Quantum Computing
量子计算

题　　目：基于 PyTorch 和 Qiskit 的 QCNN 手写数字识别系统

组　　长：杜雅然

成　　员：杨晓琦　吴之翔　余希源

中国矿业大学计算机科学与技术学院

2025 年　4　月

1、自评表

学号	姓名	手机号	承担任务
08222227	杜雅然	19852089506	设计并实现经典 CNN 中的最大池化层、非线性激活层、全连接层等作为混合模型的经典部分；负责量子-经典混合模型的训练流程；预处理和加载 MNIST 数据集。
08222213	杨晓琦	13145222498	负责设计用于手写数字识别的量子卷积层；实现角度编码和量子态处理逻辑；与其他经典层的接口对接，确保梯度传递和混合训练的正常运行。
08220875	吴之翔	13161572702	开发基于 Tkinter 的 GUI 界面，支持鼠标手写输入和实时显示；实现画布清除、数字识别触发按钮等功能；将用户输入图像预处理为模型可接受的格式：28x28 像素灰度图。
08222445	余希源	15178487380	整合量子模型 Qiskit、经典模型 PyTorch 和 UI 界面（Tkinter）的代码;在可视化界面中加载训练好的模型参数；编写最终主程序入口，让这个界面可以执行起来。

2、其他情况说明（比赛、文章等）

序号	说明
1	参加了司南杯高校赛道
2	在课堂上以 PPT 形式汇报了《量子卷积层的具体实现》

3、成绩评定

学号	姓名	建议成绩（学生）	评定成绩(教师)
08222227	杜雅然	97	
08222213	杨晓琦	95	
08220875	吴之翔	95	
08222445	余希源	95	

4、教师评语

目　　录

基于 PyTorch 与 Qiskit 的量子卷积神经网络（QCNN）

——实现鼠标手写交互式数字识别系统

1. 项目背景与概述

1.1 研究背景

在当今科技飞速发展的时代，人工智能和量子计算领域取得了令人瞩目的进展，量子机器学习（Quantum Machine Learning, QML）作为二者融合的新兴领域，正逐渐成为科研热点。传统机器学习方法在处理大规模数据集时，面临着训练效率和性能瓶颈。例如，在图像分类任务中，随着数据集规模的不断增大，传统机器学习模型的训练时间大幅增加，且模型的泛化能力难以进一步提升。

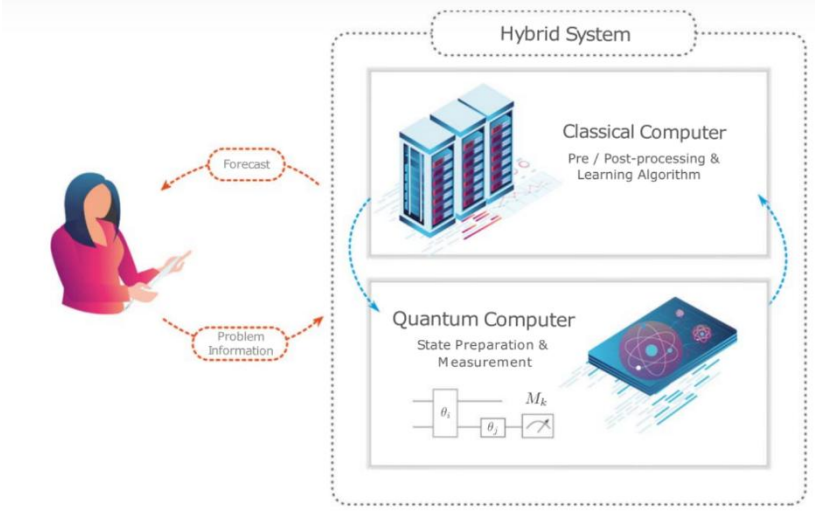

图 1 量子计算与经典卷积神经网络的混合模型

卷积神经网络作为深度学习的代表性算法之一，在图像处理领域表现卓越。它通过卷积层、池化层和全连接层的组合，能够自动提取图像的特征，从而实现高效的图像分类和识别。然而，随着数据维度的不断增加，经典卷积神经网络在处理高维数据时，计算复杂度也随之急剧上升。

量子计算的出现为解决上述问题带来了新的希望。量子计算利用量子

比特（qubit）作为基本运算单元，具有并行计算能力和指数级加速特性。量子卷积神经网络（Quantum Convolutional Neural Network, QCNN）正是结合了量子计算与经典卷积神经网络的优势，形成的一种混合模型。在图像分类任务中，QCNN 能够利用量子比特的叠加态和纠缠特性，更高效地处理高维数据，展现出更强的泛化能力和更高的训练效率。这种优势在处理大规模、复杂的图像数据集时尤为明显，为图像识别领域的发展开辟了新的道路。

1.2 系统概述

本小组成员在深入学习《量子计算》课程后，积累了丰富的量子电路与量子编程知识。基于这些知识，我们开发了一个简易的基于 PyTorch 和 Qiskit 的 QCNN 手写数字识别系统。该系统具备高达 95% 的识别准确率，实现了用户通过鼠标手写数字，系统实时识别并给出结果的功能。

在系统实现过程中，我们借助 PyTorch 框架强大的神经网络构建能力，搭建了经典卷积神经网络部分。同时，利用 Qiskit 工具包来设计、模拟和运行量子算法，将两者有机结合，构建出 QCNN 模型。此外，我们使用经典的 MNIST 手写数字图像数据集对模型进行训练和测试，确保模型的可靠性和有效性。最后，通过 Tkinter 库设计可视化界面，为用户提供了便捷的交互体验。

1.3 开发环境

表 1 系统开发环境

开发软件	版本
PyCharm	2023.2.1
Python	3.7
PyTorch	1.13.1
Qiskit	0.43.3

2. 准备工作及基础知识

2.1 MNIST 数据集介绍

MNIST 数据集是机器学习领域中经典的手写数字图像数据集，其包含 60000 张训练样本和 10000 张测试样本。每张图片均为 28×28 的灰度图像，涵盖了 0-9 这 10 个数字类别。这些图像由不同的人书写，具有丰富的手写风格和形态变化，为训练和测试手写数字识别模型提供了良好的数据基础。

在实际应用中，MNIST 数据集被广泛用于评估各种机器学习算法在图像分类任务上的性能。例如，在训练一个简单的神经网络模型时，可以使用 MNIST 数据集来验证模型的准确性和泛化能力。通过对 MNIST 数据集中图像的特征提取和分类学习，模型能够学习到不同数字的特征模式，从而对新的手写数字图像进行准确分类。

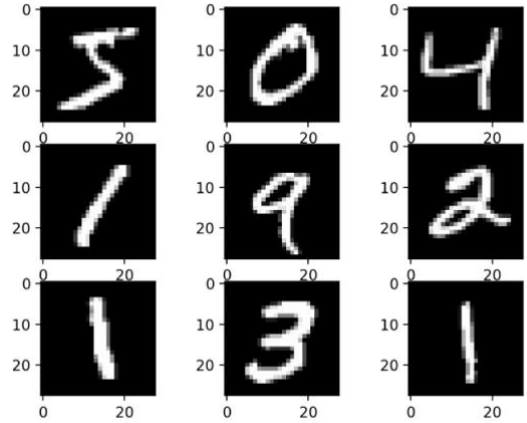

图 2 MNIST 数据集样例

2.2 量子比特

量子计算的基本运算单元是量子比特（qubit），它与经典比特不同，具有独特的量子特性。一个 qubit 的状态是一个二维复数空间的矢量，其两个极化状态 $|0\rangle$ 和 $|1\rangle$ 对应经典状态 0 和 1，但 qubit 还可以处于这两个状态的叠加态。量子比特的状态可以表示为：

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{1}$$

其中， α 和 β 为复数，满足 $|\alpha|^2 + |\beta|^2 = 1$ 。

2.3 实现 QCNN 模型所到的典型量子门

2.3.1 Hadamard 门（H 门）

(1) 作用：

H 门是将量子比特置于叠加态。在量子计算中，叠加态是实现并行计算的基础。通过 H 门的作用，可以使量子比特从初始的基态转变为叠加态，为后续的量子计算提供更丰富的状态空间，从而增加量子算法的计算能力。

(2) 数学表示：

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \tag{2}$$

(3) 作用在基态上的效果：

$$\begin{aligned} H|1\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\\ H|0\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\end{aligned} \tag{3}$$

2.3.2 CNOT 门（控制非门）

(1) 作用

CNOT 门用于在两个量子比特之间引入纠缠。纠缠是量子力学中的一种特殊现象，使得两个或多个量子比特之间存在非经典的关联。通过 CNOT 门实现的 “条件翻转” 操作，能够实现量子比特之间的信息传递和协同计算，是构建复杂量子算法的重要基础。

(2) 数学表示

$$CNOT = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix} \tag{4}$$

(3) 效果

控制位为 |0⟩ 时，目标位保持不变。

控制位为 |1⟩ 时，目标位翻转（0 变 1，1 变 0）。

例如，对于两个量子比特的状态 |01⟩，经过 CNOT 门作用后，状态仍为 |01⟩；而对于状态 |11⟩，经过 CNOT 门作用后，状态变为 |10⟩。

2.3.3 RX 门（绕 X 轴旋转门）

(1) 作用

RX 门是对量子比特在 Bloch 球的 X 轴上进行旋转，旋转幅度由参数 θ 控制。通过调整 θ 的值，可以精确地改变量子比特的状态。

(2) 数学表示

RX 门由 Pauli-X 矩阵作为生成元生成，其矩阵形式为：

$$X = \sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{5}$$

则 RX 门的表示为：

$$\begin{aligned} R_x(\theta) &= e^{-\frac{\theta}{2}iX} = \cos\left(\frac{\theta}{2}\right) I - i\sin\left(\frac{\theta}{2}\right) X \\ &= \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix} \end{aligned} \tag{6}$$

(3) 作用在基态上的效果：

$$\begin{aligned} R_x(\theta)|1\rangle &= -\mathrm{i}\sin\left(\frac{\theta}{2}\right)|0\rangle + \cos\left(\frac{\theta}{2}\right)|1\rangle \\ R_x(\theta)|0\rangle &= \cos\left(\frac{\theta}{2}\right)|0\rangle - \mathrm{i}\sin\left(\frac{\theta}{2}\right)|1\rangle \end{aligned} \tag{7}$$

(4) 图形化表示：

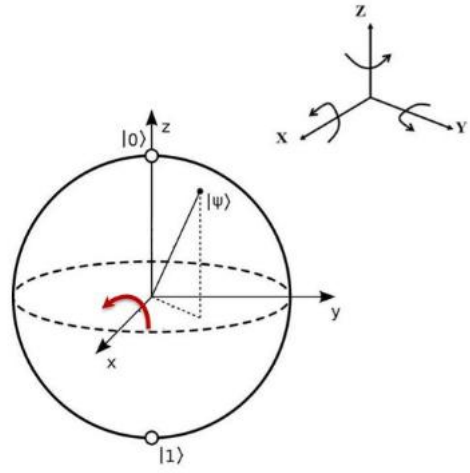

图 3 RX 门的 Bloch 球

2.3.4 RY 门（绕 Y 轴旋转门）

(1) 作用

对量子比特在 Bloch 球的 Y 轴上进行旋转，幅度由参数 θ 控制。

(2) 数学表示：

RY 门由 Pauli-Y 矩阵作为生成元生成，其矩阵形式为：

$$Y = \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \tag{8}$$

则 RY 门的表示为：

$$\begin{aligned} R_y(\theta) &= e^{-\frac{\theta}{2}iY} = \cos\left(\frac{\theta}{2}\right) I - isin\left(\frac{\theta}{2}\right) Y \\ &= \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -sin\left(\frac{\theta}{2}\right) \\ sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix} \end{aligned} \tag{9}$$

(3) 作用在基态上的效果：

$$\begin{aligned} R_y(\theta)|1\rangle &= -\sin\left(\frac{\theta}{2}\right)|0\rangle + \cos\left(\frac{\theta}{2}\right)|1\rangle \\ R_y(\theta)|0\rangle &= \cos\left(\frac{\theta}{2}\right)|0\rangle + \sin\left(\frac{\theta}{2}\right)|1\rangle \end{aligned} \tag{10}$$

(4) 图形化表示：

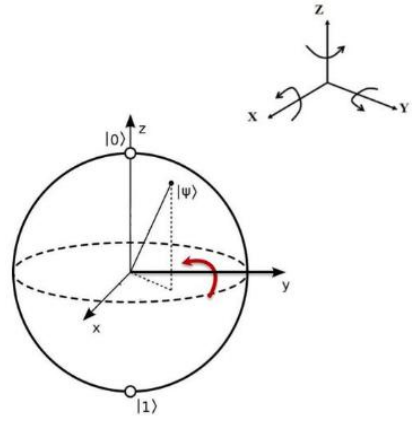

图 4 RY 门的 Bloch 球

2.3.5 RZ 门（绕 Z 轴旋转门）

(1) 作用

对量子比特在 Bloch 球的 Z 轴上进行旋转，幅度由参数 θ 控制，不会导致概率振幅的变化，只会改变相位。

(2) 数学表示

RZ 门又称相位转换门，由 Pauli-Z 矩阵作为生成元生成，其矩阵形式为：

$$Z = \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{11}$$

则 RZ 门的表示为：

$$\begin{aligned} R_z(\theta) &= e^{-\frac{\theta}{2}iZ} = \cos\left(\frac{\theta}{2}\right) I - isin\left(\frac{\theta}{2}\right) Z \\ &= \begin{bmatrix} e^{-\frac{\theta}{2}i} & 0 \\ 0 & e^{\frac{\theta}{2}i} \end{bmatrix} = e^{-\frac{\theta}{2}i} \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix} \end{aligned} \tag{12}$$

(3) 作用在基态上的效果：

$$\begin{aligned} R_z(\theta)|1\rangle &= e^{i\theta}|1\rangle \\ R_z(\theta)|0\rangle &= |0\rangle \end{aligned} \tag{13}$$

(4) 图形化表示：

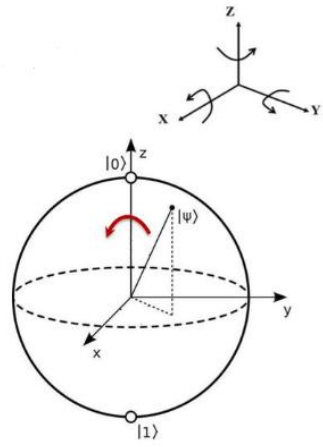

图 5 RZ 门的 Bloch 球

2.3.5 测量门

测量门的作用是将量子态坍缩回经典态，从而读取最终结果。在量子计算中，量子态是一种概率性的状态描述，测量操作使得量子态以一定的概率转换为经典数据，是量子计算与经典计算的接口。由于测量具有随机性，每次测量结果可能不同，因此在 QCNN 中通常会多次测量取平均值，以确保结果稳定。

3. QCNN 模型具体设计与实现

3.1 QCNN 的量子电路实现

QCNN 的量子电路采用 4 个量子比特，电路结构如下图所示：

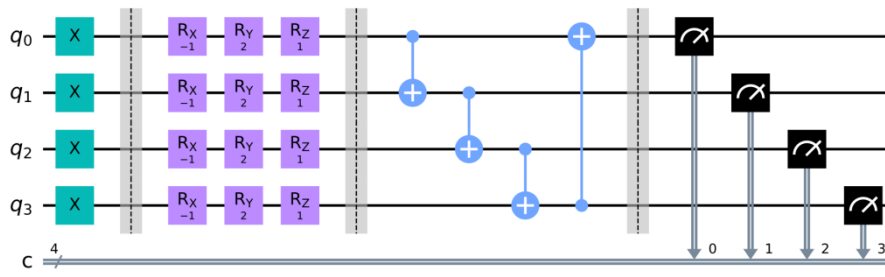

图 6 QCNN 量子电路结构

该电路结构经过以下三步来实现量子卷积层：

- **编码**：使用 RX、RY、RZ 门将经典数据转换为量子态
- **纠缠**：在不同量子比特之间加 CNOT 门来提取特征
- **测量**：对所有量子比特进行测量

对量子比特的测量结果进行分析，得出了预期投影概率、预期密度矩阵和预期振幅的可视化结果，同时也得出了测量 1024 后的统计结果。

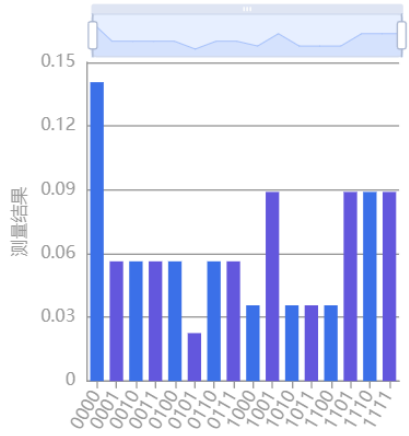

图 7 预期投影概率

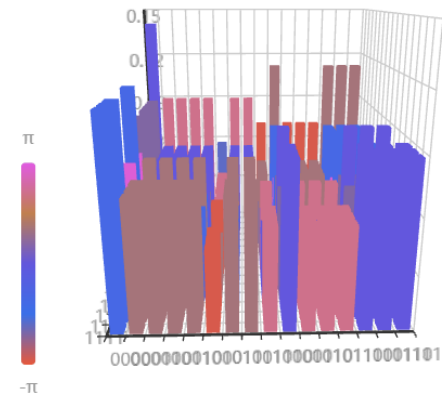

图 8 预期密度矩阵

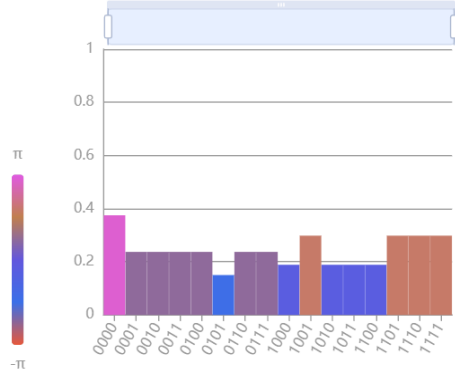

图 9 预期振幅

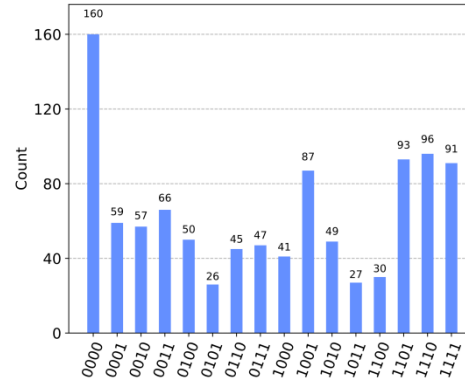

图 10 对量子电路统计 1024 次后的结果

3.2 QCNN 网络结构

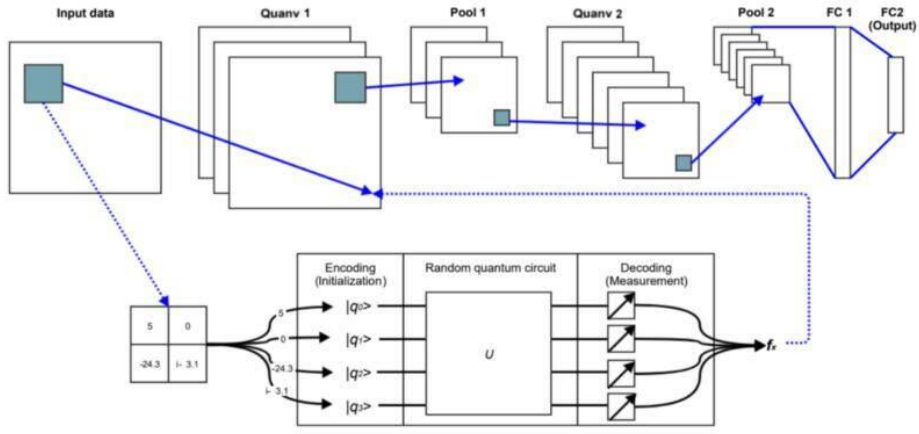

图 11 QCNN 网络结构图

QCNN 的网络结构主要包括以下四部分：

● **输入数据：**采用 MNIST 数据集作为输入，每张图片为 28×28 的灰度图像，输入维度为 (batch_size, 1, 28, 28)。在实际训练和测试过程中，数据会按照批次（batch）进行输入，这样可以提高训练效率。

● **量子卷积层：**对输入的 Tensor 数据通过旋转门角度编码转换成量子态，参数化大小为 2x2 的卷积核，再加入 CNOT 门进行量子纠缠，最后使用测量门对结果进行 1024 次统计。

● **经典卷积层：**利用大小为 3x3 的卷积核提取图像的低级和高级特征，随着网络深度的加深，每个卷积核的感受野逐渐增大，能够更充分地提取图像特征。

● **池化层：**对特征图(feature map)进行降采样，从而减小网络的模型参数量和计算成本。

● **全连接层：**将特征映射到类别得分，输出的 10 个数可以分别看作为 0-9 的概率。

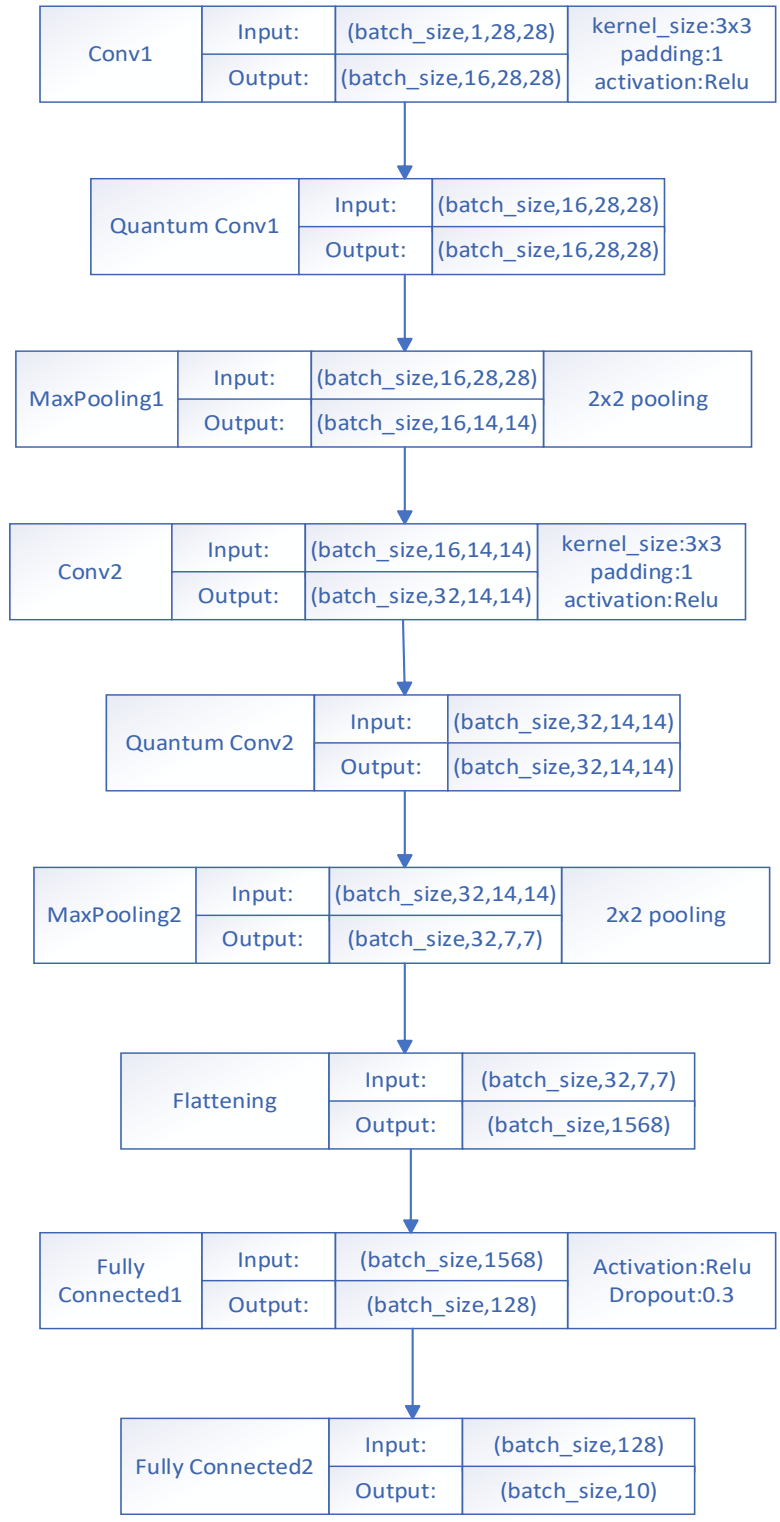
Conv1
Input:
(batch_size,1,28,28)
Output:
(batch_size,16,28,28)
kernel_size:3x3
padding:1
activation:Relu
Quantum Conv1
Input:
(batch_size,16,28,28)
Output:
(batch_size,16,28,28)
MaxPooling1
Input:
(batch_size,16,28,28)
Output:
(batch_size,16,14,14)
2x2 pooling
Conv2
Input:
(batch_size,16,14,14)
Output:
(batch_size,32,14,14)
kernel_size:3x3
padding:1
activation:Relu
Quantum Conv2
Input:
(batch_size,32,14,14)
Output:
(batch_size,32,14,14)
MaxPooling2
Input:
(batch_size,32,14,14)
Output:
(batch_size,32,7,7)
2x2 pooling
Flattening
Input:
(batch_size,32,7,7)
Output:
(batch_size,1568)
Fully Connected1
Input:
(batch_size,1568)
Output:
(batch_size,128)
Activation:Relu
Dropout:0.3
Fully Connected2
Input:
(batch_size,128)
Output:
(batch_size,10)

图 12 QCNN 模型图

4. 系统具体架构设计与实现

本系统采用模块化设计，主要模块包括：

- 数据处理模块（`data_loader.py`）：完成数据预处理、归一化及加载
- 模型模块（`model.py`）：定义 QCNN 网络结构
- 训练模块（`train.py`）：负责模型训练及参数保存
- 测试模块（`test.py`）：实现模型测试及性能评估

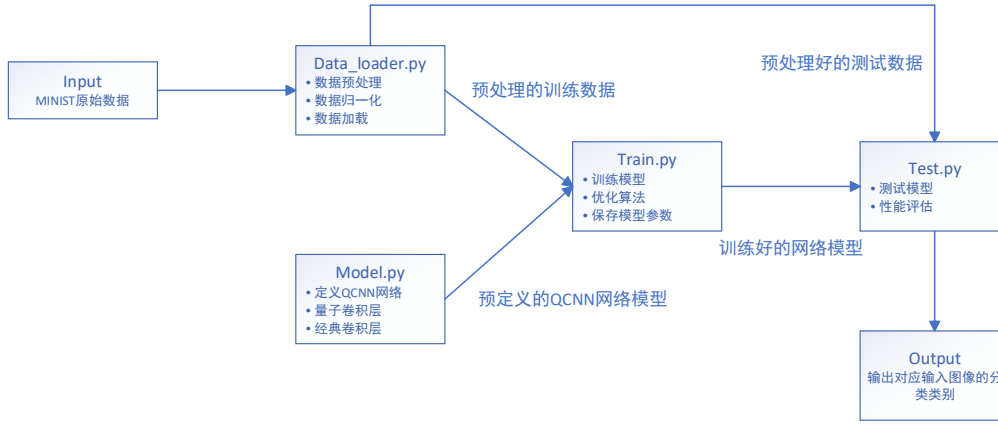

图 13 系统模块结构图

4.1 数据处理模块

数据处理模块使用 PyTorch 的 torchvision 库加载 MNIST 数据集。关键步骤包括：

(1) 将图像归一化至 [0, 1] 范围，使图像数据的分布更加均匀，避免某些特征因数值过大或过小而对模型训练产生不利影响。

(2) 将数据转换为 PyTorch 的 Tensor 类型，方便在模型中进行数据的传递、计算和处理。

(3) 在数据的读取过程中，每次加载一个小批量（batch）的样本用于训练，而非一次性加载整个数据集。这样做可以减少内存占用，提高训练效率。

4.2 模型模块

由于 3.2 部分已经详细介绍模型整体框架，所以这里仅介绍量子卷积层。

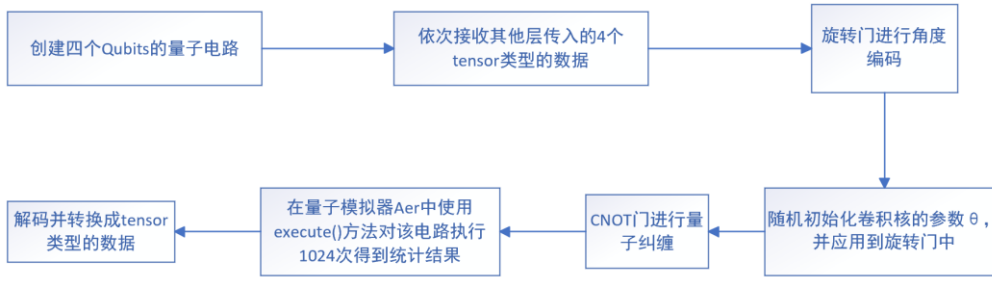

图 14 量子卷积层的工作流程

4.3 训练模块

● 损失函数

在本网络中我们使用采用交叉熵损失函数（CrossEntropyLoss），其公式如下：

$$loss(x,y) = -\log\left(\frac{e^{x_y}}{\sum_k e^{x_k}}\right) \tag{14}$$

其中：

x：代表模型的输出，由于 MNIST 为 0-9 的十分类问题，因此大小为 $(N,10)$的张量

y：代表真实标签，即每个数据集对应的真实数字

k：计算 softmax 分母时使用的一个索引变量，遍历 x 中的每个数据。

● 优化器

我们使用经典的优化器——Adam 优化器。此优化器能够为每个参数自动调整学习率，它利用一阶矩（梯度的均值）和二阶矩（梯度的未中心化方差）估计来动态调整学习率，对于 MNIST 这样的图像分类问题，不同特征的学习速度差异很大，Adam 能自动适应这些变化，提高收敛效率。

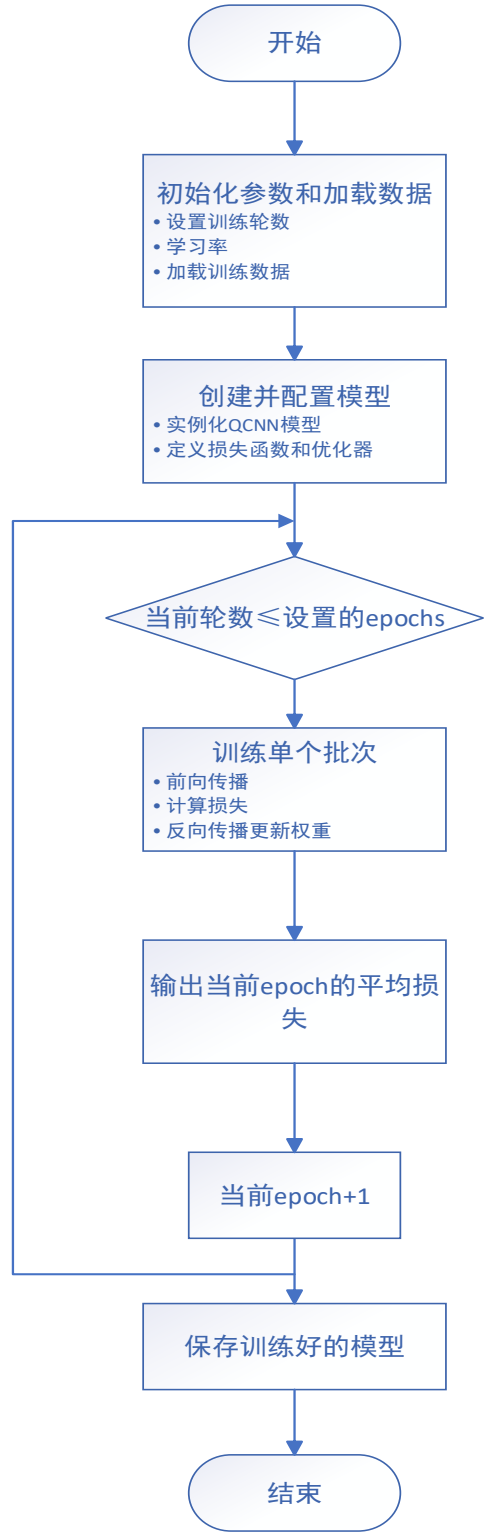
开始
初始化参数和加载数据
• 设置训练轮数
• 学习率
• 加载训练数据
创建并配置模型
• 实例化QCNN模型
• 定义损失函数和优化器
当前轮数≤设置的epochs
训练单个批次
• 前向传播
• 计算损失
• 反向传播更新权重
输出当前epoch的平均损失
当前epoch+1
保存训练好的模型
结束

图 15 训练模块流程图

4.4 测试模块

测试模块加载训练好的模型参数，对测试集进行性能评估，输出模型准确率及混淆矩阵。

5. 代码实现

5.1 数据预处理

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

4 个用法
def get_data_loaders(batch_size=32):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize( mean: (0.5,), std: (0.5,))
    ])

    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(train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return train_loader, test_loader
```

图 16 加载数据

5.2 模型预定义

```
class QuantumConvLayer(nn.Module):
    def __init__(self):
        super(QuantumConvLayer, self).__init__()
        self.num_qubits = 4  # 使用4个量子比特

        self.rx_params = nn.Parameter(torch.randn(self.num_qubits))
        self.ry_params = nn.Parameter(torch.randn(self.num_qubits))
        self.rz_params = nn.Parameter(torch.randn(self.num_qubits))
```

图 17 初始化量子门

```python
def quantum_circuit(self, data):
    """使用4个量子比特 + RX、RY、RZ、CNOT 的电路设计"""
    qc = QuantumCircuit(self.num_qubits)

    # 数据编码 (RX + RY + RZ 编码)
    for i in range(self.num_qubits):
        qc.rx(self.rx_params[i], i)
        qc.ry(self.ry_params[i], i)
        qc.rz(self.rz_params[i], i)

    # 纠缠层 (CNOT 门)
    qc.cx(0, 1)
    qc.cx(1, 2)
    qc.cx(2, 3)
    qc.cx(3, 0)

    qc.measure_all()
    return qc
```

图 18 定义量子电路

```python
def forward(self, x):
    batch_size, _, height, width = x.size()
    output = []

    for i in range(0, height, 2):
        for j in range(0, width, 2):
            patch = x[:, :, i:i + 2, j:j + 2]
            patch = patch.flatten(1)

            simulator = Aer.get_backend('aer_simulator')
            patch_output = []

            for data in patch:
                circuit = self.quantum_circuit(data.tolist()[:4])
                job = execute(circuit, simulator, shots=1024)
                result = job.result().get_counts()

                output_value = sum(int(key, 2) * count for key, count in result.items()) / 1024
                patch_output.append(output_value)

            output.append(torch.tensor(patch_output, dtype=torch.float32))

    output = torch.stack(output).view( *shape: batch_size, -1, height // 2, width // 2)
    return output.to(x.device)
```

图 19 量子卷积的前向传播

```
class QCNN(nn.Module):
    def __init__(self):
        super(QCNN, self).__init__()

        self.conv1 = nn.Conv2d( in_channels: 1,  out_channels: 16, kernel_size=3, stride=1, padding=1)
        self.qconv1 = QuantumConvLayer()

        self.conv2 = nn.Conv2d( in_channels: 16,  out_channels: 32, kernel_size=3, stride=1, padding=1)
        self.qconv2 = QuantumConvLayer()

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(32 * 7 * 7,  out_features: 128)
        self.dropout = nn.Dropout(0.3)
        self.fc2 = nn.Linear( in_features: 128,  out_features: 10)
```

图 20 初始化量子卷积网络模型

```
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.qconv1(x)
        x = self.pool(x)

        x = F.relu(self.conv2(x))
        x = self.qconv2(x)
        x = self.pool(x)

        x = x.view(x.size(0), -1)  # Flatten
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        return self.fc2(x)
```

图 21 QCNN 的前向传播

5.3 训练模型

```python
def train_qcnn(num_epochs=10, learning_rate=0.001):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = QCNN().to(device)
    train_loader, _ = get_data_loaders(batch_size=32)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss/len(train_loader):.4f}")

    torch.save(model.state_dict(), f"qcnn_model.pth")
    print("Training complete. Model saved as 'qcnn_model.pth'")
train_qcnn()
```

图 22 QCNN 的训练

5.4 测试

```python
def test_qcnn():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = QCNN().to(device)
    model.load_state_dict(torch.load( f: "qcnn_model.pth", map_location=device))
    model.eval()

    _, test_loader = get_data_loaders(batch_size=32)
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f'Test Accuracy: {100 * correct / total:.2f}%')


test_qcnn()
```

图 23 QCNN 的测试

5.5 系统 UI 界面

```python
class DrawingCanvas(tk.Canvas):
    def __init__(self, master, **kwargs):
        super().__init__(master, **kwargs)
        self.last_x, self.last_y = None, None
        self.bind("<B1-Motion>", self.paint)
        self.bind("<ButtonRelease-1>", self.reset)

    1 个用法
    def paint(self, event):
        x, y = event.x, event.y
        if self.last_x and self.last_y:
            self.create_line(self.last_x, self.last_y, x, y, width=8, fill="black", capstyle=tk.ROUND, smooth=tk.TRUE)
        self.last_x, self.last_y = x, y

    1 个用法
    def reset(self, event):
        self.last_x, self.last_y = None, None

    1 个用法
    def clear(self):
        self.delete("all")
```

图 24 UI 界面的打印、重置、清除功能

```
    def get_image(self):
        img = Image.new( mode: "L",  size: (self.winfo_width(), self.winfo_height()),  color: "white")
        draw = ImageDraw.Draw(img)
        for item in self.find_withtag('all'):
            x0, y0, x1, y1 = self.coords(item)
            draw.line( xy: [x0, y0, x1, y1], fill="black", width=8)
        img_resized = img.resize( size: (28, 28), Image.LANCZOS)
        return img_resized.convert('L')

root = tk.Tk()
root.title("手写数字识别")
root.configure(bg=BG_COLOR)

canvas = DrawingCanvas(root, width=280, height=280, bg="white", highlightthickness=2, highlightbackground="#546E7A")
canvas.pack(pady=10, padx=20)

result_label = tk.Label(root, text="识别的数字: ", bg=BG_COLOR, font=FONT_STYLE)
result_label.pack(pady=5)

processed_img_label = tk.Label(root, bg=BG_COLOR)
processed_img_label.pack(pady=5)
```

图 25 获取鼠标手写图像

```
def predict_image():
    image = canvas.get_image()
    input_tensor = transform(image).unsqueeze(0)

    img_tk = ImageTk.PhotoImage(image.resize((56, 56)).convert('RGB'))
    processed_img_label.configure(image=img_tk)
    processed_img_label.image = img_tk

    with torch.no_grad():
        output = model(input_tensor)
    _, predicted_class = torch.max(output, 1)
    result = predicted_class.item()

    result_label.config(text=f"预测的数字: {result}")
    messagebox.showinfo( title: "预测的结果",  message: f"识别的数字: {result}")

frame = tk.Frame(root, bg=BG_COLOR)
frame.pack(pady=10)

clear_button = tk.Button(frame, text="清空", width=10, command=canvas.clear, bg=BUTTON_COLOR, fg=TEXT_COLOR, font=FONT_STYLE)
clear_button.grid(row=0, column=0, padx=10)

predict_button = tk.Button(frame, text="识别", width=10, command=predict_image, bg=BUTTON_COLOR, fg=TEXT_COLOR, font=FONT_STYLE)
predict_button.grid(row=0, column=1, padx=10)

root.mainloop()
```

图 26 识别用户鼠标手写的数字

6. 实验与结果分析

6.1 训练过程

我们训练了 50 个 epoch，在每个 epoch 训练过程中，QCNN 模型的损失逐渐下降，表现出较好的收敛性。

在相同设置的条件下，我们同时训练了传统的 CNN 和全连接网络，并从损失下降趋势和准确率两方面对上述网络的训练结果进行了比较分析。

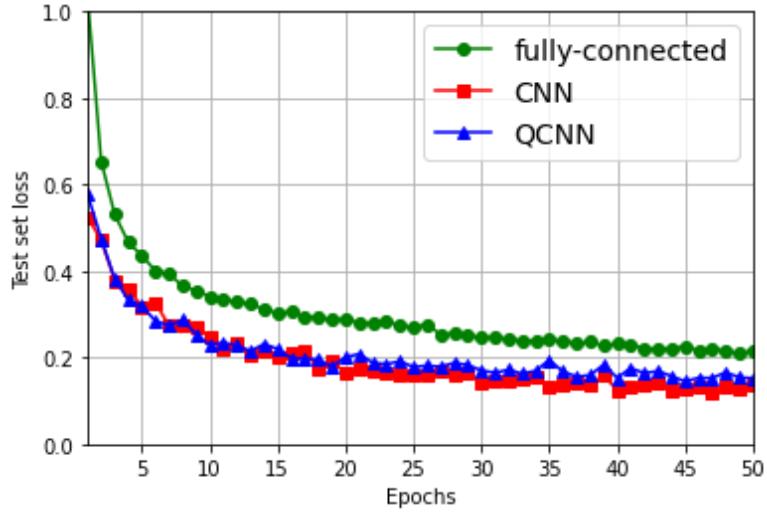

图 27 三种网络的损失下降趋势比较

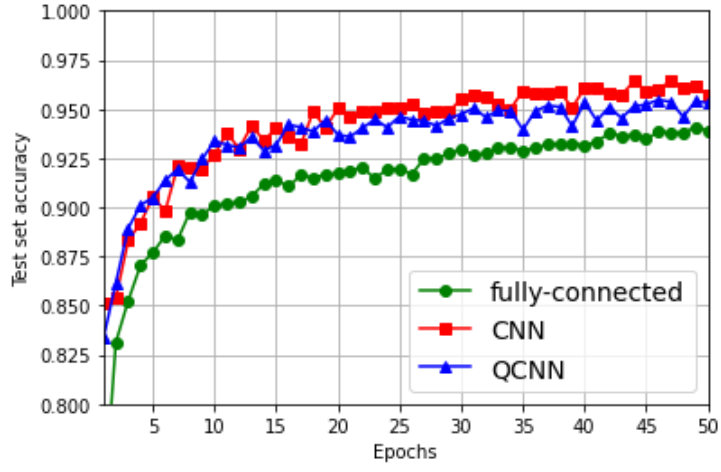

图 28 三种网络的准确率比较

由上图可知，在训练过程中损失的大小和下降趋势方面，CNN 和 QCNN 的表现效果大致一样，都优于 FC 网络。在准确率方面，CNN 效果最优，QCNN 次优，都表现出较高的识别性能。

6.2 测试结果

测试中，QCNN 模型在 MNIST 数据集上的准确率达到 95%，性能良好。混淆矩阵如下：

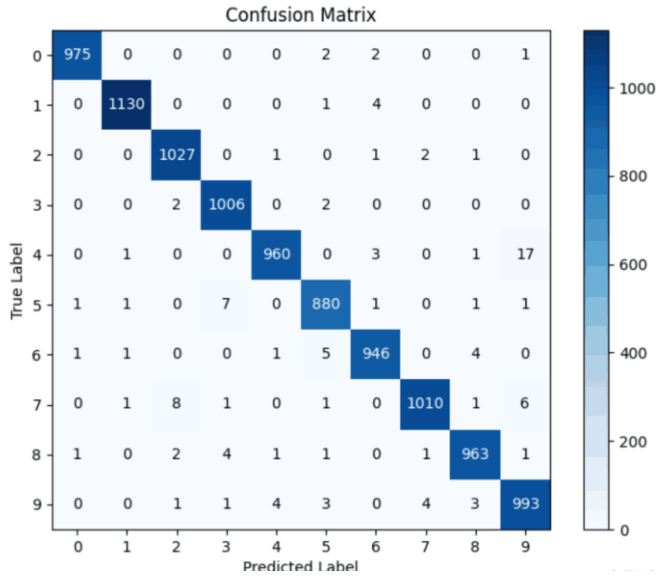

图 30 测试结果混淆矩阵

由该混淆矩阵可以看出，绝大多数数字被正确识别，特别是数字 1 和 8，几乎没有混淆。然而，数字 4 和 9 之间的混淆较为明显，说明模型在区分形状相近的数字时存在挑战。

7. 完整系统界面展示

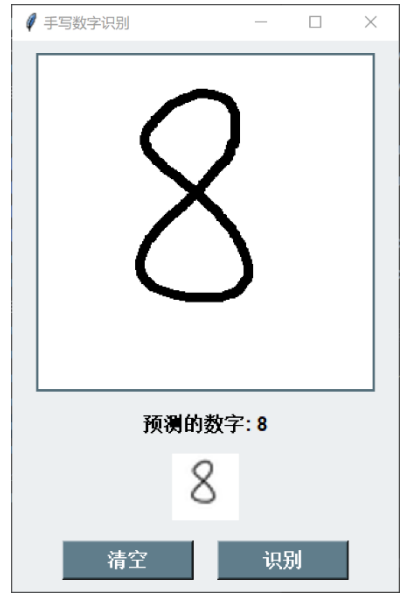

图 31 系统效果图

8. 系统不足与改进

不足：当前基于 qcnn 的手写数字识别系统完全依赖经典计算机模拟量子行为，面临指数级资源消耗。受此限制，无法实现深层量子电路，只能设计较为简单的模型架构，因此相较于传统 CNN 模型，该模型训练效果较差，在测试集上的精度较低。

表 2 资源消耗具体表现

量子比特数	内存消耗	单次前向传播时间
4	128MB	0.3s
6	2GB	2.7s
8	32GB	28.4s

改进：可以使用**本源量子**平台，这个平台允许用户通过云服务直接访问真实的量子计算机，并且提供的量子模拟器经过了特别优化，可以更好地支持大规模量子态的模拟。

9. 参考文献

[1] 向高峰.基于量子卷积神经网络的量子经典混合目标检测模型的研究[D].南京信息工程大学,2024.DOI:10.27248/d.cnki.gnjqc.2024.001079.

[2] 范兴奎,刘广哲,王浩文,等.基于量子卷积神经网络的图像识别新模型[J].电子科技大学学报,2022,51(05):642-650.

[3] 窦通.基于量子—经典混合架构的卷积神经网络优化研究[D].华南理工大学,2022.DOI:10.27151/d.cnki.ghnlu.2022.002427.