
```

训练模型

```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
```

```python
# 设备设置
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 定义残差块（Residual Block）
class ResidualBlock(nn.Module):
    def __init__(self, input_features, output_features,
dropout_prob=0.4):
        super().__init__()
        self.linear_layer = nn.Linear(input_features, output_features)
        self.batch_norm = nn.BatchNorm1d(output_features)
        self.relu_activation = nn.ReLU()
        self.dropout_layer = nn.Dropout(dropout_prob)

        # 快捷连接（shortcut）
        self.shortcut = nn.Identity()
        if input_features != output_features:
            self.shortcut = nn.Sequential(
                nn.Linear(input_features, output_features),
                nn.BatchNorm1d(output_features)
            )

        self._initialize_weights(self.linear_layer)
        if isinstance(self.shortcut, nn.Sequential):
            self._initialize_weights(self.shortcut[0])

    def _initialize_weights(self, layer):
        if isinstance(layer, nn.Linear):
            nn.init.kaiming_normal_(layer.weight, nonlinearity='relu')
            if layer.bias is not None:
                nn.init.constant_(layer.bias, 0)

    def forward(self, x):
        residual = self.shortcut(x)
        output = self.linear_layer(x)
        output = self.batch_norm(output)
        output = self.relu_activation(output)
        output = self.dropout_layer(output)
        return output + residual

# 定义模型结构
model = nn.Sequential(
    nn.Flatten(),
    ResidualBlock(28*28, 256),
    ResidualBlock(256, 128, dropout_prob=0.2),
    nn.Linear(128, 10)
).to(device)

# 超参数设置
learning_rate = 0.001
batch_size = 64
```

```
epochs = 5
l2_regularization_coeff = 1e-4  # L2 正则化系数

# 损失函数和优化器
loss_function = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=l2_regularization_coeff)
# L2 正则化由weight_decay 控制

training_losses = []

for epoch in range(epochs):
    model.train()
    total_loss = 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        predictions = model(images)
        loss = loss_function(predictions, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    epoch_loss = total_loss / len(train_loader)
    training_losses.append(epoch_loss)
    print(f"周期: {epoch+1}/{epochs} - 损失: {epoch_loss:.2f}")

torch.save(model.state_dict(), "trained_model.pth")
```

```
周期: 1/5 - 损失: 0.48
周期: 2/5 - 损失: 0.39
周期: 3/5 - 损失: 0.37
周期: 4/5 - 损失: 0.35
周期: 5/5 - 损失: 0.34
```

计算准确率

```
import numpy as np
from sklearn.metrics import classification_report, roc_auc_score

model.eval()  # 设置为评估模式
pred_list = []
label_list = []
prob_list = []

with torch.no_grad():
    for x_batch, y_batch in test_loader:
        x_batch = x_batch.to(device)
        y_batch = y_batch.to(device)
        logits = model(x_batch)
```

```
        probs = torch.softmax(logits, dim=1)       # 概率分布
        preds = torch.argmax(logits, dim=1)        # 分类结果

        prob_list.append(probs.cpu().numpy())      # 概率用于AUC
        pred_list.append(preds.cpu().numpy())      # 预测类别
        label_list.append(y_batch.cpu().numpy())   # 实际标签

# 合并所有批次结果
prob_list = np.concatenate(prob_list)
pred_list = np.concatenate(pred_list)
label_list = np.concatenate(label_list)

accuracy = (pred_list == label_list).mean()
print(f"准确率: {accuracy:.4f}")

print("\n分类报告:")
print(classification_report(label_list, pred_list))

# 多分类AUC 计算（macro-ovr 策略）
try:
    auc_score = roc_auc_score(label_list, prob_list,
multi_class='ovr', average='macro')
    print(f"AUC (macro-ovr): {auc_score:.4f}")
except Exception as err:
    print(f"Error Message : {err}")
```

```
准确率: 0.8572

分类报告:
              precision    recall  f1-score   support

           0       0.80      0.83      0.81      1000
           1       0.94      0.98      0.96      1000
           2       0.68      0.85      0.76      1000
           3       0.92      0.81      0.86      1000
           4       0.82      0.66      0.73      1000
           5       0.94      0.96      0.95      1000
           6       0.68      0.66      0.67      1000
           7       0.93      0.92      0.93      1000
           8       0.95      0.97      0.96      1000
           9       0.95      0.94      0.95      1000

    accuracy                           0.86     10000
   macro avg       0.86      0.86      0.86     10000
weighted avg       0.86      0.86      0.86     10000

AUC (macro-ovr): 0.9886
```

卷积神经网络

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import classification_report, roc_auc_score

# 自定义卷积神经网络（CNN）
class CustomCNN(nn.Module):
    def __init__(self):
        super(CustomCNN, self).__init__()

        # 第一层卷积：输入 1 通道，输出 32 通道，kernel=3, stride=1, padding=1
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32,
kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)  # 输出尺寸减
半

        # 第二层卷积：32 -> 64 通道
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64,
kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # 展平
        self.flatten = nn.Flatten()

        # 全连接层，输入是池化后的展平尺寸：64 通道x 7 x 7 = 3136
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.relu3 = nn.ReLU()

        # 输出层：10 类，使用Softmax
        self.fc2 = nn.Linear(128, 10)
        self.softmax = nn.Softmax(dim=1)  # dim=1 表示按行进行softmax

    def forward(self, x):
        x = self.pool1(self.relu1(self.conv1(x)))
        x = self.pool2(self.relu2(self.conv2(x)))
        x = self.flatten(x)
        x = self.relu3(self.fc1(x))
        x = self.fc2(x)
        x = self.softmax(x)
        return x


# L2 正则化函数
def l2_regularization(model, lambda_):
```

```
    l2_norm = sum(torch.sum(param ** 2) for param in
model.parameters() if param.requires_grad)
    return lambda_ * l2_norm


# 训练函数
def train_model(
    model,
    train_loader,
    test_loader,
    epochs=5,
    lr=0.01,
    lambda_l2=0.001,
    device='cpu'
):
    model.to(device)
    criterion = nn.CrossEntropyLoss()
    training_losses, test_accuracies = [], []

    for epoch in range(epochs):
        model.train()
        total_loss = 0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            outputs = model(images)
            loss = criterion(outputs, labels)
            loss += l2_regularization(model, lambda_l2)

            model.zero_grad()
            loss.backward()

            # 手动 SGD 更新
            with torch.no_grad():
                for param in model.parameters():
                    if param.grad is not None:
                        param -= lr * param.grad

            total_loss += loss.item()

        training_losses.append(total_loss / len(train_loader))
        accuracy = evaluate_accuracy(model, test_loader, device)
        test_accuracies.append(accuracy)
        print(f"Epoch {epoch+1}: Loss={training_losses[-1]:.4f},
Accuracy={accuracy:.2f}%")

    return training_losses, test_accuracies


# 评估准确率
```

```python
def evaluate_accuracy(model, data_loader, device='cpu'):
    model.eval()
    correct = total = 0
    with torch.no_grad():
        for X, y in data_loader:
            X, y = X.to(device), y.to(device)
            preds = model(X).argmax(dim=1)
            correct += (preds == y).sum().item()
            total += y.size(0)
    return 100 * correct / total


# 绘制训练过程的损失和准确率
def plot_training_progress(losses, accuracies):
    plt.figure(figsize=(10, 4))

    plt.subplot(1, 2, 1)
    plt.plot(losses, label="Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training Loss")
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(accuracies, label="Accuracy", color="orange")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy (%)")
    plt.title("Test Accuracy")
    plt.legend()

    plt.tight_layout()
    plt.show()


# 详细模型评估：分类报告、AUC 等
def evaluate_detailed_model(model, data_loader, device='cpu'):
    model.eval()
    all_preds, all_probs, all_labels = [], [], []

    with torch.no_grad():
        for X, y in data_loader:
            X = X.to(device)
            logits = model(X)
            probs = torch.softmax(logits, dim=1).cpu().numpy()
            preds = np.argmax(probs, axis=1)
            all_preds.extend(preds)
            all_probs.extend(probs)
            all_labels.extend(y.numpy())

    print("分类报告:")
```

```python
    print(classification_report(all_labels, all_preds, digits=4))

    try:
        auc = roc_auc_score(all_labels, all_probs, multi_class='ovr')
        print(f"AUC Score (OvR): {auc:.4f}")
    except ValueError:
        print("AUC 未计算，可能是类不平衡或其他问题。")


# 训练CNN 模型
cnn_model = CustomCNN()
training_losses, test_accuracies = train_model(cnn_model,
train_loader, test_loader, epochs=5, lr=0.01, lambda_l2=0.001,
device=device)
plot_training_progress(training_losses, test_accuracies)
evaluate_detailed_model(cnn_model, test_loader, device)
```