

# 浙江大学

## 量子计算理论基础与软件系统实验报告

课程名称: 量子计算理论基础与软件系统

作业名称: 量子机器学习

姓名: 周楠

学号: 3220102535

电子邮箱: 3220102535@zju.edu.cn

联系电话: 19858621101

指导教师: 卢丽强

2024 年 12 月 9 日

# 目录

- 0.1 一. 实验目的和要求 ..... 1
- 0.2 二. 实验环境..... 1
- 0.3 三. 问题分析..... 2
- 0.4 四. 算法原理..... 4
  - 4.1 量子神经网络（QNN） ..... 4
  - 4.2 构建与训练量子神经网络（QNN） ..... 5
- 0.5 五. 技术实现..... 6
  - 5.1 数据加载与预处理 ..... 6
  - 5.2 构建量子电路 ..... 8
  - 5.3 构建模型 ..... 9
  - 5.4 训练与评估 ..... 13
- 0.6 六. 实验测试..... 21
- 0.7 七. 仿照经典神经网络的核心组件，设计完成对应的量子模块..... 25
  - 7.1 混合模型 HybridModel ..... 25
  - 7.2 QuantumConvolutionLayer ..... 26
  - 7.3 QuantumPseudoLinearLayer ..... 29
  - 7.4 QuantumClassifier ..... 31
  - 7.5 QuantumBackbone ..... 33
  - 7.6 测试 ..... 34
  - 7.7 结果分析 ..... 37
- 0.8 八. 总结与心得 ..... 46

## 0.1 一. 实验目的和要求

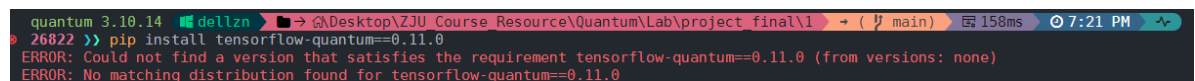
1. 针对 MNIST 数据集，选择一种量子机器学习算法（QSVM、QNN 等）完成分类任务。设置不同的分类目标数量（二分类、四分类、十分类），比较算法效果。使用经典 SVM 和 CNN 算法完成相同任务，比较模型参数量、模型精度、模型收敛速度等指标。
2. 选做：仿照经典神经网络的核心组件，设计完成对应的量子模块，并测试其效果可参考量子机器学习编程框架：PennyLane [https://pennylane.ai/qml/demos/tutorial\\_variational\\_classifier](https://pennylane.ai/qml/demos/tutorial_variational_classifier)

## 0.2 二. 实验环境

1. 操作系统：Ubuntu 24.04 LTS Windows Subsystem for Linux 2
2. 编程语言：Python 3.10
3. 深度学习框架：tensorflow 2.15.0
4. 量子计算框架：tensorflow-quantum 0.11.0

```
1 conda create -n qml python=3.10
2 conda activate qml
3 pip install tensorflow==2.15.0
4 pip install tensorflow-quantum==0.11.0
```

经测试，tensorflow-quantum 0.11.0 与 tensorflow 2.15.0 兼容。但是目前仅提供 linux 环境下的 tensorflow-quantum，因此需要使用 WSL2 来运行实验。在 windows 环境下，安装 tensorflow-quantum 0.11.0 会报错。



```
quantum 3.10.14 dellzn C:\Desktop\ZJU Course Resource\Quantum\Lab\project final\1 → (main) 158ms 7:21 PM
26822 >> pip install tensorflow-quantum==0.11.0
ERROR: Could not find a version that satisfies the requirement tensorflow-quantum==0.11.0 (from versions: none)
ERROR: No matching distribution found for tensorflow-quantum==0.11.0
```

图 1: tensorflow-quantum 0.11.0 在 windows 环境下安装报错

## 0.3 三. 问题分析

### 1. 数据集处理:

- **MNIST 数据集:** MNIST 数据集是一个经典的手写数字数据集, 包含了 28x28 像素的灰度图像。这些图像被标记为 0-9 之间的数字, 是机器学习和深度学习中常用的图像分类任务。数据集分为 60,000 张训练图像和 10,000 张测试图像, 目标是将输入的图像分类为 10 个类别 (0-9)。
- **任务划分:**
  - 二分类: 例如, 将数字 3 与数字 6 区分开来。这样的任务有助于展示量子算法在简单任务中的表现, 并突出其在不同规模问题下的优势与局限性。
  - 四分类: 例如, 选择数字 0, 1, 2 和 3 作为分类目标。这类任务增加了任务的复杂性, 测试量子神经网络在处理多类别问题时的能力。
  - 十分类: 将任务设置为标准的 MNIST 分类任务, 目标是识别 10 个数字 (0-9)。这是最具挑战性的数据集划分, 测试量子算法与经典算法的真实表现。

### 2. 量子机器学习算法:

- **量子神经网络 (QNN):** QNN 结合了经典神经网络的思想, 并在此基础上引入了量子计算的特性。量子比特 (qubit) 可以表示更复杂的超位置态, 这使得量子神经网络具有比经典神经网络更高的表达能力。量子电路通过量子门的参数化来进行学习和优化。
- **量子电路设计:** QNN 设计依赖于量子电路和量子门的构建, 通过对量子比特的操作来实现数据的分类。通常, 量子电路可以包含多个层次的量子门, 其中每个门有一个可优化的参数 (例如旋转角度)。

### 3. 经典机器学习算法:

- **经典 SVM (支持向量机):** SVM 是一种经典的监督学习算法, 常用于二分类问题。SVM 的核心思想是寻找一个最优超平面, 使得不同类别的数据点被正确分类。通过核技巧, SVM 可以处理非线性数据。对于多分类任务, 可以使用一对一或一对多的方式将问题分解为多个二分类任务。
- **经典 CNN (卷积神经网络):** CNN 是一种强大的深度学习架构, 特别适用于图像处理任务。CNN 通过卷积层自动提取图像特征, 并通过池化层降低计算复杂度。CNN 可以通过多个卷积和全连接层实现非常高的分类准确率, 尤其是在图像分类中。

#### 4. 性能比较:

- 模型参数量: 量子模型通常需要较少的参数, 因为量子计算的复杂性较低。尽管如此, 量子神经网络可能仍然需要在优化过程中使用大量的资源 (如经典优化器), 并且其收敛速度较慢。
- 模型精度: 在不同的分类任务中, 量子算法的表现可能会有所不同。经典算法 (如 CNN) 在大多数图像分类任务中已经非常成熟, 因此量子算法的主要目标是探索是否能够在某些特定场景中超越经典算法。
- 模型收敛速度: 量子算法的训练可能会受到量子硬件的限制, 尤其是在目前量子硬件还不够成熟的情况下。训练量子神经网络可能需要较长的时间, 而经典神经网络 (特别是 CNN) 在标准硬件上能够快速收敛。

## 0.4 四. 算法原理

### 4.1 量子神经网络 (QNN)

量子神经网络 (QNN) 的核心思路量子神经网络的目标是利用量子电路来对数据进行编码并进行分类。在这个过程中，量子计算提供了比经典计算更强大的数据处理能力，尤其是在处理大规模数据和多维数据时。

#### 1. 将经典数据编码为量子态。

经典数据必须通过一定的编码方法转换为量子态。这通常通过量子特征映射实现，即将经典数据（如数字图像）转化为量子比特 (qubit) 的状态。常见的编码方法包括振幅编码、基态编码等。

例如，对于 MNIST 数据集中的每张图像，可能会通过对图像的像素值进行归一化处理，然后将每个像素值映射为量子比特的幅度。

#### 2. 设计量子电路，利用参数化门（如旋转门）对量子态进行变换。

量子电路是 QNN 的核心组成部分。通过使用一系列量子门（如 Pauli-X、Hadamard、CNOT、ZZ 和 XX 等门）来处理量子数据。每个量子门的参数可以被优化，以实现最佳的分类结果。

在 QNN 中，量子门通常是参数化量子门 (PQC)，即门的参数可以在训练过程中更新。常用的量子门包括旋转门（例如  $R_z(\theta)$ ），它们控制量子比特的旋转。

#### 3. 通过测量量子态的期望值（如 Pauli-Z 算符）得到分类结果。

量子电路的输出通常通过测量量子态的期望值（如 Pauli-Z 算符）来得到分类结果。这个期望值可以被解释为量子态的分类概率。

量子神经网络通常采用变分量子电路 (VQC) 设计，这种设计将量子计算与经典优化相结合，通过优化电路的参数来最小化损失函数。

#### 4. 使用经典优化算法（如梯度下降）优化量子电路的参数。

训练过程与经典神经网络相似，使用经典优化算法（如梯度下降法）来更新量子电路中的参数。优化过程中，损失函数计算是通过量子电路的测量来完成的，而梯度计算则是通过量子反向传播 (quantum backpropagation) 或其他量子优化方法来实现。

## 4.2 构建与训练量子神经网络（QNN）

### 1. 构建量子神经网络

- (a) 将训练输入看作是二进制的字串，把其中每个比特转换成量子比特，例如 0 对应  $|0\rangle$ ，1 对应  $|1\rangle$ ；
- (b) 创建一个量子电路，输入端有  $N+1$  个量子比特，其中  $N$  个量子比特对应输入的字串长度，另有一个 read out 量子比特，用来输出分类结果；
- (c) 量子电路内部，用一组带参数的门连接起来，具体用什么门以及如何连接，似乎没有特别的要求，在文章里作者经过试验选用了带参数的  $XX$  和  $ZZ$  门。每个门带有一个控制参数（例如  $XX^a$ ）。如果量子电路有 32 个这样的门，那么就有 32 个参数需要学习；
- (d) 把量子电路作为神经网络的一层，输入训练集和标签，学习控制参数。

### 2. 训练量子神经网络

把量子电路看作神经网络的一层，那么输入训练集里的字串，从它的 read out 那里读出分类结果，和训练标签比较，产生 loss，然后反馈给神经网络进行梯度下降。经过训练，模型会学习到一组量子电路的参数，用这组参数控制量子电路，就能对未知的输入作出准确的分类预测。

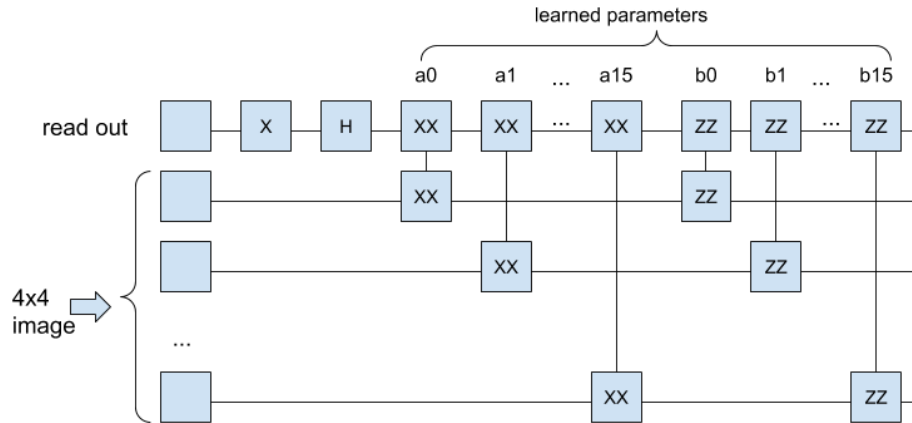


图 2: 量子神经网络示意图

## 0.5 五. 技术实现

### 5.1 数据加载与预处理

```
1 class DataPreprocessor:
2     @staticmethod
3     def load_and_preprocess_data():
4         """加载并预处理MNIST数据集"""
5         ...
6
7     @staticmethod
8     def filter_36(x, y):
9         """过滤出数字3和6的样本"""
10        ...
11
12    @staticmethod
13    def remove_contradicting(xs, ys):
14        """删除矛盾的样本"""
15        ...
```

#### 5.1.1 5.1.1 加载 MNIST 原始数据集

1. 加载 MNIST 数据集，并将像素值从整数区间  $[0, 255]$  归一化到  $[0.0, 1.0]$ 。

实现代码如下：

```
1 def load_and_preprocess_data():
2     """加载并预处理MNIST数据集"""
3     (x_train, y_train), (x_test, y_test) =
4         tf.keras.datasets.mnist.load_data()
5     x_train, x_test = x_train[..., np.newaxis]/255.0, x_test[...,
6         np.newaxis]/255.0
7     return x_train, y_train, x_test, y_test
```



### 5.1.2 5.1.2 过滤出数字 3 和 6 的样本

本实验为二分类任务，仅保留数据集中数字“3”和“6”的图片，同时将标签转换为布尔值：若为“3”则为 *True*，否则为 *False*。

```
1 def filter_36(x, y):
2     """过滤出数字3和6的样本"""
3     keep = (y == 3) | (y == 6)
4     x, y = x[keep], y[keep]
5     y = y == 3
6     return x, y
```

### 5.1.3 5.1.3 去除互相矛盾的样本

在训练集中，部分样本的图像可能被标注为不同的标签，导致标签冲突。为提高数据集的纯净度，需要移除这些样本。具体步骤为：

1. 遍历样本，将具有不同标签的重复图像过滤掉。
2. 保留唯一标注的图像及其对应的标签。

实现代码如下：

```
1 def remove_contradicting(xs, ys):
2     mapping = collections.defaultdict(set)
3     orig_x = {}
4     for x, y in zip(xs, ys):
5         orig_x[tuple(x.flatten())] = x
6         mapping[tuple(x.flatten())].add(y)
7
8     new_x, new_y = [], []
9     for flatten_x in mapping:
10         x = orig_x[flatten_x]
11         labels = mapping[flatten_x]
12         if len(labels) == 1:
13             new_x.append(x)
14             new_y.append(next(iter(labels)))
15
16     return np.array(new_x), np.array(new_y)
```

## 5.2 构建量子电路

```
1 class QuantumCircuitBuilder:
2     @staticmethod
3     def convert_to_circuit(image):
4         """将图像转换为量子电路"""
5         ...
6
7     class CircuitLayerBuilder:
8         ...
```

### 5.2.1 5.2.1 将图像转换为量子电路

1. 将图像的像素值二值化，设置阈值为 0.5，大于此值的像素设置为 1，否则为 0。
2. 将图像的数据通过 `np.ndarray.flatten(image)` 转换成一维数组 `values`。`values` 中每个元素对应图像中的一个像素点值（如二值化图像为 0 或 1）。
3. 根据二值化结果，构建  $4 \times 4$  的量子比特网格，并使用  $|0\rangle$  和  $|1\rangle$  分别表示二值化的像素。
4. 对每个像素，若其值为 1，则在其对应的量子比特上添加一个 X 门。
5. 返回构建的量子电路。

```
1 def convert_to_circuit(image):
2     """将图像转换为量子电路"""
3     values = np.ndarray.flatten(image)
4     qubits = cirq.GridQubit.rect(4, 4)
5     circuit = cirq.Circuit()
6     for i, value in enumerate(values):
7         if value:
8             circuit.append(cirq.X(qubits[i]))
9     return circuit
```

### 5.2.2 5.2.2 构建量子电路

通过 `add_layer` 方法，动态地向电路中添加不同类型的量子门，同时利用符号表示量子门的参数。

1. 输入: `data_qubits` 是数据量子比特的列表, `readout` 是一个单独的量子比特, 用于最终的测量操作。初始化时保存数据量子比特和读取量子比特信息。
2. 使用 `gate(qubit, self.readout)**symbol` 在数据量子比特上与读取量子比特之间添加一个量子门 (例如 `XX` 或 `ZZ` 门)

```
1 class CircuitLayerBuilder:
2     def __init__(self, data_qubits, readout):
3         self.data_qubits = data_qubits
4         self.readout = readout
5
6     def add_layer(self, circuit, gate, prefix):
7         for i, qubit in enumerate(self.data_qubits):
8             symbol = sympy.Symbol(prefix + '-' + str(i))
9             circuit.append(gate(qubit, self.readout)**symbol)
```

## 5.3 构建模型

```
1 class ModelBuilder:
2     @staticmethod
3     def create_quantum_model():
4         """创建量子神经网络模型"""
5         ...
6     @staticmethod
7     def create_classical_model():
8         """创建经典CNN模型"""
9         ...
10    @staticmethod
11    def create_fair_classical_model():
12        """创建公平的经典模型"""
13        ...
```

### 5.3.1 5.3.1 构建量子神经网络

1. `data_qubits = cirq.GridQubit.rect(4, 4)` 创建了一个 4x4 的量子比特网格，意味着模型中将使用 16 个量子比特来表示数据。
2. `readout = cirq.GridQubit(-1, -1)` 创建了一个额外的量子比特，用于测量操作。
3. 初始化电路: `circuit = cirq.Circuit()` 创建一个空的量子电路。  
应用 X 门（翻转）在读取量子比特上。  
应用 Hadamard 门（H 门）在读取量子比特上，使其处于叠加态。
4. 添加层: 创建 `CircuitLayerBuilder` 对象并使用它的 `add_layer` 方法添加两层量子门。
  - 第一层: 使用 `cirq.XX` 门（XX 门作用于两个量子比特）和符号前缀 `xx1` 添加门。
  - 第二层: 使用 `cirq.ZZ` 门（ZZ 门作用于两个量子比特）和符号前缀 `zz1` 添加门。
  - 最后，使用 `circuit.append(cirq.H(readout))` 将 Hadamard 门应用到读取量子比特，以便为测量做准备。

```
1 def create_quantum_model():
2     """创建量子神经网络模型"""
3     data_qubits = cirq.GridQubit.rect(4, 4)
4     readout = cirq.GridQubit(-1, -1)
5     circuit = cirq.Circuit()
6     circuit.append(cirq.X(readout))
7     circuit.append(cirq.H(readout))
8
9     builder = QuantumCircuitBuilder.CircuitLayerBuilder(data_qubits,
10                                                            readout)
11     builder.add_layer(circuit, cirq.XX, "xx1")
12     builder.add_layer(circuit, cirq.ZZ, "zz1")
13     circuit.append(cirq.H(readout))
14
15     return circuit, cirq.Z(readout)
```

### 5.3.2 5.3.2 构建经典 CNN 神经网络

定义了一个经典的卷积神经网络（CNN）模型，使用 TensorFlow 和 Keras 框架。创建一个包含卷积层、池化层、Dropout 层和全连接层的经典 CNN 模型。

1. 使用 `tf.keras.layers.Conv2D` 添加卷积层, `activation='relu'` 指定了该层使用 ReLU 激活函数
2. 使用 `tf.keras.layers.MaxPooling2D` 添加池化层。最大池化层（`MaxPooling2D`）用于空间降维，减小图像的大小（宽度和高度），同时保留最重要的信息。这里使用了一个 `2x2` 的池化窗口，即对每个 `2x2` 的区域取最大值，从而将图像的尺寸减小为原来的一半。
3. 使用 `tf.keras.layers.Dropout` 添加 Dropout 层。Dropout 层用于防止过拟合，随机将一部分神经元的输出置为 0。这里设置的 dropout 比例为 0.25，即随机将 25% 的神经元输出置为 0。
4. 使用 `tf.keras.layers.Flatten` 将多维输入展平为一维
5. 使用 `tf.keras.layers.Dense` 添加全连接层。最后一层是一个全连接层，只有一个神经元，通常用于回归问题或者二分类问题。输出层没有使用激活函数（默认是线性激活），表示该层输出的是一个实数，可能代表某个分类概率（如果用于二分类任务）。

```
1 @staticmethod
2 def create_classical_model():
3     """创建经典CNN模型"""
4     return tf.keras.Sequential([
5         tf.keras.layers.Conv2D(32, [3, 3], activation='relu',
6                                 input_shape=(28, 28, 1)),
7         tf.keras.layers.Conv2D(64, [3, 3], activation='relu'),
8         tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
9         tf.keras.layers.Dropout(0.25),
10        tf.keras.layers.Flatten(),
11        tf.keras.layers.Dense(128, activation='relu'),
12        tf.keras.layers.Dropout(0.5),
13        tf.keras.layers.Dense(1),
14    ])
```

### 5.3.3 5.3.3 构建公平的经典模型

量子神经网络（QNN）由于只有 32 个参数，和传统的神经网络相比，可能在参数量和表达能力上存在差距。因此可以构建一个包含 37 个参数的经典神经网络。

1. 使用 `tf.keras.layers.Flatten` 将输入的 4x4 的量子比特网格展平为一维向量。
2. 添加一个包含 2 个神经元的隐藏层，使用 ReLU 激活函数。
3. 添加一个包含 1 个神经元的输出层，没有激活函数。

```
1 @staticmethod
2 def create_fair_classical_model():
3     """创建公平的经典模型"""
4     return tf.keras.Sequential([
5         tf.keras.layers.Flatten(input_shape=(4, 4, 1)),
6         tf.keras.layers.Dense(2, activation='relu'),
7         tf.keras.layers.Dense(1),
8     ])
```

## 5.4 训练与评估

```
1 class Trainer:
2     def __init__(self):
3         self.preprocessor = DataPreprocessor()
4         self.quantum_builder = QuantumCircuitBuilder()
5         self.model_builder = ModelBuilder()
6
7     def train_all_models(self):
8         # 加载和预处理数据
9         ...
10        # 准备量子模型数据
11        ...
12        # 转换为量子电路
13        ...
14        # 训练量子模型
15        ...
16        # 训练经典CNN模型
17        ...
18        # 训练公平经典模型
19        ...
20        # 修改返回值，加入训练历史
21
22    def plot_results(results):
23        """绘制比较结果"""
24        ...
```

### 5.4.1 5.4.1 训练模型

#### 1. 加载和预处理数据

加载 MNIST 数据集，并将像素值从整数区间  $[0, 255]$  归一化到  $[0.0, 1.0]$ 。

二分类任务，仅保留数据集中数字“3”和“6”的图片，同时将标签转换为布尔值：若为“3”则为 *True*，否则为 *False*。

```

1  x_train, y_train, x_test, y_test =
    self.preprocessor.load_and_preprocess_data()
2  x_train, y_train = self.preprocessor.filter_36(x_train,
    y_train)
3  x_test, y_test = self.preprocessor.filter_36(x_test, y_test)

```

## 2. 准备量子模型数据

由于量子比特数量的限制，将训练集和测试集的图像数据从 28x28 的尺寸调整为 4x4 的尺寸，并移除矛盾的样本。

```

1  x_train_small = tf.image.resize(x_train, (4, 4)).numpy()
2  x_test_small = tf.image.resize(x_test, (4, 4)).numpy()
3  x_train_nocon, y_train_nocon =
    self.preprocessor.remove_contradicting(x_train_small,
    y_train)

```

## 3. 转换为量子电路

先将图像的像素值二值化，设置阈值为 0.5，大于此值的像素设置为 1，否则为 0。

将二值化后的图像数据转换为量子电路。

```

1  # 二值化
2  x_train_bin = np.array(x_train_nocon > THRESHOLD,
    dtype=np.float32)
3  x_test_bin = np.array(x_test_small > THRESHOLD,
    dtype=np.float32)
4
5  # 转换为量子电路
6  x_train_circ =
    [self.quantum_builder.convert_to_circuit(x) for x in
    x_train_bin]
7  x_test_circ =
    [self.quantum_builder.convert_to_circuit(x) for x in
    x_test_bin]
8  x_train_tfcirc = tfq.convert_to_tensor(x_train_circ)

```



```
9      x_test_tfcirc = tfq.convert_to_tensor(x_test_circ)
```

#### 4. 训练量子模型

- (a) 构建量子神经网络（QNN）模型通过 `tf.keras.Sequential` 构建了一个 Keras 模型, 包含 input layer 和 PQC layer。
- (b) 数据预处理, 将标签转换为 hinge 损失函数所需的格式。经过转换后变成了 -1 和 1
- (c) 编译量子模型, 使用 Hinge 损失函数和 Adam 优化器, 并设置 metrics 为 accuracy。
- (d) 训练量子模型, 使用 `fit` 方法训练模型, 并设置 `batch_size` 和 `epochs`。  
`x_train_tfcirc` 和 `y_train_hinge` 是训练数据和标签, `validation_data=(x_test_tfcirc, y_test_hinge)` 用于在每个训练周期后验证模型的性能, 验证数据通常用于评估模型的泛化能力。
- (e) 评估量子模型, `qnn_model.evaluate(x_test_tfcirc, y_test)` 对模型在测试集上的性能进行评估, 输出模型在测试集上的损失值和准确率。

```
1  # 训练量子模型
2  circuit, readout = self.model_builder.create_quantum_model()
3  qnn_model = tf.keras.Sequential([
4      tf.keras.layers.Input(shape=(), dtype=tf.string),
5      tfq.layers.PQC(circuit, readout),
6  ])
7
8  y_train_hinge = 2.0 * y_train_nocon - 1.0
9  y_test_hinge = 2.0 * y_test - 1.0
10
11  qnn_model.compile(
12      loss=tf.keras.losses.Hinge(),
13      optimizer=tf.keras.optimizers.Adam(),
14      metrics=['accuracy'])
15
16  qnn_history = qnn_model.fit(
17      x_train_tfcirc, y_train_hinge,
18      batch_size=BATCH_SIZE,
19      epochs=EPOCHS,
20      validation_data=(x_test_tfcirc, y_test_hinge))
```

21

```
22     qnn_results = qnn_model.evaluate(x_test_tfcirc, y_test)
```

## 5. 训练经典 CNN 模型

- (a) 创建经典 CNN 模型: 调用 `create_classical_model` 方法来构建一个经典卷积神经网络 (CNN) 模型。返回一个 Keras Sequential 模型, 通常包含几个卷积层、池化层和全连接层
- (b) 编译 CNN 模型: 使用 Binary Crossentropy (二元交叉熵) 作为损失函数, 使用 Adam 优化器, 使用准确率 (accuracy) 作为评估指标
- (c) 训练 CNN 模型: 在每轮训练后使用验证集 `x_test` 和 `y_test` 评估模型的性能
- (d) 评估 CNN 模型: 训练完成后, 使用测试集 `x_test` 和 `y_test` 来评估模型的最终性能。这个方法返回模型在测试集上的损失值和准确率 (根据前面定义的评估指标)。

```
1     # 训练经典CNN模型
2     cnn_model = self.model_builder.create_classical_model()
3     cnn_model.compile(
4         loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
5         optimizer=tf.keras.optimizers.Adam(),
6         metrics=['accuracy'])
7
8     cnn_history = cnn_model.fit(
9         x_train, y_train,
10        batch_size=128,
11        epochs=20,
12        validation_data=(x_test, y_test))
13
14     cnn_results = cnn_model.evaluate(x_test, y_test)
```

## 6. 训练公平经典模型与训练 CNN 经典模型相同。

```
1  # 训练公平经典模型
2  fair_model =
3      self.model_builder.create_fair_classical_model()
4  fair_model.compile(
5      loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
6      optimizer=tf.keras.optimizers.Adam(),
7      metrics=['accuracy'])
8
9  fair_history = fair_model.fit(
10     x_train_bin, y_train_nocon,
11     batch_size=128,
12     epochs=20,
13     validation_data=(x_test_bin, y_test))
14
15 fair_results = fair_model.evaluate(x_test_bin, y_test)
```

### 5.4.2 5.4.2 可视化结果

plot\_results 函数，主要用于绘制和保存不同模型（量子神经网络、经典神经网络、经典公平模型）的比较结果，包括它们的准确率对比和训练过程中的准确率与损失曲线。

#### 1. 绘制准确率柱状图:sns.barplot() 使用 seaborn 库绘制柱状图：

- x 参数是一个列表，表示三个模型的名字：量子模型（"Quantum"），经典全模型（"Classical, full"），经典公平模型（"Classical, fair"）。
- y 参数是一个包含三个模型准确率的列表，准确率数据来自 results['accuracy'] 字典，分别对应'qnn'（量子神经网络）、'cnn'（经典神经网络）和'fair'（经典公平模型）的准确率。

#### 2. 绘制每个模型的训练历史: 遍历每个模型: for model in model\_names: 通过循环，分别为每个模型绘制训练历史图。对每个模型，生成两个子图（准确率曲线和损失曲线）

```
1 def plot_results(results):
```

```

2     """绘制比较结果"""
3     # 绘制准确率柱状图
4     plt.figure(figsize=(10, 6))
5     sns.barplot(
6         x=["Quantum", "Classical, full", "Classical, fair"],
7         y=[results['accuracy']['qnn'],
8             results['accuracy']['cnn'],
9             results['accuracy']['fair']])
10    plt.title('Model Accuracy Comparison')
11    plt.ylabel('Accuracy')
12    plt.savefig('accuracy_comparison.png')
13    plt.close()
14
15    # 为每个模型分别绘制训练历史
16    model_names = ['qnn', 'cnn', 'fair']
17    display_names = {'qnn': 'Quantum Neural Network',
18                     'cnn': 'Classical CNN',
19                     'fair': 'Fair Classical Model'}
20
21    for model in model_names:
22        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 12))
23
24        # 准确率曲线
25        ax1.plot(results['history'][model]['accuracy'], 'b-',
26                label='Accuracy')
27        ax1.set_title(f'{display_names[model]} - Accuracy over
28                      Epochs')
29        ax1.set_xlabel('Epoch')
30        ax1.set_ylabel('Accuracy')
31        ax1.legend()
32        ax1.grid(True)
33
34        # 损失曲线
35        ax2.plot(results['history'][model]['loss'], 'r-',
36                label='Loss')

```

```
34     ax2.set_title(f'{display_names[model]} - Loss over Epochs')
35     ax2.set_xlabel('Epoch')
36     ax2.set_ylabel('Loss')
37     ax2.legend()
38     ax2.grid(True)
39
40     plt.tight_layout()
41     plt.savefig(f'training_history_{model}.png')
42     plt.close()
```

## 0.6 六. 实验测试

```
1 def main():
2     trainer = Trainer()
3     results = trainer.train_all_models()
4     plot_results(results)
5
6 if __name__ == "__main__":
7     main()
```

运行后分别得到

- 不同模型（量子神经网络、经典神经网络、经典公平模型）的准确率柱状图

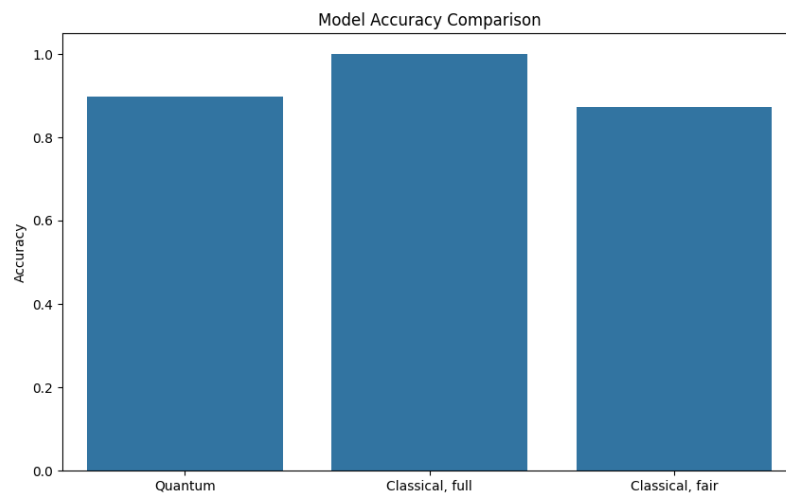


图 3: 准确率柱状图

由上图可以看出，对于训练参数足够大的经典 CNN 神经网络，对于 MNIST 二分类问题，可以轻松实现 100% 准确率识别，其次是 quantum 量子神经网络，性能略微由于同等参数规模的经典 CNN fair 模型。

- 每个模型训练历史图

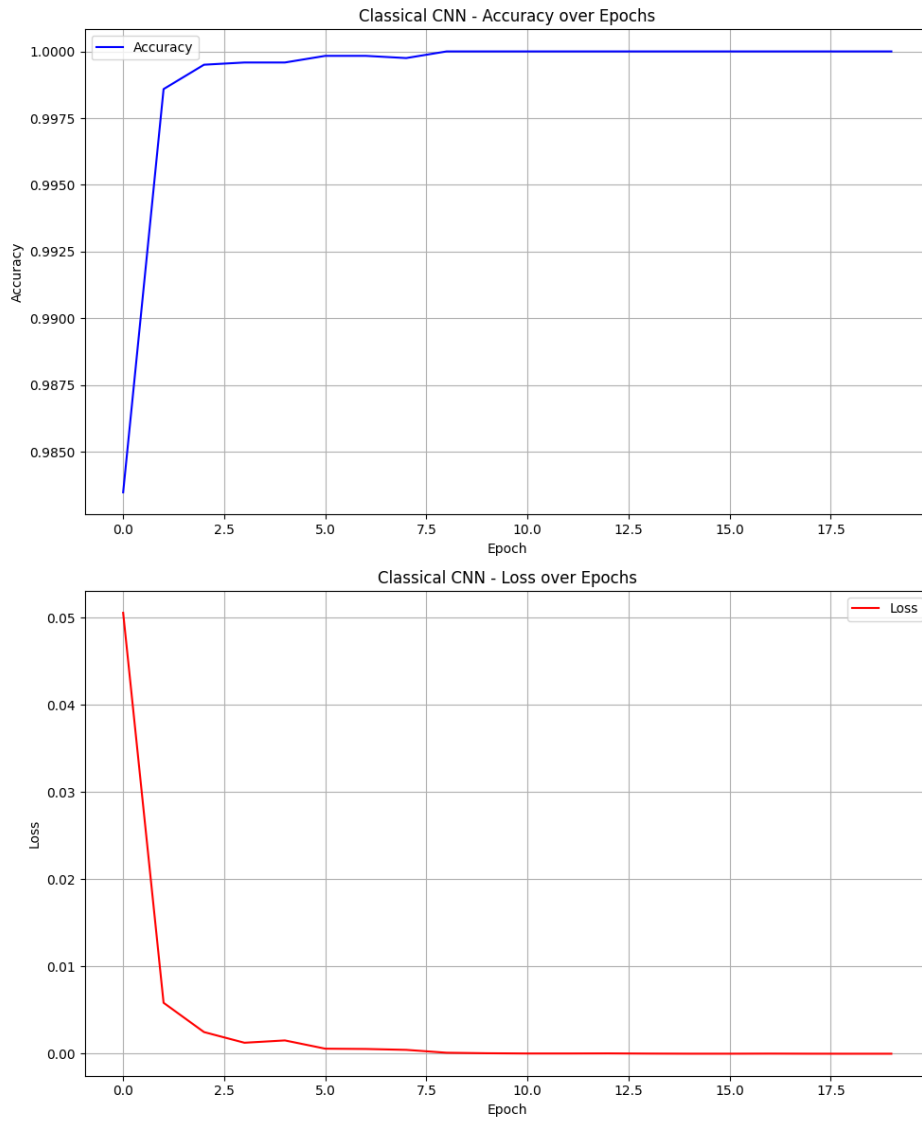


图 4: 经典 CNN 模型训练历史图



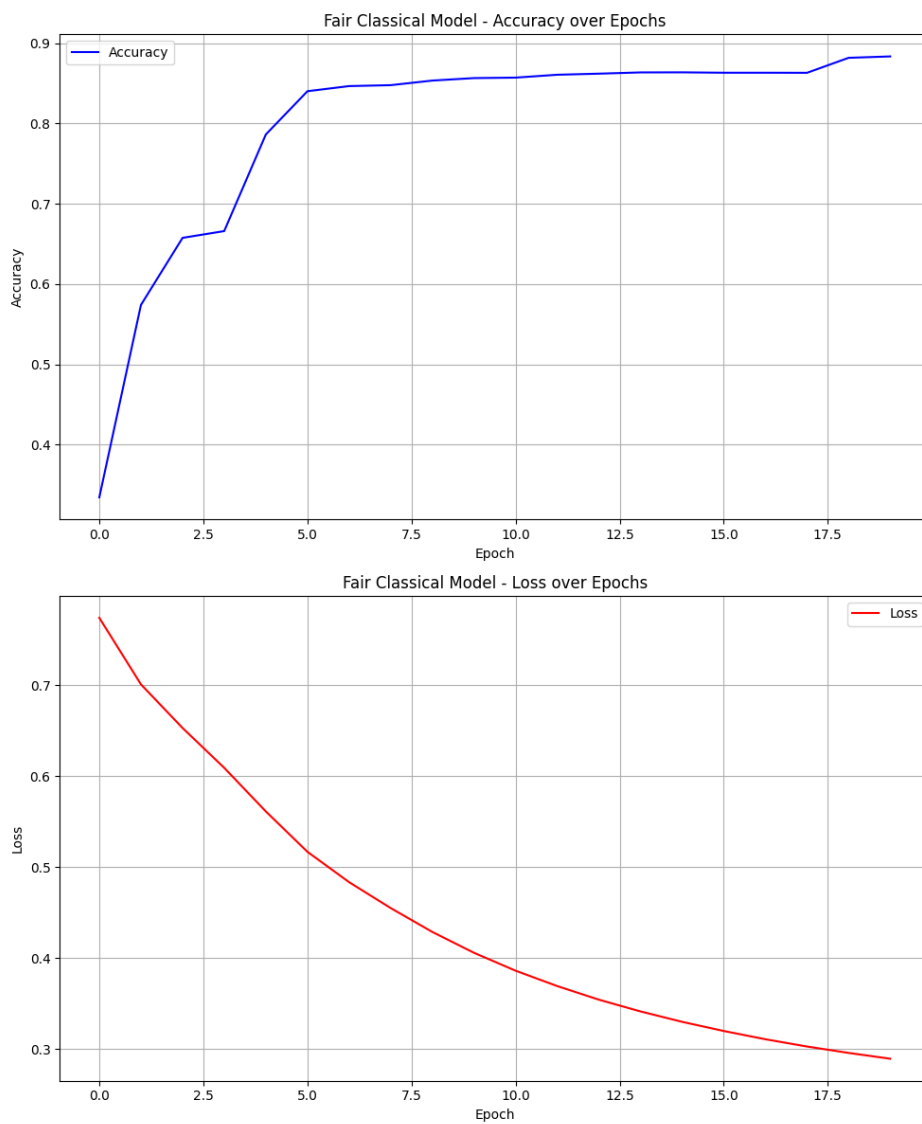


图 5: 经典 CNN fair 模型训练历史图

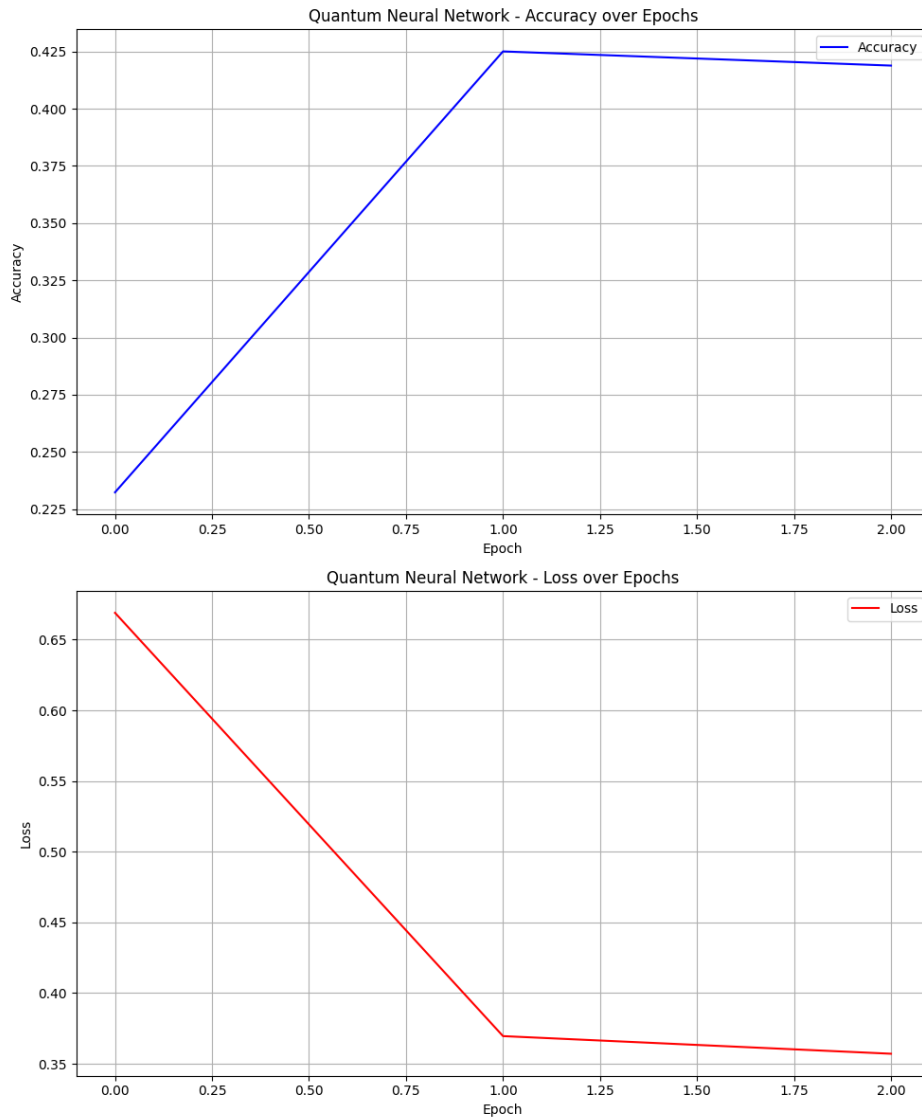


图 6: QNN 模型训练历史图

- 模型参数量：量子模型的参数量通常较少，因为量子电路的复杂度较低。相比之下，经典 CNN 模型的参数量较大，尤其是在卷积层和全连接层中。
- 模型精度：在二分类任务中，经典 CNN 模型表现最好，能够轻松实现 100
- 模型收敛速度：量子模型的训练速度较慢，主要原因是量子计算的效率受限，尤其是在模拟量子电路时，计算复杂度较高。相比之下，经典 CNN 模型的训练速度较快，能够在较少的 epoch 内收敛。

## 0.7 七. 仿照经典神经网络的核心组件，设计完成对应的量子模块

### 7.1 混合模型 HybridModel

定义了一个混合模型（HybridModel），继承自 BaseModel。这个模型的目的是根据不同的配置使用量子或经典的网络架构（backbone 和分类器）

1. 选择骨干网络类型（Backbone）：self.backbone\_type 从超参数字典 self.hp 中获取”backbone\_type”键的值。如果没有提供该键，则使用默认值’Q’（即默认使用量子骨干网络）。然后根据 self.backbone\_type 的值，决定是使用量子骨干网络（QuantumBackbone）还是经典骨干网络（ClassicalBackbone）：
  - 如果 backbone\_type == ’Q’，则实例化 QuantumBackbone。
  - 如果 backbone\_type != ’Q’（即任何非’Q’的值），则实例化 ClassicalBackbone。
2. 选择分类器类型（Classifier）：根据 self.classifier\_type 的值，决定是使用量子分类器（QuantumClassifier）还是经典分类器（ClassicalClassifier）：
  - 如果 classifier\_type == ’Q’，则实例化 QuantumClassifier。
  - 如果 classifier\_type != ’Q’，则实例化 ClassicalClassifier。

```
1 class HybridModel(BaseModel):
2     def __init__(self, input_shape, output_shape, **hp):
3         super().__init__(**hp)
4
5         self.backbone_type = self.hp.get("backbone_type", "Q")
6         self.backbone = QuantumBackbone(input_shape, output_shape,
7                                         **self.hp)
8         if self.backbone_type == 'Q' else
9             ClassicalBackbone(input_shape, output_shape, **self.hp)
10        backbone_output_shape = self.backbone.get_output_shape()
11
12        self.classifier_type = self.hp.get("classifier_type", "Q")
```

```

11         self.classifier = QuantumClassifier(backbone_output_shape,
        output_shape, **self.hp) \
12         if self.classifier_type == 'Q' else
        ClassicalClassifier(backbone_output_shape,
        output_shape, **self.hp)
13
14     def forward(self, x):
15         # print(x.shape, x.device)
16         features = self.backbone(x)
17         # print(features.shape, features.device)
18         y_hat = self.classifier(features)
19         # print(y_hat.shape, y_hat.device)
20         # print([param.device for param in self.parameters()])
21         return y_hat

```

## 7.2 QuantumConvolutionLayer

量子卷积层（QuantumConvolutionLayer），是一个集成了量子计算和经典卷积操作的神经网络层。它使用了量子计算框架 PennyLane（qml）与 PyTorch 结合来实现一个量子卷积操作。

```

1 class QuantumConvolutionLayer(torch.nn.Module):
2     def __init__(self, **hp):
3         ...
4
5     def circuit(self, phi=None, layer_params=None):
6         ...
7
8     def forward(self, x):
9         ...
10
11     def convolve(self, x):
12         ...
13
14     def q_filter(self, phi):
15         ...

```

## 1. 初始化方法:

- `dev = qml.device("default.qubit", wires=self.nb_qubits)`: 创建了一个量子设备, 使用了 PennyLane 提供的 `default.qubit` 后端, 并指定了量子比特的数量 (`wires=self.nb_qubits`), 即该设备可以同时操作 2 个量子比特。
- `self.qnode = qml.QNode(get_circuit, dev, interface='torch')`: 创建一个 PennyLane QNode, 它是量子电路与 PyTorch 的接口, `interface='torch'` 表示可以与 PyTorch 张量互操作。`self.qnode` 是一个量子节点, 可以通过它执行量子计算并返回经典结果。

```
1 def __init__(self, **hp):
2     super().__init__()
3     self.kernel_size = hp.get("kernel_size", (2, 2))
4     self.nb_qubits = hp.get("nb_qubits", 2)
5     dev = qml.device("default.qubit", wires=self.nb_qubits)
6
7     # Random circuit parameters
8     np.random.seed(0)
9     rand_params = np.random.uniform(high=2 * np.pi,
10                                     size=self.kernel_size)
11
12     get_circuit = lambda phi, layer_param: self.circuit(phi,
13                                                         layer_param)
14
15     self.qnode = qml.QNode(get_circuit, dev, interface='torch')
16
17     self.output_shape = None
```

## 2. 量子电路 circuit:

- 对于每个量子比特 `j`, 使用旋转门 (RY) 对量子比特应用一个旋转, 旋转角度为 `* phi[j]`
- `RandomLayers` 是一个假定的量子层操作 (可能是一个随机生成的量子电路), 通过 `layer_params` 参数控制层的具体细节。它对量子比特应用一系列随机的量子门。

- `qml.expval(qml.PauliZ(j))` for `j` in `range(self.nb_qubits)`: 测量每个量子比特在 Pauli-Z 基底下的期望值，并返回结果。该操作测量量子比特的状态，并将其转换为经典信息（例如 0 或 1）。

```

1 def circuit(self, phi=None, layer_params=None):
2     for j in range(self.nb_qubits):
3         qml.RY(np.pi * phi[j], wires=j)
4
5     RandomLayers(layer_params,
6                   wires=list(range(self.nb_qubits)))
7
8     return [qml.expval(qml.PauliZ(j)) for j in
9             range(self.nb_qubits)]

```

### 3. 量子卷积层 forward:

- 创建一个全零的张量 `out`，作为输出结果
- 对于输入数据中的每个样本和每个通道，`self.convolve(x[i, j])` 调用 `convolve` 方法来执行量子卷积。

```

1 def forward(self, x):
2     if self.output_shape is None:
3         self.output_shape = (
4             x.shape[0],
5             x.shape[1] * self.nb_qubits,
6             x.shape[2] // self.kernel_size[0],
7             x.shape[3] // self.kernel_size[1]
8         )
9     output_shape = (x.shape[0], x.shape[1]*self.nb_qubits,
10                     x.shape[2] // self.kernel_size[0], x.shape[3] //
11                     self.kernel_size[1])
12     out = torch.zeros(output_shape).to(x.device)
13     for i in range(x.shape[0]):
14         for j in range(x.shape[1]):
15             out[i] = self.convolve(x[i, j])
16     return out

```

#### 4. 量子卷积层 convolve:

- 遍历输入张量  $x$  的空间维度，进行卷积操作。
- 提取输入张量的子区域，并将其展平后传递给量子滤波器（`q_filter`）进行处理。
- 将量子滤波器返回的结果存储在输出张量 `out` 中。

```
1 def convolve(self, x):
2     out = torch.zeros((self.nb_qubits, x.shape[0] //
3                        self.kernel_size[0], x.shape[1] //
4                        self.kernel_size[1])).to(x.device)
5     for j in range(0, x.shape[0] - 1):
6         for k in range(0, x.shape[1] - 1):
7             q_results = self.q_filter(x[j:j +
8                                       self.kernel_size[0], k:k +
9                                       self.kernel_size[1]].flatten())
10            for c in range(self.nb_qubits):
11                out[c, j // self.kernel_size[0], k //
12                    self.kernel_size[1]] = q_results[c]
13    return out
```

#### 5. 量子滤波器 `q_filter`:

- `self.qnode(phi, self.weights)`: 调用量子电路，执行量子计算，并返回结果。

```
1 def q_filter(self, phi):
2     return self.qnode(phi, self.weights)
```

### 7.3 QuantumPseudoLinearLayer

量子伪线性层（`QuantumPseudoLinearLayer`），是一个基于量子计算的神经网络层。它使用了量子计算框架 PennyLane（`qml`）与 PyTorch 结合来实现一个量子线性层。

- `qml.templates.AngleEmbedding(inputs, wires=range(self.nb_qubits))`: 将输入 `inputs` 映射到量子比特的角度。`AngleEmbedding` 是一种量子电路模板，用于将

经典数据（例如张量或数组）编码到量子比特中。输入 `inputs` 被转化为每个量子比特的旋转角度。

- `qml.templates.BasicEntanglerLayers(weights, wires=range(self.nb_qubits))`: 对量子比特应用 `BasicEntanglerLayers`，这是一个常用的量子电路模板，用于在量子比特之间创建纠缠。`weights` 参数控制该层中的量子门，`BasicEntanglerLayers` 用于在量子比特间实现更复杂的量子操作。
- `return [qml.expval(qml.PauliZ(wires=i)) for i in range(self.nb_qubits)]`: 对每个量子比特 `i`，计算其在 Pauli-Z 基底下的期望值。`qml.expval(qml.PauliZ(wires=i))` 是量子测量操作，返回每个量子比特的测量值（即 0 或 1）。

```
1 class QuantumPseudoLinearLayer(torch.nn.Module):
2     def __init__(self, **hp):
3         super().__init__()
4         self.nb_qubits = hp.get("nb_qubits", 2)
5         self.n_layers = hp.get("n_layers", 6)
6         dev = qml.device("default.qubit", wires=self.nb_qubits)
7         get_circuit = lambda inputs, weights: self.circuit(inputs,
8             weights)
9         self.qnode = qml.QNode(get_circuit, dev, interface='torch')
10        weight_shapes = {"weights": (self.n_layers, self.nb_qubits)}
11        self.seq = qml.qnn.TorchLayer(self.qnode, weight_shapes)
12
13    def circuit(self, inputs, weights):
14        qml.templates.AngleEmbedding(inputs,
15            wires=range(self.nb_qubits))
16        qml.templates.BasicEntanglerLayers(weights,
17            wires=range(self.nb_qubits))
18        return [qml.expval(qml.PauliZ(wires=i)) for i in
19            range(self.nb_qubits)]
20
21    def forward(self, x):
22        return self.seq(x)
```



## 7.4 QuantumClassifier

这个量子分类器（QuantumClassifier）是一个混合模型，结合了经典神经网络层和量子计算层。

经典部分：输入通过展平层和线性层进行处理。然后，输入通过多个量子层（QuantumPseudoLinearLayer）进行量子计算处理。

量子部分：每个量子层（QuantumPseudoLinearLayer）对数据进行量子变换，并返回量子计算的输出。

输出部分：最终的量子计算结果会通过一个经典的全连接层（seq\_1）进行进一步映射，并通过 Softmax 函数输出分类概率。

### 1. 初始化方法：

- 构建网络层 seq\_0, seq\_0 是一个包含经典全连接层和多个量子层的序列。  
当输入数据通过 seq\_0 时，数据先被展平，然后经过线性层映射到量子比特空间，接着通过多个量子层进行处理。
- 构建输出部分网络 seq\_1, 将来自 seq\_0 层的输出映射到最终的输出空间。

### 2. get\_output\_shape\_seq\_0: 该方法用于计算 seq\_0 层的输出形状

### 3. forward: self.seq\_0(x): 首先将输入数据 x 通过 seq\_0 网络层，这一层包括经典层和量子层，经过量子计算和经典映射后得到结果。self.seq\_1(...): 接着将 seq\_0 的输出传递给 seq\_1 层，进行最终的线性映射和分类操作。输出的结果是通过 Softmax 转换后的概率分布。

```

1 class QuantumClassifier(BaseModel):
2     def __init__(self, input_shape, output_shape, **hp):
3         super().__init__(**hp)
4         self.input_shape = input_shape
5         self.output_shape = output_shape
6         self.nb_qubits = hp.get("nb_qubits", 2)
7         self.seq_0 = torch.nn.Sequential(
8             torch.nn.Flatten(),
9             torch.nn.Linear(np.prod(self.input_shape),
10                             self.nb_qubits),
11             *[QuantumPseudoLinearLayer(nb_qubits=self.nb_qubits, **hp)
12               for _ in range(hp.get("nb_q_layer", 2))],
13         )
14         self.seq_1 = torch.nn.Sequential(
15             torch.nn.Flatten(),
16             torch.nn.Linear(np.prod(self.get_output_shape_seq_0()),
17                             self.output_shape),
18             torch.nn.Softmax(),
19         )
20
21     def get_output_shape_seq_0(self):
22         ones = torch.Tensor(np.ones((1, *self.input_shape))).float()
23         return self.seq_0(ones).shape
24
25     def forward(self, x):
26         return self.seq_1(self.seq_0(x))

```

## 7.5 QuantumBackbone

### 1. 初始化方法:

- 初始化超参数
- 构建量子卷积层序列 (QuantumConvolutionLayer)。每个量子卷积层对输入数据进行量子计算处理。

### 2. get\_output\_shape: 该方法用于计算量子卷积层的输出形状

### 3. forward: 该方法用于执行量子卷积层的正向传播, 将输入数据通过量子卷积层序列进行量子计算处理。

```
1 class QuantumBackbone(torch.nn.Module):
2     def __init__(self, input_shape, output_shape, **hp):
3         super().__init__()
4         self.hp = hp
5         self.input_shape = input_shape
6         self.seq = torch.nn.Sequential(
7             *[QuantumConvolutionLayer(kernel_size=self.hp.get("kernel_size",
8                 (2, 2)),
9
10                                nb_qubits=self.hp.get("nb_qubits",
11                                    2))
12             for _ in range(self.hp.get("nb_q_conv_layer", 1))]
13         )
14
15     def forward(self, x):
16         return self.seq(x)
17
18     def get_output_shape(self):
19         ones = torch.tensor(np.ones((1, *self.input_shape))).float()
20         return self(ones).shape
```

## 7.6 测试

`run_quantum_test`: 该方法用于执行量子模型的测试。`run_classification_test`: 该方法用于执行经典模型的测试。

### 1. 超参数优化:

- 使用 `GPOParamGen` 类生成超参数的优化过程，最多进行 30 次迭代。
- 调用 `optimize_parameters` 函数，根据给定的训练数据集和超参数生成函数优化超参数，优化的目标是选择最适合的超参数 `hp`。

### 2. 测试混合模型:

- 创建 `HybridModel`，该模型根据 `backbone_type` 和 `classifier_type` 配置，并传入优化后的超参数 `hp`。
- 调用 `fit` 方法，训练混合模型，并记录训练历史。
- 测试混合模型在测试集上的表现，得到测试分数 `test_score`。

```
1 def run_quantum_test(num_classes, combinations):
2     print(f"\n{'='*20} 测试{num_classes}分类 {'='*20}")
3     # 创建对应类别数的数据集
4     mnist_dataset = MNISTDataset(num_classes=num_classes)
5     print(f"\n数据集信息:")
6     train_data, train_labels = mnist_dataset.getTrainData()
7     test_data, test_labels = mnist_dataset.getTestData()
8     print(f"训练集大小: {len(train_data)}")
9     print(f"测试集大小: {len(test_data)}")
10
11     # 超参数优化
12     bounds_params = {
13         "nb_hidden_neurons": [1, 30, 1]
14     }
15
16     gpo = GPOParamGen(bounds_params, max_itr=30)
17     hp = optimize_parameters(
18         lambda **kwargs: ClassicalModel(output_shape=num_classes,
19                                         **kwargs),
```

```

19         *mnist_dataset.getTrainData(),
20         gpo,
21         fit_kwargs={"epochs": 20}
22     )
23     print(f"\n优化后的超参数: {hp}")
24     # gpo.show_expectation()
25
26     # 测试混合模型
27     results = []
28     print("\n测试混合模型:")
29     for backbone_type, classifier_type in combinations:
30         model_name = f"Hybrid_{backbone_type}{classifier_type}"
31         print(f"\n测试 {model_name}")
32
33         hybrid_model = HybridModel(
34             input_shape=(1, 8, 8),
35             output_shape=num_classes,
36             backbone_type=backbone_type,
37             classifier_type=classifier_type,
38             **hp
39         )
40
41         history_h = hybrid_model.fit(
42             *mnist_dataset.getTrainData(),
43             *mnist_dataset.getValidationData(),
44             batch_size=32,
45             verbose=True
46         )
47
48         test_score = hybrid_model.score(*mnist_dataset.getTestData())
49         print(f"{model_name} 测试分数: {test_score:.4f}")
50
51     # 保存训练历史
52     # 在result.csv文件中添加一行
53     with open('result.csv', 'a') as f:

```

```

54         f.write(f"{model_name}_{num_classes}_class\n")
55     hybrid_model.show_history(history_h,
56                               name=f'{model_name}_{num_classes}_class')
57     # 在result.csv文件中添加一行
58     with open('result.csv', 'a') as f:
59         f.write(f"{model_name} test score: {test_score:.4f}\n")
60     # 收集结果
61     results.append({
62         'model': model_name,
63         'num_classes': num_classes,
64         'test_score': test_score
65     })
66
67     return results
68
69 if __name__ == '__main__':
70     # 定义要测试的模型组合
71     combinations = [
72         ('C', 'Q'), ('Q', 'Q')
73     ]
74
75     # 存储所有结果
76     all_results = []
77
78     # 测试不同的分类数量
79     for num_classes in [2, 4, 10]:
80         results = run_classification_test(num_classes, combinations)
81         all_results.extend(results)
82         results = run_quantum_test(num_classes, combinations)
83         all_results.extend(results)
84
85     # 创建结果DataFrame并显示
86     results_df = pd.DataFrame(all_results)
87     print("\n所有测试结果汇总:")
88     print(results_df.to_string(index=False))

```

```

88
89     # 保存结果到CSV文件
90     results_df.to_csv('classification_results.csv', index=False)
91     print("\n结果已保存到 classification_results.csv")

```

## 7.7 结果分析

设置四种混合模型，分别测试二分类、四分类、十分类。四种混合模型分别是：

- ClasscialBackbone+ClasscialClassifier
- ClasscialBackbone+QuantumClassifier
- QuantumBackbone+ClasscialClassifier
- QuantumBackbone+QuantumClassifier

```

1  if __name__ == '__main__':
2
3      combinations = [
4          ('C', 'C'), ('C', 'Q'), ('Q', 'C'), ('Q', 'Q')
5      ]
6
7      # 存储所有结果
8      all_results = []
9
10     # 测试不同的分类数量
11     for num_classes in [2, 4, 10]:
12         results = run_classification_test(num_classes, combinations)
13         all_results.extend(results)
14         results = run_quantum_test(num_classes, combinations)
15         all_results.extend(results)

```

### 7.7.1 7.7.1 二分类测试

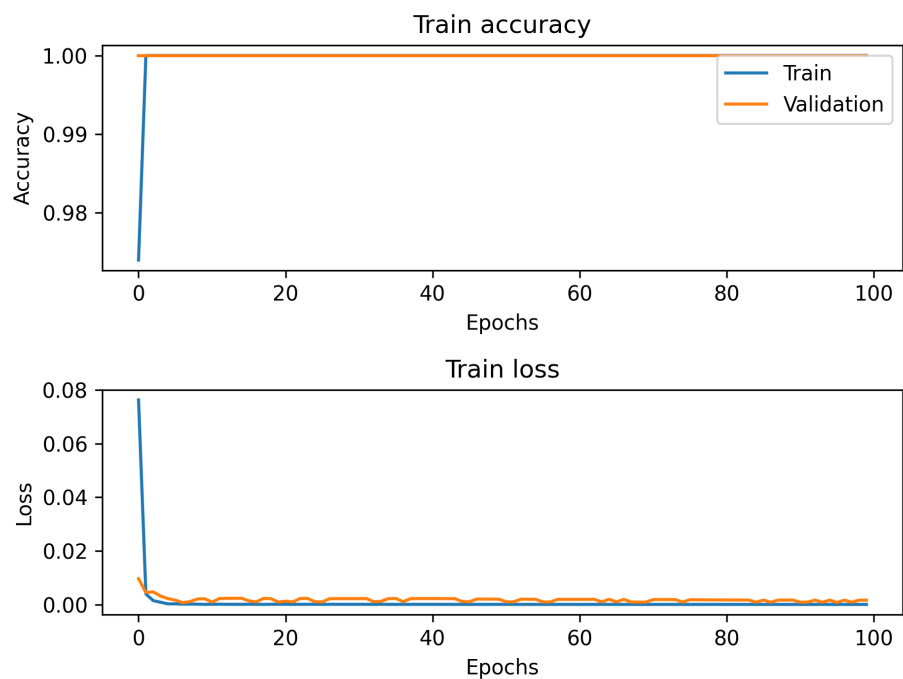


图 7: 经典神经网络模型测试二分类

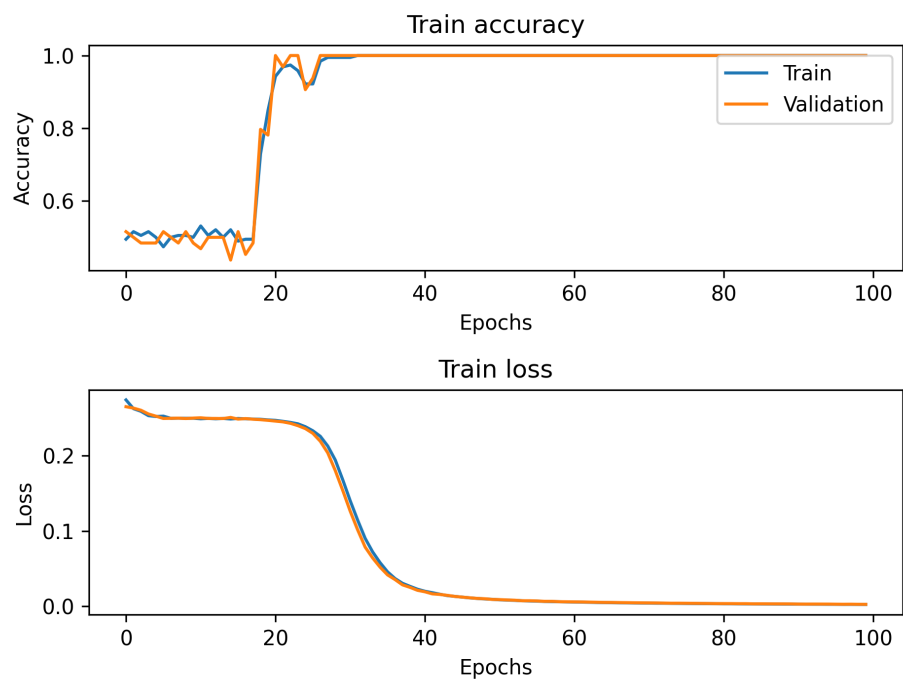


图 8: 混合模型 CQ 测试二分类



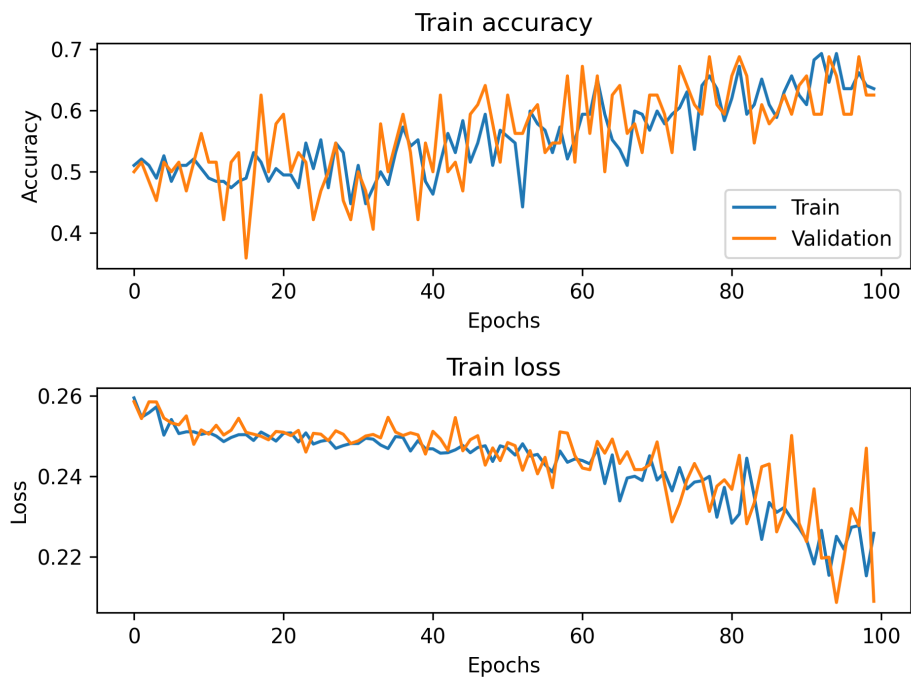


图 9: 混合模型 QC 测试二分类

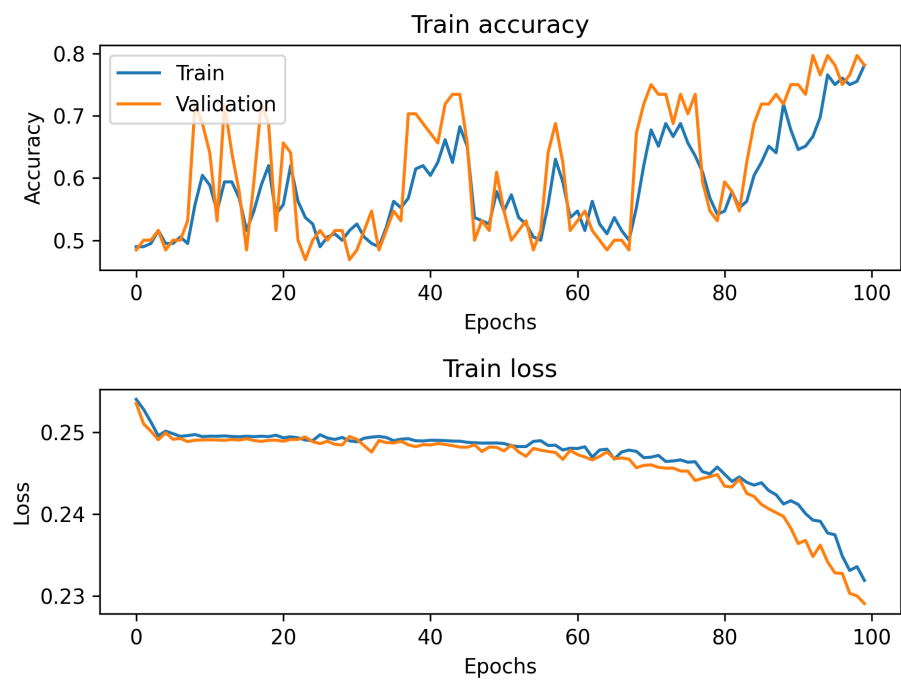


图 10: 混合模型 QQ 测试二分类

在二分类问题中：

- Classical test score: 1.0000
- Hybrid\_CQ test score: 1.0000
- Hybrid\_QC test score: 0.6250
- Hybrid\_QQ test score: 0.7361

在混合模型 CQ 中，Train accuracy 和 Test accuracy 都达到了 100%，说明混合模型在训练和测试集上的表现都较好。

但在混合模型 QC 和 QQ 中，虽然 Train loss 曲线始终在下降，但是 Train accuracy 和 Test accuracy 曲线在训练过程中出现了剧烈波动，说明混合模型在训练过程中易受噪声影响。

### 7.7.2 7.7.2 四分类测试

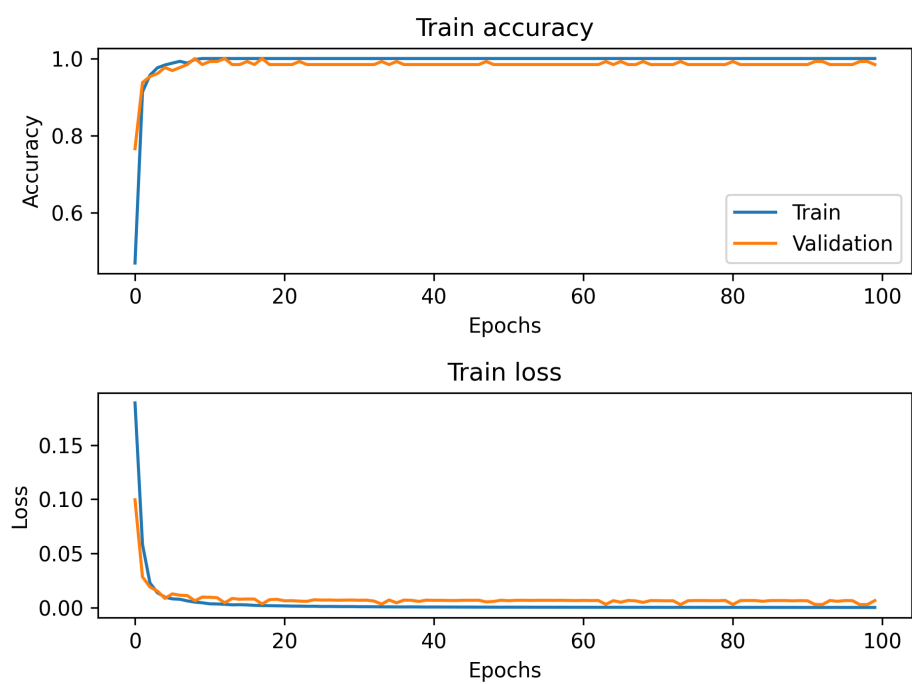


图 11: 经典神经网络模型测试四分类

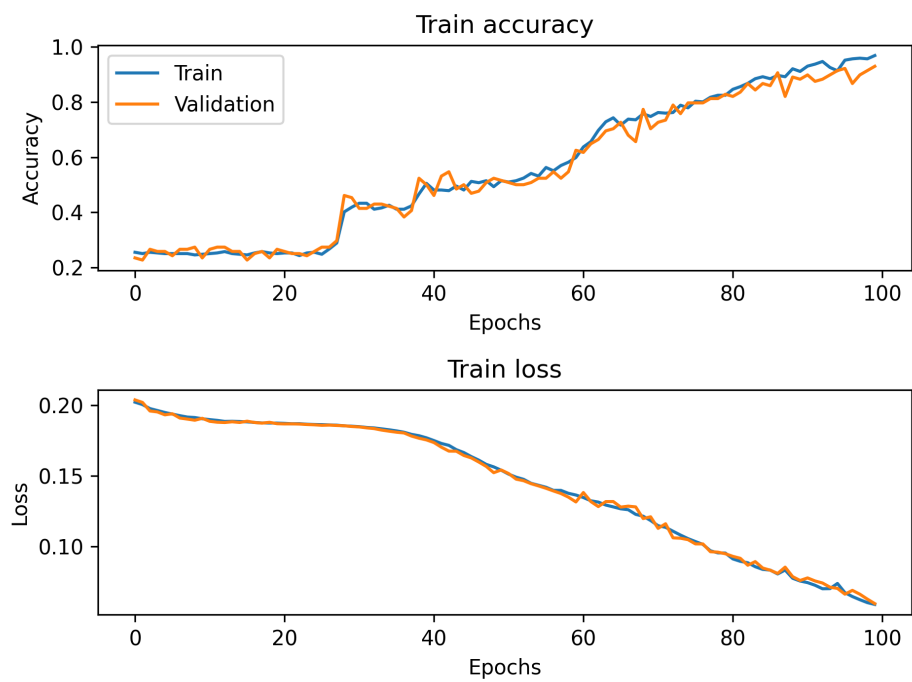


图 12: 混合模型 CQ 测试四分类

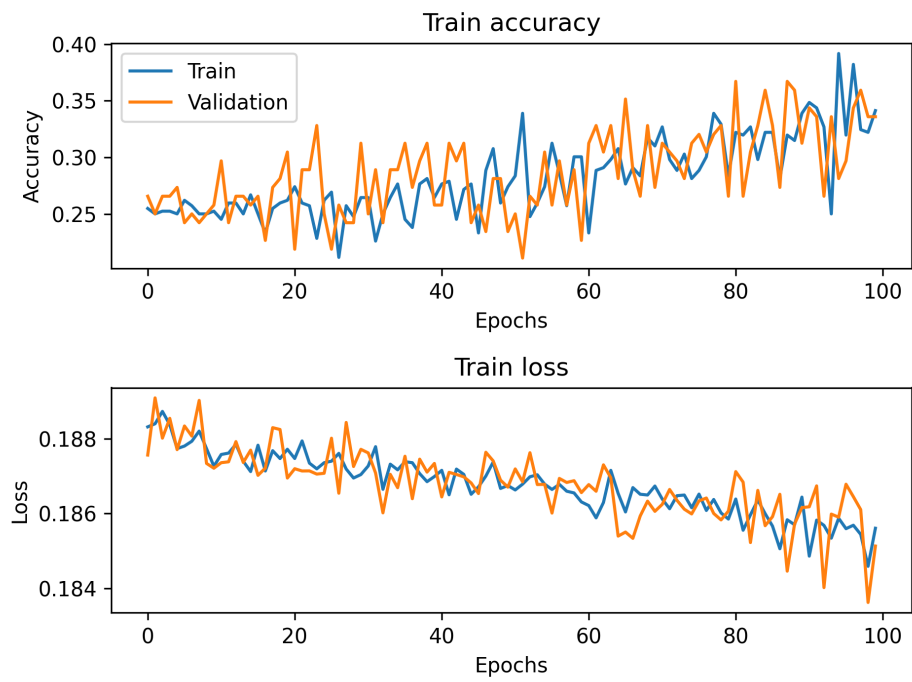


图 13: 混合模型 QC 测试四分类

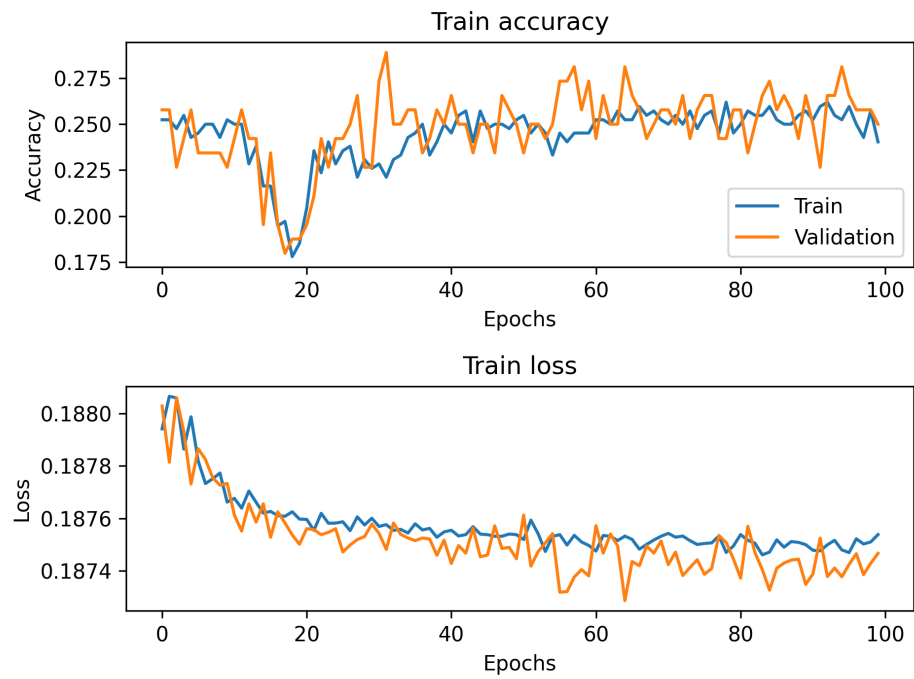


图 14: 混合模型 QQ 测试四分类

在四分类问题中：

- Classical test score: 1.0000
- Hybrid\_CQ test score: 0.9722
- Hybrid\_QC test score: 0.3611
- Hybrid\_QQ test score: 0.2431

对于经典神经网络模型，在四分类问题中，Train accuracy 和 Test accuracy 都达到了 100%，说明经典神经网络模型在训练和测试集上的表现都较好。

在混合模型 CQ 中，Train accuracy 和 Test accuracy 都达到了 97%，但是收敛速度显然小于经典神经网络模型。

在混合模型 QC 中，Train accuracy 和 Test accuracy 都达到了 36%，虽然 loss 曲线始终在下降，但是 Train accuracy 和 Test accuracy 曲线在训练过程处于震荡上升状态，说明混合模型在训练过程中易受噪声影响，且无法收敛到最优解。

在混合模型 QQ 中，Train accuracy 和 Test accuracy 基本不发生变化，无法处理四分类问题。

### 7.7.3 7.7.3 十分类测试

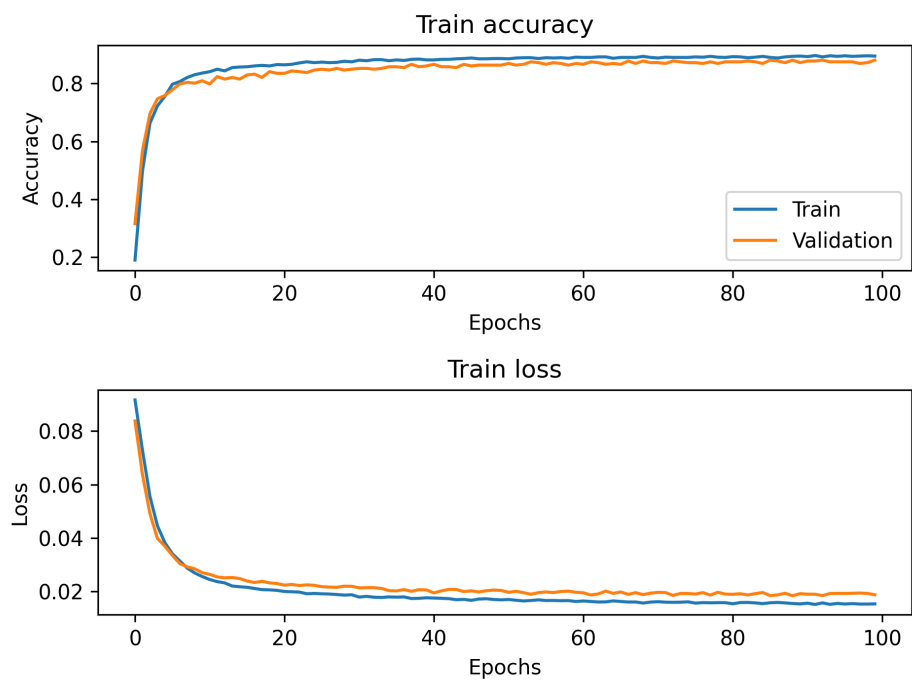


图 15: 经典神经网络模型测试十分类

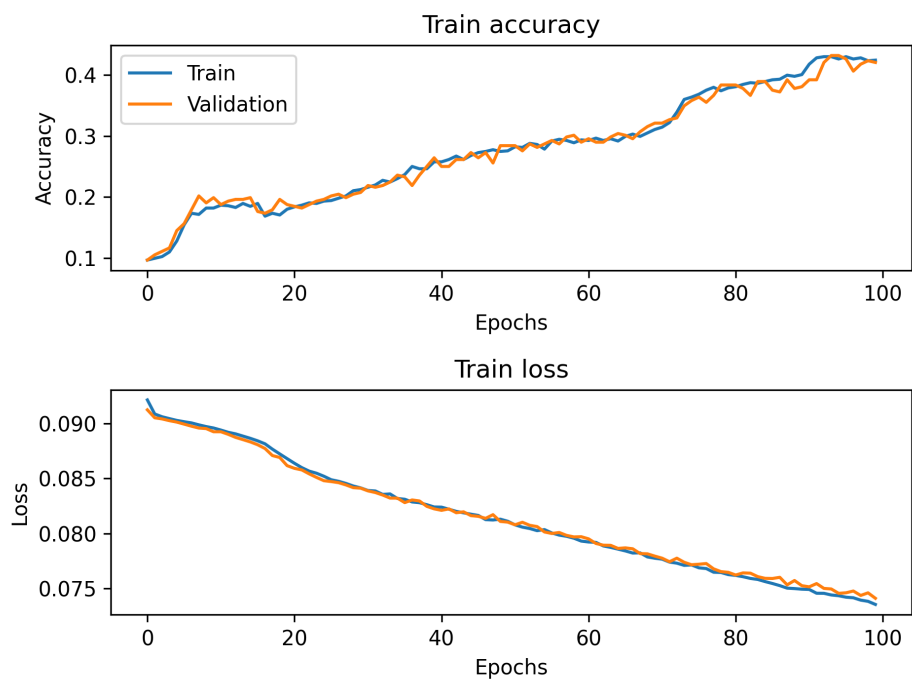


图 16: 混合模型 CQ 测试十分类

在十分类问题中：

- Classical test score: 0.8607
- Hybrid\_CQ test score: 0.4373

对于经典神经网络模型，由于缺乏对超参数的调优，导致在十分类问题中，Train accuracy 和 Test accuracy 没有收敛到 100%.

在混合模型 CQ 中，Train accuracy 和 Test accuracy 都达到了 43%，虽然 loss 曲线始终在下降，但是 Train accuracy 和 Test accuracy 曲线在训练过程处于震荡上升状态，说明混合模型在训练过程中易受噪声影响，

对于混合模型 QC 和 QQ，由于训练时间过长，无法在短时间内完成训练，因此没有进行测试。

## 0.8 八. 总结与心得

量子模型的优势：量子神经网络在参数量较少的情况下，仍然能够取得不错的分类效果，尤其是在二分类任务中。这表明量子模型在某些特定任务中具有一定的优势，尤其是在数据量较小或特征较为简单的情况下。

经典模型的优势：经典 CNN 模型在处理复杂图像数据时表现出色，尤其是在参数量较大的情况下，能够轻松实现高精度分类。经典模型的训练速度也较快，适合大规模数据集的训练。

量子模型的局限性：当前量子计算的硬件和算法仍然存在局限性，量子模型的训练速度较慢，且在处理复杂任务时性能不如经典模型。此外，量子模型的参数量较少，可能在表达能力上存在一定的限制。

本次实验通过比较量子神经网络与经典机器学习算法在 MNIST 数据集上的表现，展示了量子模型在某些特定任务中的潜力。尽管量子模型在当前阶段尚未完全超越经典模型，但随着量子计算硬件和算法的不断发展，未来量子机器学习有望在更多领域取得突破。

通过本次实验，我深入了解了量子机器学习的基本原理和实现方法，并对量子模型与经典模型的性能进行了比较。实验过程中，我认识到量子计算在当前阶段的局限性，但也看到了其未来的潜力。未来，我将继续关注量子计算的发展，探索其在更多领域的应用。