西南交通大學

《机器学习实验》

课程设计报告

报告题目：基于卷积神经网络的手写数字识别

年　　级：_____

姓　　名：_____

学　　号：_____

二〇二四年六月

基于卷积神经网络的手写数字识别

一、 模型简介

1.1 研究现状

手写数字识别是计算机视觉和机器学习领域的一个重要研究方向，它涉及到从手写文本中自动识别和分类数字，目前已经有了很多成熟的方法。

目前使用最为广泛的数据集是 MNIST 数据集。MNIST 数据集是手写数字识别研究中最常用的数据集之一，包含 60,000 张训练图像和 10,000 张测试图像，每张图像都是 28x28 像素的手写数字。深度学习技术在手写数字识别领域取得了显著进展。卷积神经网络（CNN）因其强大的特征提取能力而成为主流方法。例如，LeNet-5 是早期的 CNN 结构，已被用于手写数字识别。手写数字识别算法已经从最初的传统机器学习算法（如 SVM、KNN）发展到基于深度学习的复杂网络结构，如卷积神经网络（CNN）、循环神经网络（RNN）等。手写数字识别技术在银行汇款单号识别、邮政编码自动识别、车牌号识别等多个领域具有实际应用价值。

现有的识别方法主要分为传统机器学习方法和深度学习方法。传统机器学习方法包括支持向量机（SVM）、K 最近邻（KNN）、决策树等，这些方法依赖于手工特征提取。深度学习方法主要包括卷积神经网络（CNN）、循环神经网络（RNN）、深度置信网络（DBN）等。近年来，注意力机制也被引入到深度学习模型中，以提高模型对关键特征的聚焦能力。

1.2 卷积神经网络（CNN）模型

卷积神经网络进行数字识别是一种深度学习方法。通过卷积层、池化层、全连接层等对于数据进行处理，并最终建立模型，获得预测结果。

卷积层：卷积层是 CNN 中用于提取图像特征的关键组件。它通过卷积核在输入图像上滑动，计算滤波器与图像的局部区域的点积，生成特征图

激活层：$ReLU$是一种非线性激活函数，用于引入非线性，从而使网络能够学习更复杂的模式。它将所有负值置为 0，保留正值，公式为$f(x) = \max(0, x)$。

池化层：池化层用于降低特征图的空间尺寸，从而减少参数数量和计算量，同时使特征检测更加鲁棒。

全连接层：全连接层是 CNN 中的密集层，它将卷积层和池化层提取的二维特征图转换为一维特征向量，然后进行分类。

卷积神经网络（CNN）在手写数字识别任务上具有一系列优点，但同时也存在一些局限性。

优点：

1.CNN 通过其卷积层自动学习图像的局部特征，无需手动设计特征提取器。

2.CNN 可以从原始像素直接学习到分类结果，无需额外的预处理或特征工程。

3.CNN 的识别准确率较高。

缺点：

1.CNN 模型通常需要大量的计算资源，包括高性能的 GPU 来进行训练。

2.CNN 可能会过拟合，尤其是在模型非常深或复杂时。

3.为了训练一个鲁棒的 CNN 模型，通常需要大量的标注数据。在某些情况下，获取足够的训练数据可能是具有挑战性的。
4.CNN 通常在大规模数据集上表现更好，而在只有少量样本的类别上可能表现不佳。

1.3 支持向量机（SVM）

先对于数据集进行预处理，然后再选用不同的核函数（'linear'、'rbf'、'poly'）训练模型，然后再在测试集上进行测试得出准确率。
支持向量机（SVM）是一种广泛使用的监督学习算法，用于分类和回归任务。以下是 SVM 的一些主要优点和缺点：
优点：
1. 在许多应用中，SVM 已经显示出比其他算法更高的准确率。
2. SVM 能够通过核函数将数据映射到高维空间，有效地处理非线性问题。
3. SVM 对数据的噪声和异常值具有一定的鲁棒性。
缺点：
1. SVM 的性能在很大程度上依赖于正确的核函数和参数选择。
2. 对于大规模数据集，SVM 的训练过程可能非常耗时。
3. SVM 在小样本问题上可能表现不佳，因为它依赖于数据分布的统计特性。并且对于不平衡数据集较为敏感。
4.

1.4 logistic 回归

对于数据进行预处理之后训练模型，选用的为默认参数 penalty='l2'L（正则化）和 solver='lbfgs' （L-BFGS 算法）。
逻辑回归（Logistic Regression）是一种广泛使用的统计方法，用于二分类问题，也可以扩展到多类分类问题。以下是逻辑回归的一些主要优点和缺点：
优点：
1. 逻辑回归通常比其他机器学习算法（如决策树、随机森林、SVM 等）训练更快。
2. 对于线性可分的数据，逻辑回归能够提供出色的分类性能。
3. 逻辑回归模型相对简单，容易理解和实现。、
缺点：
1. 逻辑回归假设数据是线性可分的，对于非线性问题，它可能无法捕捉复杂的数据模式。
2. 逻辑回归对异常值比较敏感，异常值可能会对模型的预测结果产生较大影响。
3. 逻辑回归对不平衡的数据集比较敏感，可能需要进行重采样或调整类别权重。

1.5 决策树

对于数据进行预处理之后训练模型，采用的为默认参数 criterion='gini'。
决策树的主要优缺点：
优点：
1.能够处理非线性数据和复杂的决策边界。

2.与线性模型不同，决策树不需要假设数据的分布。
3. 决策树模型的结构清晰，容易可视化和解释。
缺点：
1. 在没有限制的情况下，决策树可能会生成过于复杂的树，导致过拟合。
2. 对样本不均衡敏感，决策树可能偏向于多数类。
3. 单个决策树的预测能力有限，通常需要集成方法（如随机森林）来提高性能。

研究现状及相关工作

二、模型及其参数详细解析

2.1 卷积神经网络（CNN）

2.1.1 下载数据

```
data_train = datasets.MNIST(
    root='./data', train=True, download=True,
transform=transforms.Compose([transforms.ToTensor()])
)
data_test = datasets.MNIST(
    root='./data', train=False, download=True,
transform=transforms.Compose([transforms.ToTensor()])
)
```

该代码是对数据进行下载。因邮箱发送大小限制不能将数据一同上传，运行代码即可下载数据。

2.1.2 卷积层(Conv2d)

```
from torch.nn import (
    Module, Sequential, Conv2d, ReLU, Linear, MaxPool2d, Dropout
)
class Model(Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = Sequential(
            Conv2d(1, 64, kernel_size=3, padding=1),
            ReLU(),
            Conv2d(64, 128, kernel_size=3, padding=1),
            ReLU(),
            MaxPool2d(stride=2, kernel_size=2)
        )

        self.dense = Sequential(
            Linear(14 * 14 * 128, 1024),
            ReLU(),
            Dropout(p=0.5),
```

```
            Linear(1024, 10)
        )

    def forward(self, x):
        x = self.conv1(x)
        x = x.view(x.size(0), -1)
        x = self.dense(x)
        return xConv2d
```

Conv2d 的第一个参数是输入通道数，这里是 1，因为 MNIST 数据集的图像是灰度图像。第二个参数是输出通道数，即滤波器的数量。kernel_size=3 表示每个卷积核的大小是 3x3 像素。padding=1 表示在输入图像边缘添加一圈 0，以保持特征图的空间维度不变。
模型中有两个卷积层，第一个卷积层将输入特征图的通道数从 1 增加到 64，第二个卷积层进一步将 64 通道的特征图增加到 128 通道。这种堆叠有助于学习更复杂的特征表示。

2.1.3 激活层(ReLU)

直接调用 torch.nn.ReLU()
$ReLU$是一种非线性激活函数，用于引入非线性，从而使网络能够学习更复杂的模式。它将所有负值置为 0，保留正值，公式为$f(x) = \max(0, x)$
在每个卷积层之后，$ReLU$被用来激活特征图，增加模型的表达能力。

2.1.4 池化层 (MaxPool2d)

池化层（特别是最大池化）用于降低特征图的空间尺寸，从而减少参数数量和计算量，同时使特征检测更加鲁棒。
实现：MaxPool2d 中的 stride=2 和 kernel_size=2 表示池化窗口的大小是 2x2，步长也是 2，这意味着窗口每次移动两个像素。最大池化会选择每个窗口中的最大值作为输出。

2.1.5 全连接层

全连接层是 CNN 中的密集层，它将卷积层和池化层提取的二维特征图转换为一维特征向量，然后进行分类。
实现：在您的模型中，第一个 Linear 层将 14x14 的特征图（由于池化层的作用，原始 28x28 的特征图尺寸减半）的每个像素点连接到 1024 个节点。第二个 Linear 层将 1024 个节点的输出映射到 10 个输出类别（对应数字 0 到 9）。

2.1.6 模型前向传播

在 forward 方法中，输入图像首先通过卷积层部分，然后通过 view 方法展平为一维向量，最后通过全连接层部分进行分类。

2.1.7 超参数配置

优化器 (Optimizers)
选择了以下优化器来进行模型的参数更新:
- `torch.optim.Adam`: 一种自适应学习率优化算法，广泛用于各种任务。

- `torch.optim.SGD`: 标准随机梯度下降优化器，带动量项。
- `torch.optim.RMSprop`: 另一种自适应学习率方法，主要在处理非平稳目标时表现良好。
- `torch.optim.Adamax`: Adam 的一个变种，使用无限小的梯度进行自适应学习率调整。

尝试不同的步长：
- 0.0001
- 0.001
- 0.005
- 0.01

2.2 SVM 支持向量机

```
from sklearn.svm import SVC

svm_model = SVC(kernel='poly')      # kernel='linear' kernel='rbf'
```

2.3 Logistic 回归

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
```

2.3 决策树

```
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier()
```

三、 项目结构

- `/data/`
  - 存放所有原始数据集文件，包括训练集和测试集。
- `/CNN-number-recognition.py`
  - 卷积神经网络实现的 MNIST 数字识别模型。
- `/data_conversion.py`
  - 数据预处理脚本，用于将`torchvision`的`datasets`下载的数据转换为适用于`sklearn`的数据格式。
- `/decision_tree.py`
  - 决策树模型实现，使用`sklearn.tree.DecisionTreeClassifier`，针对 MNIST 数字识别任务。
- `/logistic.py`
  - 逻辑回归模型实现，使用`sklearn.linear_model.LogisticRegression`，用于 MNIST 数字识别。
- `/SVM.py`
  - 支持向量机模型实现，使用`sklearn.svm.SVC`，用于 MNIST 数字识别。

四、 评价指标

在模型的性能评估中，我选用了以下模型衡量模型效果：

4.1 Loss（损失函数）

损失函数用于衡量模型预测值与实际值之间的差异。常用的损失函数是交叉熵损失（Cross-Entropy Loss）：

$$cost = -\sum_{c=1}^{C} y_{o,c} \log(p_{o,c})$$

其中，C是类别的数量，$y_{o,c}$是一个二进制指示器（如果类别c是观测样本的分类，则为 1，否则为 0），$p_{o,c}$是模型预测观测样本属于类别c的概率。

4.2 Test Accuracy（测试准确率）

测试准确率是衡量模型在测试集上正确分类的比例：

$$Test\ Accuracy = \frac{\text{正确预测的数量}}{\text{总预测数量}}$$

4.3 Precision（精确度）

精确度是针对每个类别的，表示为模型预测为正类别中实际为正类别的比例：

$$Precision = \frac{\text{真正例(TP)}}{\text{真正例(TP)} + \text{假正例(FP)}}$$

4.4 Recall（召回率）

召回率也称为真正例率或灵敏度，表示为实际正类别中被正确预测的比例：

$$Recall = \frac{\text{真正例(TP)}}{\text{真正例(TP)} + \text{假负例(FN)}}$$

4.5 F1 Score（F1 分数）

F1 分数是精确度和召回率的调和平均数，用于衡量模型的准确性和完整性的平衡：

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

F1 分数在精确度和召回率之间取得平衡，是一个综合考虑两者的指标，特别适

用于类别不平衡的情况。

五、需求库及实验环境

需求库

matplotlib==3.9.0
numpy==1.24.3
scikit_learn==1.4.2
torch==2.3.0
torchvision==0.18.0

硬件环境

CPU：Inter(R) Core(TM) Ultra 7 155H 1.40GHz
RAM:32GB
GPU:NVIDIA GeForce RTX 4060 Laptop GPU

软件环境

操作系统：Windows11
开发工具：PyCharm2023.1.1
虚拟环境：Anaconda
Python 版本：3.11

六、训练结果及分析

6.1 超参数：步长

模型：`torch.optim.Adam`

	0.00001	0.0001	0.0005	0.001	0.0015	0.003	0.005	0.01
Loss:	0.0110	0.0042	0.0023	0.0024	0.0024	0.0051	0.0053	0.0076
Test Accuracy:	91.1100%	97.3100%	98.5000%	98.3700%	97.7900%	95.0000%	95.1200%	93.8000%
Precision:	0.9111	0.9733	0.9849	0.9838	0.9779	0.9504	0.9517	0.9408
Recall:	0.9098	0.9729	0.9850	0.9836	0.9777	0.9497	0.9504	0.9376
F1:	0.9105	0.9731	0.9850	0.9837	0.9778	0.9501	0.9510	0.9392

结论：在 0.0005，0.001 附近，各项指标数据较好，当步长变大/变小时，指标数据出现了不同程度的变差，可以理解为过拟合/欠拟合。

6.2 超参数：优化器

参数按照下表排列

Loss	Test Accuracy	
Precision	Recall	F1

	0.0001			0.001			0.005			0.01		
`torch.optim.Adam`	0.0042	97.31%		0.0024	98.37%		0.0053	95.12%		0.0076	93.80%	
	0.9733	0.9729	0.9731	0.9838	0.9836	0.9837	0.9517	0.9504	0.951	0.9408	0.9376	0.9392
`torch.optim.SGD`	0.0359	26.23%		0.0322	74.09%		0.0131	90.43%		0.0098	90.87%	
	0.3023	0.2633	0.2814	0.7632	0.7332	0.7479	0.9033	0.9031	0.9032	0.9145	0.9072	0.9108
`torch.optim.RMSprop`	0.0032	97.86%		0.0053	97.86%		0.1091	96.01%		1.0133	94.26%	
	0.9786	0.9785	0.9785	0.9785	0.9786	0.9786	0.9602	0.9599	0.96	0.9458	0.9418	0.9438
`torch.optim.Adamax`	0.0065	94.23%0		0.0027	98.09%		0.0051	96.37%		0.008	94.68%	
	0.9422	0.9416	0.9419	0.9809	0.9808	0.9809	0.9639	0.9633	0.9636	0.9469	0.9464	0.9467

结论：从该数据中可以看出，不同模型之间的测试集正确率存在一定差距。其中，`torch.optim.SGD`的正确率较低，在各指标上明显低于另外几个模型，而另外几个模型差距并不是很大。针对这种情况，对于`torch.optim.SGD`优化器增加一个实验。

参数设置为：optimizer = torch.optim.SGD(model.parameters())

通过增加 Epoch 的方式，对其进行实验。

	Loss	Test Accuracy	Precision	Recall	F1
Epoch=1	0.0322	74.5300%	0.7578	0.7374	0.7475
Epoch=2	0.0127	85.9600%	0.8604	0.8570	0.8587
Epoch=3	0.0071	88.6900%	0.8861	0.8850	0.8856
Epoch=4	0.0060	89.7900%	0.8970	0.8963	0.8967
Epoch=5	0.0054	90.6300%	0.9062	0.9045	0.9054

可见：通过提高轮次的方式，可以明显提高各指标。可见提升轮次也是训练的一种方式。

6.3 各模型横向对比

	神经网络(Adam)lr=0.001	决策树	逻辑回归	SVM('linear')	SVM('rbf')	SVM('poly')
Test Accuracy:	98.3700%	87.59%	92.56%	94.04%	97.92%	97.71%
Precision:	0.9733	0.9247	0.9247	0.9399	0.9792	0.9771
Recall:	0.9729	0.9245	0.9245	0.9394	0.9791	0.9769
F1:	0.9731	0.9246	0.9246	0.9397	0.9791	0.9770

结论：总体上来看，神经网络在各指标上仍是最为优秀的。支持向量机核函数为

'rbf'和'poly'次之，然后是逻辑回归和向量机核函数为' linear'，决策树模型最差。这是初步的结论，需要通过更多的模型和调整参数来佐证该结论。

七、总结

本次实验中，使用 PyTorch 对模型进行训练，并且对于步长、优化器、Epoch 数、与几个机器学习模型的比较等进行探究，初步得出了一些结论。但是仍有很多地方有可以优化的的地方。如对于模型的训练数可以再提高一些，从而使结论更为有力。并且也可以选用其他深度学习模型或机器学习模型，与结果进行比较。

参考文献：

[1]周志华.机器学习[M].清华大学出版社,2016.
[2]（美）伊莱·史蒂文斯,（意）卢卡·安蒂加,（德）托马斯·菲曼.PyTorch 深度学习实战.北京.人民邮电出版社,2022.2

附录：程序源码

CNN-number-recogniton.py
实现卷积神经网络识别 MNIST 的代码

```
import torch
import torchvision
from matplotlib import pyplot as plt
from torch.utils.data import DataLoader
from torchvision import transforms, datasets
from torch.nn import (
    Module, Sequential, Conv2d, ReLU, Linear, MaxPool2d, Dropout
)
from sklearn.metrics import confusion_matrix, precision_score, recall_score

# 在有 GPU 的情况下使用 cuda
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 下载数据
data_train = datasets.MNIST(
    root='./data',
    train=True,
    download=True, transform=transforms.Compose([transforms.ToTensor()])
)
data_test = datasets.MNIST(
    root='./data',
    train=False,
    download=True, transform=transforms.Compose([transforms.ToTensor()])
)

# 将数据分批次并随机打乱
data_loader_train = DataLoader(data_train, batch_size=64, shuffle=True)
data_loader_test = DataLoader(data_test, batch_size=64, shuffle=True)


# 模型结构
class Model(Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = Sequential(
            Conv2d(1, 64, kernel_size=3, padding=1),
            ReLU(),
```

```
            Conv2d(64, 128, kernel_size=3, padding=1),
            ReLU(),
            MaxPool2d(stride=2, kernel_size=2)
        )

        self.dense = Sequential(
            Linear(14 * 14 * 128, 1024),
            ReLU(),
            Dropout(p=0.5),
            Linear(1024, 10)
        )

    def forward(self, x):
        x = self.conv1(x)
        x = x.view(x.size(0), -1)
        x = self.dense(x)
        return x


model = Model().to(device)
cost = torch.nn.CrossEntropyLoss()

# optimizer = torch.optim.Adam(model.parameters())  # Adam 优化器的学习率默认为 0.001
# optimizer = torch.optim.Adam(model.parameters(),lr=0.01)  # 采用不同的优化器比较结果
# optimizer = torch.optim.SGD(model.parameters(),lr=0.01)
optimizer = torch.optim.RMSprop(model.parameters(),lr=0.01)
# optimizer = torch.optim.Adamax(model.parameters(),lr=0.005)


number_epochs = 1

for epoch in range(number_epochs):
    running_loss = 0.0
    running_correct = 0
    all_predicted = []
    all_targets = []
    print('Epoch {}/{}'.format(epoch, number_epochs))
    print("-" * 10)
    # print('Epoch: {}/{}'.format(epoch, number_epochs))
    for data, target in data_loader_train:
        data, target = data.to(device), target.to(device)
        outputs = model(data)
```

```
            _, predicted = torch.max(outputs, 1)
            optimizer.zero_grad()
            loss = cost(outputs, target)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            running_correct += torch.sum((predicted == target).type(torch.FloatTensor))
        testing_correct = 0
        for data, target in data_loader_test:
            data, target = data.to(device), target.to(device)
            outputs = model(data)
            _, predicted = torch.max(outputs, 1)
            testing_correct += torch.sum((predicted == target).type(torch.FloatTensor))
            all_predicted.extend(predicted.cpu().numpy())
            all_targets.extend(target.cpu().numpy())
        print("Loss: {:.4f}, Train Accuracy is {:.4f}%, Test Accuracy is {:.4f}%"
              .format(running_loss / len(data_train),
                      100 * running_correct / len(data_train),
                      100 * testing_correct / len(data_test)))
        conf_matrix = confusion_matrix(all_targets, all_predicted)
        # 计算每一个类别的查全率,查准率
        precision = precision_score(all_targets, all_predicted, average=None,
zero_division=0)
        recall = recall_score(all_targets, all_predicted, average=None, zero_division=0)
        # 计算总体的查全率,查准率
        overall_precision = precision_score(all_targets, all_predicted, average='macro',
zero_division=0)
        overall_recall = recall_score(all_targets, all_predicted, average='macro',
zero_division=0)

        for i in range(10):
            print(f"类别{i} precision:{precision[i]:.4f} , recall:{recall[i]:.4f} , "
                  f"F1:{2 * precision[i] * recall[i] / (precision[i] + recall[i]):.4f}")
        print(f"Overall Precision: {overall_precision:.4f} , Overall Recall:
{overall_recall:.4f}"
              f"F1:{2 * overall_precision * overall_recall / (overall_precision +
overall_recall):.4f}")
```

data_conversion.py

对于数据进行预处理，从而可以使用 sklearn 进行训练机器学习模型

```
import numpy as np


def torch_data_to_sklearn(data_loader):
```

```
    images = []
    labels = []
    for data, target in data_loader:
        images.append(data.view(data.size(0), -1).numpy())
        labels.append(target.numpy())
    images = np.concatenate(images, axis=0)
    labels = np.concatenate(labels).ravel()  # 将标签展平为一维数组
    return images, labels
```

decision_tree.py

决策树训练模型

```
    from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score
    from sklearn.tree import DecisionTreeClassifier
    from torchvision import datasets, transforms
    from torch.utils.data import DataLoader
    import numpy as np
    from data_conversion import torch_data_to_sklearn

    # 下载数据
    data_train = datasets.MNIST(
        root='./data', train=True, download=True, transform=transforms.Compose([transforms.ToTensor()])
    )
    data_test = datasets.MNIST(
        root='./data', train=False, download=True, transform=transforms.Compose([transforms.ToTensor()])
    )

    # 将数据分批次并随机打乱
    data_loader_train = DataLoader(data_train, batch_size=64, shuffle=True)
    data_loader_test = DataLoader(data_test, batch_size=64, shuffle=True)

    x_train, y_train = torch_data_to_sklearn(data_loader_train)
    x_test, y_test = torch_data_to_sklearn(data_loader_test)

    tree = DecisionTreeClassifier()
    tree.fit(x_train, y_train)
    prediction = tree.predict(x_test)

    y_test = y_test.reshape(-1, 1)
    prediction = prediction.reshape(-1, 1)
    print(np.sum(y_test == prediction) / len(y_test))
```

```
    conf_matrix = confusion_matrix(y_test, prediction)
    overall_precision = precision_score(y_test, prediction, average='macro', zero_division=0)
    overall_recall = recall_score(y_test, prediction, average='macro', zero_division=0)
    print(f"Overall Precision: {overall_precision:.4f} , Overall Recall: {overall_recall:.4f}"
          f"F1:{2 * overall_precision * overall_recall / (overall_precision + overall_recall):.4f}")
    f1 = f1_score(y_test, prediction, average='macro', zero_division=0)
    print(f"F1 Score: {f1:.4f}")
```

logistic.py

逻辑回归训练模型

```
    from sklearn.linear_model import LogisticRegression
    from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score
    from torchvision import datasets, transforms
    from torch.utils.data import DataLoader
    import numpy as np
    from data_conversion import torch_data_to_sklearn

    # 下载数据
    data_train = datasets.MNIST(
        root='./data', train=True, download=True, transform=transforms.Compose([transforms.ToTensor()])
    )
    data_test = datasets.MNIST(
        root='./data', train=False, download=True, transform=transforms.Compose([transforms.ToTensor()])
    )

    # 将数据分批次并随机打乱
    data_loader_train = DataLoader(data_train, batch_size=64, shuffle=True)
    data_loader_test = DataLoader(data_test, batch_size=64, shuffle=True)

    x_train, y_train = torch_data_to_sklearn(data_loader_train)
    x_test, y_test = torch_data_to_sklearn(data_loader_test)

    lr = LogisticRegression()
    lr.fit(x_train, y_train)
    prediction = lr.predict(x_test)
```

```
    y_test = y_test.reshape(-1, 1)
    prediction = prediction.reshape(-1, 1)
    print(np.sum(y_test == prediction)/len(y_test))
    conf_matrix = confusion_matrix(y_test, prediction)
    overall_precision = precision_score(y_test, prediction, average='macro', zero_division=0)
    overall_recall = recall_score(y_test, prediction, average='macro', zero_division=0)
    print(f"Overall Precision: {overall_precision:.4f} , Overall Recall: {overall_recall:.4f}"
          f"F1:{2 * overall_precision * overall_recall / (overall_precision + overall_recall):.4f}")
    f1 = f1_score(y_test, prediction, average='macro', zero_division=0)
    print(f"F1 Score: {f1:.4f}")
```

SVM.py

支持向量机训练模型

```
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score
from sklearn.svm import SVC
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import numpy as np
from data_conversion import torch_data_to_sklearn

# 下载数据
data_train = datasets.MNIST(
    root='./data', train=True, download=True, transform=transforms.Compose([transforms.ToTensor()])
)
data_test = datasets.MNIST(
    root='./data', train=False, download=True, transform=transforms.Compose([transforms.ToTensor()])
)

# 将数据分批次并随机打乱
data_loader_train = DataLoader(data_train, batch_size=64, shuffle=True)
data_loader_test = DataLoader(data_test, batch_size=64, shuffle=True)

x_train, y_train = torch_data_to_sklearn(data_loader_train)
x_test, y_test = torch_data_to_sklearn(data_loader_test)

svm_model = SVC(kernel='poly')      # kernel='linear'
```

```
svm_model.fit(x_train, y_train)
prediction = svm_model.predict(x_test)

y_test=y_test.reshape(-1,1)
prediction=prediction.reshape(-1,1)
print(np.sum(y_test == prediction)/len(y_test))
conf_matrix = confusion_matrix(y_test, prediction)
overall_precision = precision_score(y_test, prediction, average='macro', zero_division=0)
overall_recall = recall_score(y_test, prediction, average='macro', zero_division=0)
print(f"Overall Precision: {overall_precision:.4f} , Overall Recall: {overall_recall:.4f}"
      f"F1:{2 * overall_precision * overall_recall / (overall_precision + overall_recall):.4f}")

f1 = f1_score(y_test, prediction, average='macro', zero_division=0)
print(f"F1 Score: {f1:.4f}")
```