量超融合赛道：基于 Measure-and-Prepare 的量子线路拆分技术及其在 MNIST 图片分类上的应用

Anonymous

2025 年 4 月 2 日

摘要

量子线路拆分技术（Quantum Circuit Cutting）是应对大规模量子计算挑战的重要方法之一。在当前量子计算硬件受限于纠缠操作和逻辑量子比特数量的背景下，拆分技术通过将复杂的量子线路分解为若干较小的子线路，使得在现有的硬件条件下实现大规模量子计算成为可能。其中，“测量-制备”（Measure-and-Prepare）技术是量子线路拆分中的关键方法之一，其主要通过在量子线路中插入测量和再准备步骤，来替代原有的量子操作，极大地简化了量子线路的实现难度。在量子线路拆分的实际应用中，我们以基于量子神经网络的 MNIST 图片分类任务为例，展示了测量-制备技术的实际价值。在这一任务中，每张图片从最原始的 28x28 像素经过预处理变为 5x5 像素的低质量图片，每一个像素点通过单比特旋转门编码到一个量子比特上。编码所有像素点需要使用 25 个量子比特，因此也需要使用高达 2^{25} 维数的量子神经网络对编码后的量子态数据进行处理，这对于目前的量子计算真机和经典模拟器来说都是非常具有挑战性的任务。一方面，量子计算真机受限于量子比特数目、连接性和噪声的影响，很难运行复杂且深度高的量子神经网络；另一方面，经典模拟器的存储开销和计算开销随着量子比特数目的增多而指数级增大。实验结果表明，基于量子线路拆分技术的量子分类器在保持高分类准确率的同时，显著降低了量子计算的复杂性和资源消耗。例如，通过将量子线路拆分技术应用于 MNIST 图片分类任务，我们能够在现有硬件条件下实现原本无法执行的大规模量子计算。这不仅验证了量子线路拆分技术在实际量子计算任务中的有效性，还为未来大规模量子计算任务的实现提供了新的思路。

1 背景介绍

量子线路拆分技术（Quantum Circuit Cutting）[1, 2] 是解决大规模量子计算中固有挑战的重要方法之一。在量子计算领域，尽管在设计和构建大规模量子计算机方面取得了显著进展，但目前的量子计算硬件仍然受到纠缠操作和逻辑量子比特数量的限制。这些限制使得直接实现大规模量子线路变得困难。因此，量子线路拆分技术通过将复杂的量子线路分解为若干较小的子线路，提供了一种有效的方法，使得在现有硬件条件下执行大规模量子计算成为可能。

其中，基于“测量-制备”（Measure-and-Prepare）[3] 的量子线路拆分技术在应对这些挑战方面具有特别重要的作用。这种技术通过在量子线路中插入测量和再准备步骤，来替代原有的量子操作，从而极大地简化了量子线路的实现难度。在“测量-制备”技术中，量子态首先通过测量被转换为经典信息，然后根据这些经典信息重新准备新的量子态。这一过程允许将复杂的量子线路拆分为多个较小的部分，每个部分都可以在现有的量子硬件上独立执行，从而有效地减少了对纠缠操作和逻辑量子比特数量的需求。

此外，时间类切割（Time-like Cuts）[4] 是“测量-制备”技术的一种具体实现形式。时间类切割通过将量子线路中的部分操作替换为“测量-制备”通道，从而实现量子线路的拆分。具体而言，这种方法通过测量现有量子态并基于测量结果重新准备量子态，从而实现量子线路的拆分。在这些过程中，测量和再准备步骤能够有效地减少量子线路中纠缠操作的数量，从而降低计算的复杂性和所需的量子资源。

在量子线路拆分的实际应用中，我们以基于量子神经网络的 MNIST 图片分类任务为例，展示了“测量-制备”技术的实际价值。在这一任务中，每张图片从最原始的 28x28 像素经过预处理变为 5x5 像素的低质量图片，每一个像素点通过单比特旋转门编码到一个量子比特上。编码所有像素点需要使用 25 个量

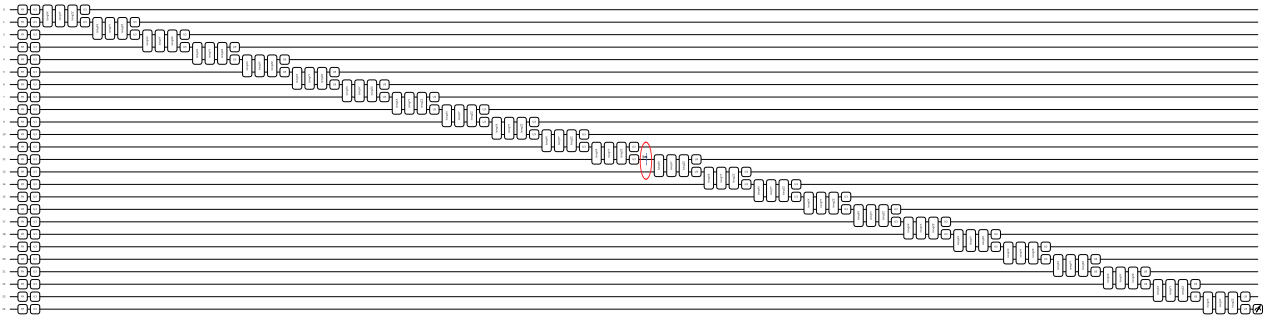

图 1: 线路拆分之前的 25 个量子比特的量子神经网络结构图。红色标记处代表线路切割位置。

子比特，因此也需要使用高达 2^{25} 维数的量子神经网络对编码后的量子态数据进行处理，这对于目前的量子计算硬件和经典模拟器来说都是极具挑战性的任务 [5]。

通过量子线路拆分技术，我们将原本复杂的量子分类器分解为若干较小的子线路，从而能够在当前的量子硬件上进行有效的训练和推理。实验结果表明，基于量子线路拆分技术的量子分类器在保持高分类准确率的同时，显著降低了量子计算的复杂性和资源消耗。例如，通过将量子线路拆分技术应用于 MNIST 图片分类任务，我们能够在现有硬件条件下实现原本无法执行的大规模量子计算。这不仅验证了量子线路拆分技术在实际量子计算任务中的有效性，还为未来大规模量子计算任务的实现提供了新的思路。总的来说，基于“测量-制备”的量子线路拆分技术在提升量子计算的可扩展性、降低计算复杂性和资源需求方面具有显著的价值。未来的研究将继续优化这些拆分技术，并探索其在更广泛量子计算任务中的应用，推动量子计算向实用化迈进。

本工作针对用于 MNIST 图像分类的量子神经网络模型，采用 Measure-and-Prepare 算法对量子线路进行拆分。拆分后的各个量子线路所需要的比特数相比拆分之前的量子线路大幅减少，这使得可以通过内存有限的经典计算机或者比特数有限的量子计算真机模拟训练更大规模的量子神经网络，并利用训练完成的大规模量子神经网络对新的图片数据进行分类。

2 结果展示和结论

基于量子神经网络实现图像分类任务的一大技术瓶颈是利用有限的量子计算资源实现更高分辨率的图片编码，并获得理想的分类效果。由于赛题要求每个像素必须使用单个量子比特进行编码，因此在线路拆分之前，我们总共使用了 25 个量子比特来编码 25 个像素（已经经过下采样和特征归一化处理）的 MNIST 图片，并且搭建对应的量子神经网络。**该量子神经网络特别适用于量子线路拆分操作，因为只需在线路上裁剪一次便可以将线路平均拆成两个子线路部分，每个子线路的运行只需要 13 个量子比特。**由于赛题只要求完成图像的二分类任务，所以我们只测量最后一个比特来作为量子神经网络的模型预测输出。我们在图1中给出了实际编程中所使用的量子神经网络的结构图。拆分后的两部分子线路的结构图请参考图6和图7。在整个算法实现中，除了这两部分子线路是量子计算部分，其他所有的计算都属于经典计算。在线下实验中，实验设备采用 CPU 的型号为 AMD EPYC 7302 16-Core Processor @ 3.0GHz，计算内存为 128GB。

对于部分有兴趣的读者，请阅读3.1章节来获取有关图片的量子态编码、量子神经网络的具体结构和网络训练配置的内容。请读者移步3.2章节来获取有关基于 Measure-and-Prepare 的量子线路拆分技术的内容。

我们基于 PyQPanda 实现了拆分后的量子神经网络的训练和推理过程，相关代码保存在 `train.py` 和 `eval.py` 文件中。我们同时提供了 `readme.md` 文件供读者方面地复现文档中涉及到的实验结果。在 5000 个训练样本，1000 个测试样本，学习率设置为 0.1，优化器选择为 Adam，最大训练迭代次数设置为 100 的情况下，**结合基于 Measure-and-Prepare 的量子线路拆分技术，我们设计的量子神经网络在测**

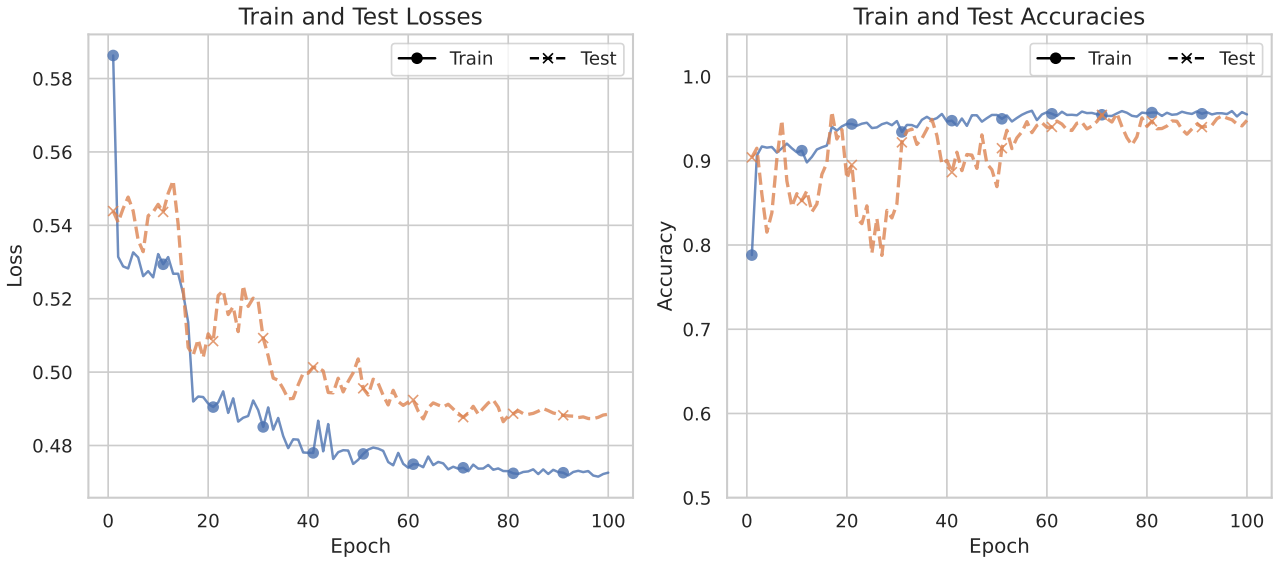

图 2: 结合 Measure-and-Prepare 的量子线路拆分技的量子神经网络在训练集和测试集上的损失函数变化曲线（左）以及分类准确率变化曲线（右）

试集上达到 95.3% 的分类准确率。

我们在图2提供了在训练集和测试集上的损失函数变化曲线以及分类准确率变化曲线。实验结果显示，在训练初期，训练和测试的损失都较高，但随着训练的进行，损失逐渐下降，准确率显著提升。具体来说，训练和测试的损失在前 20 个 epoch 内迅速下降，随后逐渐趋于平稳，在第 100 个 epoch 时，训练损失降至约 0.475，而测试损失约为 0.49。

从准确率来看，训练和测试的准确率在前 20 个 epoch 内迅速提升，随后逐渐趋于平稳。在第 100 个 epoch 时，训练准确率达到约 97%，而测试准确率也达到 95.3%。这表明，通过量子线路拆分技术，可以获得高分类准确率。同时，我们在实验前期准备过程中发现，如果直接使用经典计算机运行未经拆分的 25 个量子比特的线路，内存占用将超过我们实验设备的最大内存（128GB），而分别运行拆分后的量子线路所使用的内存控制在 18GB 以内。这表明通过量子线路拆分技术可以显著降低计算复杂性和资源消耗，使得我们可以在有限资源的设备上模拟和运行更大规模的量子线路。

进一步分析结果可以发现，训练过程中训练和测试的损失和准确率曲线趋于平稳，说明模型在当前训练参数下具有良好的泛化能力。学习率设置为 0.1 在本实验中表现出色，Adam 优化器在 100 次迭代内有效地优化了模型参数。

总结来说，本次实验验证了量子线路拆分技术在 MNIST 图片分类任务中的有效性。通过将复杂的量子线路分解为较小的子线路，并结合测量-准备方法，成功在现有硬件条件下实现了高效的量子计算。在未来的研究中，可以进一步优化这些拆分技术，提高量子线路拆分的效率和灵活性，探索其在更多应用场景中的可行性，包括量子机器学习、量子模拟和量子优化等领域。通过不断完善这些技术，量子计算将在科学研究和工业应用中发挥越来越重要的作用，推动量子技术的实用化进程。

3 算法原理思路

在这一章节中，我们将介绍如何基于 PyQPanda 编码 MNIST 数据集、搭建量子神经网络、对量子神经网络进行拆分、使用训练集中的图片训练量子神经网络，以及使用训练好的网络对测试集中的图片类别进行预测。上述这五个步骤将在接下来的两个子章节中进行说明，首先我们将在3.1章节介绍算法所使用的量子态编码方法和量子神经网络的结构、训练方法，然后在3.2章节介绍量子线路拆分如何作用到量子神经网络上面。

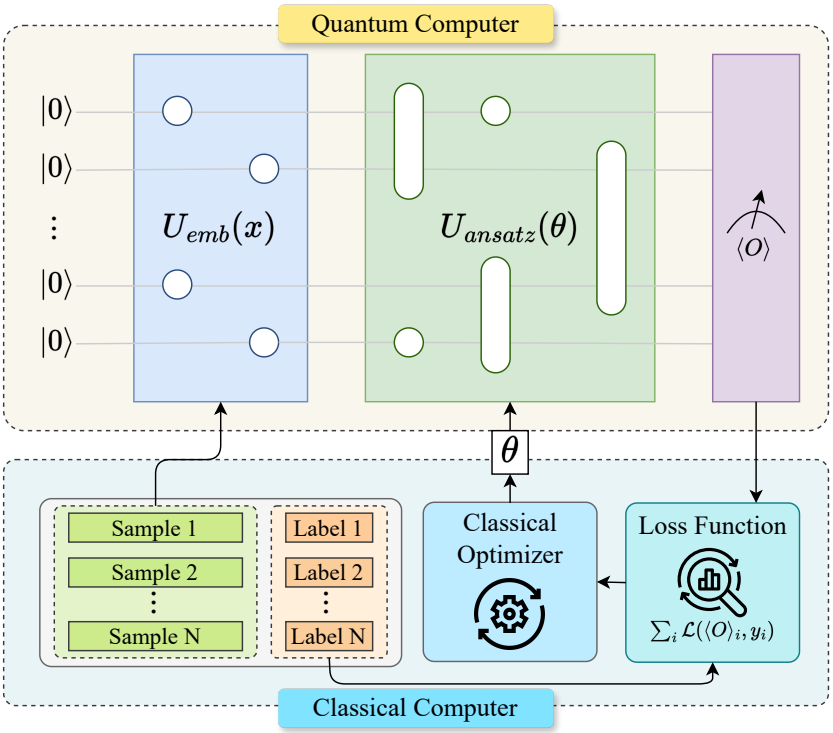

图 3: 量子神经网络示意图

3.1 量子神经网络

量子神经网络（Quantum Neural Networks, QNNs）是将量子计算与神经网络模型相结合的一种新兴计算框架。它利用量子计算的特性，如量子叠加和量子纠缠，以实现经典神经网络在某些任务上无法比拟的计算能力和效率。图3中给出了量子神经网络的示意图。量子神经网络可以类比于经典神经网络，它们都含有可学习的参数，不同点在于经典数据输入到量子神经网络时需要将通过嵌入层 $U_{emb}(x)$ 将经典数据 x 编码到量子态，并且在经过含有可学习的参数 θ 的学习层 $U_{ansatz}(\theta)$ 的处理后，需要对量子态进行多次测量得到观测量的期望值 $\langle O\rangle$。我们可以在经典计算机上运用适当后处理将观测量的期望值转化为 QNN 的模型预测，并定义该预测值与样本的实际标签 y 的差异（如二分类交叉熵）作为损失函数。利用经典计算机中成熟的优化器依照损失函数进行梯度下降并求取更新后的参数 θ 的值，我们便可以像训练经典神经网络那样训练量子神经网络。训练好的量子神经网络可以预测测试集中样本的标签。

本文研究的主要目标是利用 QNN 进行 MNIST 图片分类，并引入量子线路拆分技术以应对现有量子硬件的限制。由于赛题要求每个像素必须使用一个量子比特进行编码，所以在引入量子线路拆分技术之前我们需要使用 25 个量子比特去映射 25 个像素点。我们首先介绍如何构建量子神经网络。

3.1.1 图片到量子态的映射

在量子神经网络中，输入数据（如题目中所给的 MNIST 图片）需要映射到量子态，题目要求每一个像素必须使用一个量子比特来编码。这一过程通过以下步骤完成：

1. **图片预处理**：首先，将原始的 28x28 像素的图片进行降维处理，降低到 5x5 像素。这一步的目的是减少量子比特的需求，从而降低硬件实现的复杂性。这里我们用数学符号 $x_1, x_2, \ldots, x_{25}$ 表示一张图片的 25 个像素值。在编码到量子比特之前，我们将每张图片进行了归一化，也就是

$$x_j = \frac{x_j}{\sqrt{\sum_{j=1}^{25} (x_j)^2}}. \tag{1}$$

2. **量子态编码**：然后，每一个像素点通过单比特旋转门编码到一个量子比特上。具体而言，像素值被归

一化为一个量子态幅度，并通过旋转门 R_y 操作将其映射到 Bloch 球面上的一个点。这一步生成的量子态可以表示为 $|\psi_i\rangle = R_y(x_i)|0\rangle = \cos(x_i/2)|0\rangle + \sin(x_i/2)|1\rangle$。那么编码到 25 个量子比特所形成的初始量子态为

$$|\Psi(x)\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_{25}\rangle = R_y(x_1)|0\rangle \otimes R_y(x_2)|0\rangle \otimes \cdots \otimes R_y(x_{25})|0\rangle. \tag{2}$$

3.1.2 拟设

在量子神经网络中，拟设（Ansatz）指的是量子线路的具体结构和参数化形式。为了构建一个有效的 QNN，我们需要设计一个合理的量子线路结构，使其能够有效地表示和处理输入数据。常见的量子线路拟设包括参数化的量子门阵列，如旋转门、控制非门（CNOT）、及其他纠缠门。拟设的设计原则包括：

1. **表达能力**：量子线路必须具有足够的表达能力，以捕捉输入数据的复杂特征。
2. **参数优化**：量子线路中的参数需要能够通过训练过程进行优化，以最小化分类误差。

在具体实现中，我们采用如图4所示的层叠结构的量子线路，每一个块（block）包含多个参数化的单量子门 $U3$ 门和若干个双比特量子门 IsingXX, IsingYY 和 IsingZZ 门，我们称每一个由单量子门 $U3$ 门和若干个双比特量子门组成的块为一个可学习块（learnable block）。其中 $U3$ 门的数学表达式为

$$U_3(\theta,\phi,\delta) = \begin{bmatrix} \cos(\theta/2) & -\exp(i\delta)\sin(\theta/2) \\ \exp(i\phi)\sin(\theta/2) & \exp(i(\phi+\delta))\cos(\theta/2) \end{bmatrix} \tag{3}$$

还有双比特量子门 IsingXX, IsingYY 和 IsingZZ 门，它们的数学表达式分别为

$$\begin{aligned}
\mathrm{IsingXX}(\theta) &= \exp\left(-i\frac{\theta}{2}(X\otimes X)\right) = \begin{bmatrix} =\cos(\theta/2) & 0 & 0 & -i\sin(\theta/2) \\ 0 & \cos(\theta/2) & -i\sin(\theta/2) & 0 \\ 0 & -i\sin(\theta/2) & \cos(\theta/2) & 0 \\ -i\sin(\theta/2) & 0 & 0 & \cos(\theta/2) \end{bmatrix} \\
\mathrm{IsingYY}(\theta) &= \exp\left(-i\frac{\theta}{2}(Y\otimes Y)\right) = \begin{bmatrix} \cos(\theta/2) & 0 & 0 & i\sin(\theta/2) \\ 0 & \cos(\theta/2) & -i\sin(\theta/2) & 0 \\ 0 & -i\sin(\theta/2) & \cos(\theta/2) & 0 \\ i\sin(\theta/2) & 0 & 0 & \cos(\theta/2) \end{bmatrix} \\
\mathrm{IsingZZ}(\theta) &= \exp\left(-i\frac{\theta}{2}(Z\otimes Z)\right) = \begin{bmatrix} e^{-i\theta/2} & 0 & 0 & 0 \\ 0 & e^{i\theta/2} & 0 & 0 \\ 0 & 0 & e^{i\theta/2} & 0 \\ 0 & 0 & 0 & e^{-i\theta/2} \end{bmatrix}
\end{aligned} \tag{4}$$

这些参数通过经典优化器进行调整，直到达到最大训练迭代次数。经过简单计算可以得出，假设线路宽度为 L，那么共有 $3(L-1)+1$ 个 $U3$ 门，IsingXX, IsingYY 和 IsingZZ 门各有 $L-1$ 个。所以可学习层总共有 $6L-5$ 个量子门，$12L-9$ 个可学习的参数。

3.1.3 量子线路的测量输出和损失函数

在量子神经网络中，量子线路的输出通过测量得到，测量结果用于计算损失函数以指导模型的训练。具体而言，我们采用对量子线路最后一个量子比特进行测量的方法，并利用交叉熵损失函数完成 MNIST 图片的二分类任务。具体步骤如下：

1. **量子态测量**：在量子线路的末端，我们仅测量最后一个量子比特。测量结果为比特状态 $|0\rangle$ 或者 $|1\rangle$，通过多次测量可以得到该比特坍缩到 $|0\rangle$ 的概率。假设测量次数为 N，$|0\rangle$ 出现的概率为 n_0，那么该比特坍缩到 $|0\rangle$ 态的概率估计值为：

$$p(0) = \frac{n_0}{N}. \tag{5}$$

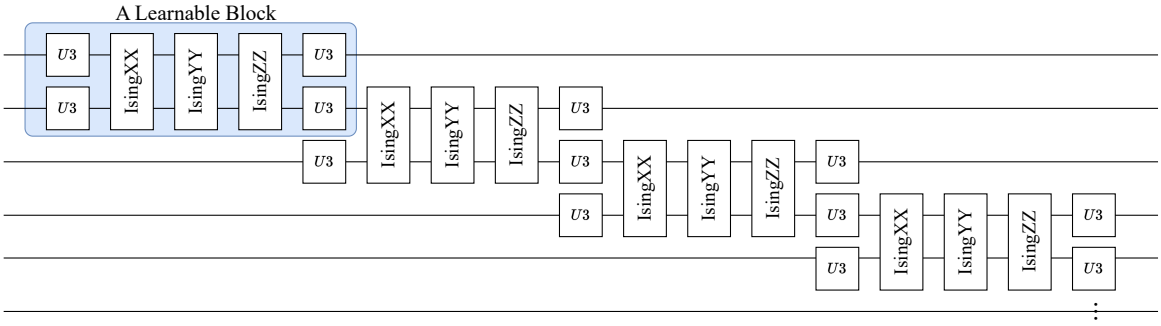

图 4: 量子神经网络可学习层的结构设计

2. **预测结果**：测量最后一个量子比特坍缩到 $|0\rangle$ 的概率用于二分类任务中的类别预测。我们可以预先设定一个阈值（实验中设置为 0.5），如果 $p(0)$ 大于阈值，则预测为类别为正类（手写数字 3），否则预测为类别负类（手写数字 6）。

3. **损失函数**：为了训练量子神经网络，我们使用交叉熵损失函数来衡量预测结果与真实标签之间的差异。交叉熵损失函数定义为：

$$\mathcal{L} = -\sum_i \left(y_i \log p_i(0) + (1 - y_i) \log(1 - p_i(0))\right), \tag{6}$$

其中 y 为真实标签，取值为 0 或 1；$p(0)$ 为测量得到的量子比特坍缩到 $|0\rangle$ 态的概率。

4. **参数优化**：在训练过程中，通过最小化交叉熵损失函数来优化量子线路中的参数。优化算法可以采用经典的梯度下降或其变种，我们在本工作中采用深度学习中常用的 Adam 优化器。每次迭代中，根据损失函数的梯度更新量子线路中的参数，从而逐步提高分类性能。

通过上述过程，我们可以有效地利用量子神经网络实现对 MNIST 图片的二分类任务。量子线路的测量输出为训练和预测提供了基础，而交叉熵损失函数则保证了模型的优化方向。在训练过程中，通过对量子线路最后一个量子比特的测量以及损失函数的最小化，我们能够逐步提高模型的分类准确率，从而实现高效的量子计算机视觉应用。

这种方法不仅充分利用了量子计算的潜力，同时也有效地结合了经典机器学习的优化策略，为实现复杂的量子神经网络应用提供了坚实的基础。下一节中，我们将详细探讨量子线路拆分技术，尤其是“测量-制备”技术及其在量子神经网络中的应用。

3.2 基于 Measure-and-Prepare 的量子线路拆分技术

尽管上一节中所提出的量子神经网络理论上已经可以拿来训练和推理来完成 MNIST 图像分类的任务，但受限于目前的量子计算真机的比特数目，直接将 25 个量子比特的量子神经网络进行部署在量子计算真机上一般比较困难；即使使用全振幅的经典模拟器来运行，所占用的内存开销也是特别大的。因此我们采用基于 Measure-and-Prepare 的量子线路拆分技术使得规模较大的线路可以被拆分为规模较小的子线路，子线路相比原始线路的线路宽度大大缩小，并且可以独立地运行在量子真机或者经典模拟器上。子线路运行后的结果经过适当的后处理，我们便可以近似得到原始线路的期望输出。这相当于是一种“时间换空间”的解决方法，也就是说，量子线路拆分技术使得我们可以用有限资源的量子真机或者经典模拟器去运行或仿真更大规模的量子线路，代价是需要额外运行多次子线路以及额外的经典后处理。下面我们将介绍量子线路拆分技术的主要概念以及算法细节。

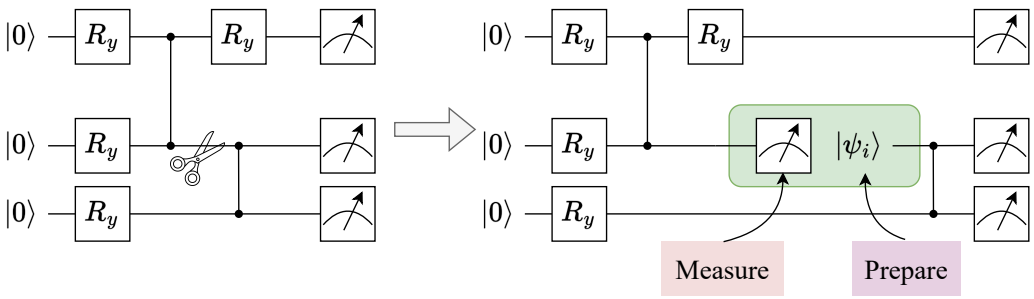

图 5: 测量和制备（Measure-and-Prepare）示意图

3.2.1 Measure-and-Prepare

“测量-制备”（Measure-and-Prepare）是量子线路拆分技术中一种关键的方法，通过在量子线路中插入测量和再准备步骤，将复杂的量子操作简化为易于实现的局部操作。这种方法的核心思想是利用测量将量子态转换为经典信息，再基于这些经典信息重新准备新的量子态，从而有效地打破量子纠缠，使得原本复杂的量子操作变得更加易于实现。

在量子计算中，量子态的变换通常表示为一个完全正定（completely positive）且是迹保持（trace-preserving）的映射，即 CPTP 映射。具体来说，测量-制备通道是一种特殊的量子通道，其形式为：

$$\Phi(\rho) = \sum_k R_k \mathrm{tr}(F_k \rho)$$

其中，$\{F_k\}$ 是正定操作值测量（positive operator-valued measure，POVM），每个 R_k 是密度矩阵。该形式称为 “Holevo form”，以其提出者 Holevo 命名。

测量-制备方法可以打破纠缠的特性，即任意纠缠态在经过该通道后都会变为分离态。具体地，一个映射 Φ 是打破纠缠的（entanglement breaking），如果对于任意的密度矩阵 Γ，$(I \otimes \Phi)(\Gamma)$ 是可分的，即使 Γ 是纠缠态。形式上，测量-制备方法也可以表示为：

$$\Phi(\rho) = \sum_k |\psi_k\rangle\langle\psi_k|\langle\phi_k|\rho|\phi_k\rangle$$

其中，$|\psi_k\rangle$ 和 $|\phi_k\rangle$ 是正交态，并且满足 $\sum_k |\phi_k\rangle\langle\phi_k| = I$。通过这种方式，测量-制备方法能够将复杂的量子操作分解为局部测量和态准备，从而简化了量子线路的实现。

要实现测量-制备方法，可以按照以下步骤操作，我们也在图：

1. **测量**：对输入量子态 ρ 进行测量，测量算子为 $\{F_k\}$。测量结果为 k 的概率为 $p_k = \mathrm{tr}(F_k\rho)$，测量后量子态坍缩为：
$$\rho_k = \frac{F_k \rho F_k^\dagger}{\mathrm{tr}(F_k \rho)}$$
2. **经典信息传输**：将测量结果 k 通过经典通道传输给接收者。
3. **态制备**：接收者根据测量结果 k，准备对应的量子态 R_k。这一过程可以表示为：
$$R_k = p_k |\psi_k\rangle\langle\psi_k|$$
其中，$|\psi_k\rangle$ 是预先设定的量子态。
4. **重构量子线路**：将插入测量-制备通道后的量子线路重新组合，形成若干较小的子线路。这些子线路可以独立执行，并通过经典通道进行信息传递和协作。

在量子线路拆分的实际应用中，特别是在时间类切割（Time-like Cuts）背景下，测量-制备方法的作用尤为重要。时间类切割通过将量子线路中的部分操作替换为测量-制备，从而实现量子线路的拆分。这种方法有效地减少了量子线路中纠缠操作的数量，从而降低了计算的复杂性和所需的量子资源。

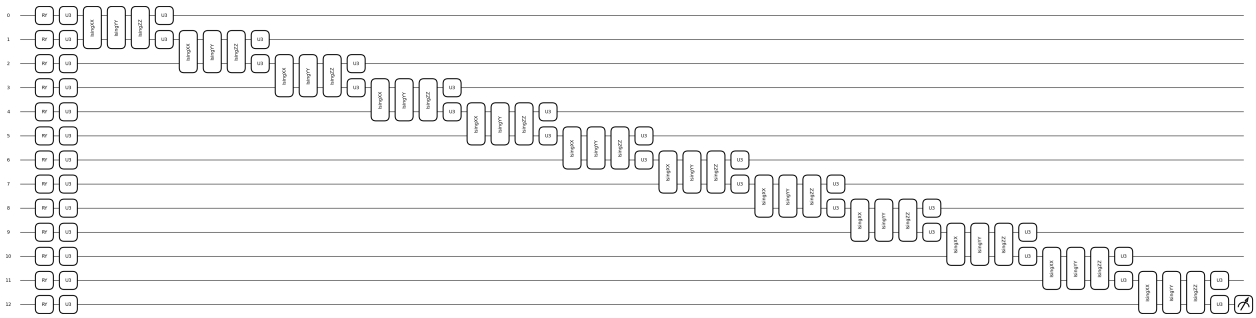

图 6: 拆分后的第一部分子线路

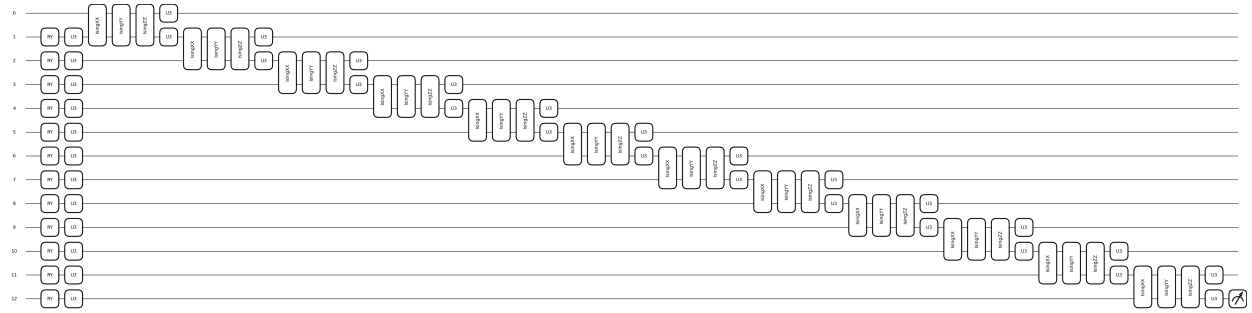

图 7: 拆分后的第二部分子线路

在实际应用中，例如在 MNIST 图片分类任务中，通过量子线路拆分技术，我们可以将原本复杂的量子分类器分解为若干较小的子线路，从而能够在当前的量子硬件上进行有效的训练和推理。对于图1中的 25 量子比特的线路，经过线路拆分后会得到两个子线路。如图6和图7所示，运行每个线路只需要 13 个量子比特。实验结果表明，基于量子线路拆分技术的量子分类器在保持高分类准确率的同时，显著降低了量子计算的复杂性和资源消耗。

3.2.2 算法细节

在量子计算中，基于 Measure-and-Prepare 的量子态测量和制备是一种有效的量子线路拆分技术。在的算法实际实现中，我们利用泡利（Pauli）测量和泡利态制备来实现 Measure-and-Prepare，因为泡利测量和泡利制备在实际的量子硬件中是非常容易实现的。我们通过将量子线路分割为多个子线路，并利用经典信息协调计算，从而实现复杂量子线路在现有硬件上的执行。具体地，对于任何一个量子态 ρ，都可以写成下面这种形式

$$\rho = \frac{1}{2}\sum_{k=1}^{8} c_k \mathrm{tr}(\rho\sigma_k)\rho_k, \tag{7}$$

其中 σ_k 表示 Pauli 矩阵，用来进行量子测量使得被测量的量子比特坍缩到 Pauli 矩阵对应的本征态 ρ_k 上，并得到对应的测量结果 c_k（取值为 1 或者-1，也就是本征态对应的本征值）。Pauli 矩阵包括四个基本矩阵 I, X, Y, Z, 这四个矩阵被分摊为：

$$\begin{aligned}
\sigma_1 = \sigma_2 = I &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \\
\sigma_3 = \sigma_4 = X &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \\
\sigma_5 = \sigma_6 = Y &= \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \\
\sigma_7 = \sigma_8 = Z &= \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.
\end{aligned} \tag{8}$$

我们也给出 Pauli 矩阵对应的本征态，对应关系为

$$\begin{aligned} F_1 = F_7 &= |0\rangle\langle 0|, \\ F_2 = F_8 &= |1\rangle\langle 1|, \\ F_3 &= |+\rangle\langle +|, \\ F_4 &= |-\rangle\langle -|, \\ F_5 &= |+i\rangle\langle +i|, \\ F_6 &= |-i\rangle\langle -i|. \end{aligned} \tag{9}$$

利用公式7，我们可以方便地将一个量子线路拆分成两个独立的子线路，子线路相比原来的线路所占用的量子比特数目更少，因此可以使用量子比特数目较少的量子计算机运行更大规模的量子线路，或者是在经典计算上用更少的内存开销去仿真更大规模的量子线路，当然代价是引入了额外的 Measure-and-Prepare 的计算开销。针对公式7，下面我们给出具体的算法流程，并在图5中给出了一个算法流程示意图。

1. 切割点的选择：量子线路切割的第一步是确定切割点。切割点通常选择在产生大量纠缠的量子门（如双比特量子门的前后位置）处。

2. 插入测量操作：对量子态 ρ 进行测量，选择适当的 Pauli 矩阵 $\{\sigma_k\}_{k=1}^{8}$。测量结果为 k 的概率为 $p_k = \mathrm{tr}(\sigma_k \rho)$，测量后量子态坍缩为：
$$\rho_k = \frac{F_k \rho F_k^\dagger}{\mathrm{Tr}(F_k \rho)}$$

3. 插入态制备操作：接收者根据测量结果 k，准备对应的量子态 R_k。这一过程可以表示为：
$$R_k = p_k \rho_k$$

4. 重构子线路：根据插入测量-制备操作的位置，将原始量子线路分割为多个子线路。这包括首先将原始量子线路分割为若干子线路，每个子线路包含一个测量操作和一个态准备操作，然后将每个子线路中的测量和态准备操作组合，形成新的量子线路结构。

5. 执行子线路并整合结果：在量子硬件上独立执行每个子线路，并收集其测量结果。在量子硬件上独立执行每个子线路，收集每个子线路的测量结果。最后根据经典信息，将各子线路的测量结果进行整合，得到最终的计算结果。根据公式和公式，最终的计算结果为
$$\hat{\rho} = \frac{1}{2} \sum_{k=1}^{8} c_k R_k$$
当 Measure-and-Prepare 这个操作执行的次数足够多时，通过后处理恢复出的量子态 $\hat{\rho}$ 将近似趋近于未拆分量子线路前的理想输出 ρ。假设近似误差一定，那么要求的执行次数随着线路切割次数的增大而呈现指数级增大。因此在实际应用时，应尽量选择合适的切割位置，使得尽可能以最少的切割次数将原始线路分割开来。

通过这些步骤，可以有效地利用泡利（Pauli）测量和泡利态制备来实现 Measure-and-Prepare，进而实现量子线路拆分，从而在现有的量子硬件上实现复杂的量子计算任务。例如，在 MNIST 图片分类任务中，通过量子线路拆分技术，可以将原本复杂的量子分类器分解为若干较小的子线路，从而能够在当前的量子硬件上进行有效的训练和推理。

4 结论和展望

本文详细探讨了基于“测量-准备”方法的量子线路拆分技术及其在 MNIST 图片分类中的应用。通过引入测量和再准备步骤，量子线路的复杂性得以显著降低，使得在现有硬件条件下执行大规模量子计算成

为可能。实验结果表明，采用这种技术的量子分类器不仅在分类准确率上表现出色，而且在计算资源和复杂性上有显著的降低。这一技术展示了其在实际量子计算任务中的巨大潜力，证明了量子线路拆分技术的有效性和实用性。

展望未来，量子线路拆分技术的优化和扩展将继续推进量子计算的发展。随着量子硬件技术的不断进步，结合测量-准备等拆分技术，有望实现更大规模、更复杂的量子计算任务。未来研究将致力于提高量子线路拆分的效率和灵活性，探索其在更多应用场景中的可行性，包括量子机器学习、量子模拟和量子优化等领域。通过不断完善这些技术，量子计算将在科学研究和工业应用中发挥越来越重要的作用，推动量子技术的实用化进程。

5 代码分析

本文实现所涉及到的所有实验均基于 Python 语言编写，Python 版本为 3.9。第三方包方面，我们依赖 numpy 包实现给定训练和测试数据集的读取和处理，并基于最新版本的 vqnet 实现线路的构建、Measure-and-Prepare 的实现和参数的优化，下面将详细解释代码的组成以及逻辑思路。

本文的线路搭建以及 Measure-and-Prepare 的实现保存在文件 cutcircuit.py 文件中，其中声明了 QModel 类，定义了模型所包含的参数 weights 的大小与数据类型以及 forward 函数 q_net()。代码为

```
class QModel(Module):
    def __init__(self):
        super(QModel, self).__init__()
        self.q_net = q_net
        self.weights = Parameter(shape=(1, 15*num_wires-15), dtype=kcomplex64)

    def forward(self, input):
        return self.q_net(self.weights, input)
```

Listing 1: QModel 类的定义

我们将原有的 25 比特量子线路拆分为两个 13 比特的子线路，分别命名为 circuit_front() 和 circuit_back()，采用 VQC_AngleEmbedding() 方法对 5x5 大小的 features 数据在 25 个比特上进行编码，并按顺序搭建由 U3 门、IsingXX 门、IsingYY 门和 IsingZZ 门构成的含有模型参数 weights 的学习层。值得注意的是，作为原 25 比特量子线路中被切割的比特，前半部分线路的最后一个比特将与后半部分线路的第一个比特通过 Measure-and-Prepare 相关联。前半部分线路中最后一个比特将用于 Measure 的实现，采用四个基本泡利矩阵 I, X, Y, Z 对最后一个量子比特进行测量，后半部分线路中对应的第一个比特将用于 Prepare 的实现，根据不同的泡利测量矩阵搭建对应的量子门 $I, X, H, -H - S-$。相应的代码如下，

```
def obs_list(wire):
    # measurements for the front circuit part
    return [I(wires = wire), PauliX(wires = wire), PauliY(wires = wire), PauliZ(wires = wire)]

def circuit_front(qm, features, wires, weights):
    VQC_AngleEmbedding(features[:, :num_qubits], wires, qm, rotation = 'Y')
    for ind, w in enumerate(wires):
        if ind < num_qubits - 1:
            if ind == 0:
                count = 12*num_wires-12
                u3(qm, wires[ind], tensor.to_tensor(weights[count+3*ind: count+3*ind+3]))
            count = 3*num_wires-3
            u3(qm, wires[ind+1], tensor.to_tensor(weights[count+3*ind: count+3*ind+3]))
            isingxx(qm, [w, wires[ind+1]], weights[3*ind+0])
            isingyy(qm, [w, wires[ind+1]], weights[3*ind+1])
            isingzz(qm, [w, wires[ind+1]], weights[3*ind+2])
            count = 6*num_wires-6
            u3(qm, wires[ind], tensor.to_tensor(weights[count+3*ind: count+3*ind+3]))
            count = 9*num_wires-9
```

```
            u3(qm, wires[ind+1], tensor.to_tensor(weights[count+3*ind: count+3*ind+3]))
    return qm

def circuit_back(qm, features, wires, weights, idx):
    VQC_AngleEmbedding(features[:, num_qubits:], wires[1:], qm, rotation = 'Y')

    # the prepare gates for the back circuit part
    # the first qubit in the back part is the last qubit in the front part
    if idx == 0:
        i(qm, wires[0])
    elif idx == 1:
        x(qm, wires[0])
    elif idx == 2:
        hadamard(qm, wires[0])
    elif idx == 3:
        hadamard(qm, wires[0])
        s(qm, wires[0])

    for ind, w in enumerate(wires):
        if ind >= num_qubits - 1 and ind < num_wires - 1:
            if ind == num_qubits - 1:
                count = 12*num_wires-12
                u3(qm, wires[ind-num_qubits+1], tensor.to_tensor(weights[count+3*ind: count+3*ind+3]))
            count = 3*num_wires-3
            u3(qm, wires[ind-num_qubits+2], tensor.to_tensor(weights[count+3*ind: count+3*ind+3]))
            isingxx(qm, [w-num_qubits+1, wires[ind-num_qubits+2]], weights[3*ind+0])
            isingyy(qm, [w-num_qubits+1, wires[ind-num_qubits+2]], weights[3*ind+1])
            isingzz(qm, [w-num_qubits+1, wires[ind-num_qubits+2]], weights[3*ind+2])
            count = 6*num_wires-6
            u3(qm, wires[ind-num_qubits+1], tensor.to_tensor(weights[count+3*ind: count+3*ind+3]))
            count = 9*num_wires-9
            u3(qm, wires[ind-num_qubits+2], tensor.to_tensor(weights[count+3*ind: count+3*ind+3]))
    return qm
```

Listing 2: 经过切割后的两部分线路

对于前半部分线路经过四种测量得到的四个 Measure 结果以及后半部分线路通过添加不同量子门使用与原 25 比特量子线路中对应的测量相同的测量操作得到的四个 Prepare 结果，我们采用不同的经典计算方法进行数据处理，这由不包含可学习参数的函数 process_tensor() 实现。我们采用了 vqnet 中基于 QTensor 的向量化计算函数，从而使代码能够对 batch 输入下的结果进行批处理。代码如下，

```
CHANGE_OF_BASIS = np.array([[1.0, 1.0, 0.0, 0.0], [-1.0, -1.0, 2.0, 0.0], [-1.0, -1.0, 0.0, 2.0], [1.0,
    -1.0, 0.0, 0.0]])

def process_tensor(results, n_prep, n_meas):
    '''
    n_prep is the num of the prepare operations in Measure-and-Prepare
    n_meas is the num of the measure operations in Measure-and-Prepare
    in the front part of the circuit, we measure the last qubit, so n_prep is 0 and n_meas is 1
    in the back part of the circuit, we prepare in the first qubit, so n_prep is 1 and n_meas is 0
    '''
    results = tensor.reshape(results, (4, -1))
    results = tensor.transpose(results, [1, 0])
    for _ in range(n_meas):
        final_result = results

    for _ in range(n_prep):
        for i in range(len(CHANGE_OF_BASIS)):
            change = tensor.to_tensor(CHANGE_OF_BASIS[i])
            if i == 0:
                temp_result = tensor.sums(tensor.mul(change, results), axis=1)
                final_result = tensor.reshape(temp_result, (-1, 1))
```

```
            else:
                temp_result = tensor.sums(tensor.mul(change, results), axis=1)
                temp_result = tensor.reshape(temp_result, (-1, 1))
                final_result = tensor.concatenate([final_result, temp_result], axis=1)
    final_result *= np.power(2, -(n_meas + n_prep) / 2)
    return final_result
```

Listing 3: 结果处理函数

最终在前向函数中，我们分别搭建 13 比特的量子线路，实现前文提及的对前半部分线路的四种测量以及对应的对后半部分线路的四种制备。经过处理的前半部分 13 比特子线路 Measure 结果与经过处理的后半部分 13 比特子线路 Prepare 结果通过内积得到整个 25 比特线路的执行结果。代码为，

```
H = QTensor([[1.+0.j, 0.+0.j,], [0.+0.j, 0.+0.j,]], dtype=kcomplex64)
o = tensor.dense_to_csr(H)

num_wires = 25
num_qubits = 13
wires = list(range(num_wires))

def q_net(weights, features):
    weights = tensor.squeeze(weights)

    #measure method
    for i in range(len(obs_list(num_qubits - 1))):
        qm_front = QMachine(num_qubits)
        circuit_front(qm_front, features, wires, weights)
        if i == 0:
            result_front = expval(qm_front, wires[num_qubits - 1], obs_list(num_qubits - 1)[i])
        else:
            result_front = tensor.concatenate([result_front, expval(qm_front, wires[num_qubits - 1],
     obs_list(num_qubits - 1)[i])])

    #prepare method
    for i in range(len(obs_list(num_qubits - 1))):
        qm_back = QMachine(num_qubits)
        circuit_back(qm_back, features, wires, weights, i)
        measure = SparseHamiltonian(obs = {"observables": o, "wires":(num_qubits - 1,)})
        if i == 0:
            result_back = measure(qm_back)
        else:
            result_back = tensor.concatenate([result_back, measure(qm_back)])

    result_front = process_tensor(result_front, 0, 1)
    result_back = process_tensor(result_back, 1, 0)
    result_tensor = tensor.sums(tensor.mul(result_front, result_back), axis=1)
    return result_tensor
```

Listing 4: 前向函数

本文的模型训练保存在文件 train.py 中，导入 QModel 类搭建模型，输入图片进行归一化后作为前向函数的输入。前向函数输出的结果将落在区间 $(0, 1)$ 内部，我们认为大于 0.5 的输出将被归类为 1，小于等于 0.5 的输出将被归类到 0，对输出结果直接采用二分类交叉熵作为损失函数。我们的模型完全在 vqnet 上搭建，实现梯度回传后通过 Adam 优化器对参数进行优化。每次迭代后，我们记录下模型在测试数据集上的分类准确率，并记录下直至当前最高的准确率及其对应的模型参数，保存至文件 weights.pt 中。对应的代码如下，

```
n_train = 5000
n_test = 1000
n_epochs = 100
batch_size = 32
```

```

def train(n_train, n_epochs, batch_size):
    model = QModel()
    optimizer = Adam(model.parameters(), lr=0.1)

    train_data = np.load("train_data.npy").reshape(-1, 25)
    test_data = np.load("test_data.npy").reshape(-1, 25)
    train_labels = np.load("train_label.npy")[:,0].flatten()
    test_labels = np.load("test_label.npy")[:,0].flatten()
    train_data = train_data / np.linalg.norm(train_data, axis=1).reshape((-1, 1))
    test_data = test_data / np.linalg.norm(test_data, axis=1).reshape((-1, 1))
    train_data = train_data[:n_train]
    test_data = test_data[:n_test]
    train_labels = train_labels[:n_train]
    test_labels = test_labels[:n_test]
    train_cost_epochs, train_acc_epochs, test_acc_epochs = [], [], []
    max_acc = 0

    for _ in range(n_epochs):
        model.train()
        for x_train, y_train in dataloader(train_data, train_labels, batch_size, True):
            x_train, y_train = tensor.to_tensor(x_train), tensor.to_tensor(y_train)
            optimizer.zero_grad()

            train_out = model(x_train)
            train_cost = -y_train * tensor.log(train_out) - (1 - y_train) * tensor.log(1 - train_out)
            train_cost.backward()
            optimizer.step()
            train_pred = tensor.greater(train_out, QTensor(0.5))
            train_acc = tensor.equal(train_pred, y_train)

            if len(train_acc_epochs) == 0:
                train_acc_epoch = train_acc.to_numpy()
                train_cost_epoch = train_cost.to_numpy()
            else:
                train_acc_epoch = np.concatenate((train_acc_epoch, train_acc.to_numpy()))
                train_cost_epoch = np.concatenate((train_cost_epoch, train_cost.to_numpy()))
            train_acc_epochs.append(np.mean(train_acc_epoch))
            train_cost_epochs.append(np.mean(train_cost_epoch))

        model.eval()
        for x_test, y_test in dataloader(test_data, test_labels, batch_size, False):
            x_test, y_test = tensor.to_tensor(x_test), tensor.to_tensor(y_test)

            test_out = model(x_test)
            test_pred = tensor.greater(test_out, QTensor(0.5))
            test_acc = tensor.equal(test_pred, y_test)

            if len(test_acc_epochs) == 0:
                test_acc_epoch = test_acc.to_numpy()
            else:
                test_acc_epoch = np.concatenate((test_acc_epoch, test_acc.to_numpy()))
            test_acc_epochs.append(np.mean(test_acc_epoch))

        final_acc = np.mean(test_acc_epoch)
        if final_acc > max_acc:
            max_acc = final_acc
            save_parameters(model.state_dict(), "weights.pt")

train(n_train, n_epochs, batch_size)
```

Listing 5: 模型训练

最后，本文的模型测试保存在文件 eval.py 中。创建的模型读取 weights.pt 中的训练后参数，对输入的测试数据进行同样的归一化操作，模型分类后记录下最终在所有测试数据下的分类准确率。代码如下，

```
n_test = 1000
batch_size = 32

def eval(n_test, batch_size):
    model = QModel()
    model_para = load_parameters("weights.pt")
    model.load_state_dict(model_para)

    test_data = np.load("test_data.npy").reshape(-1, 25)
    test_labels = np.load("test_label.npy")[:,0].flatten()
    test_data = test_data / np.linalg.norm(test_data, axis=1).reshape((-1, 1))
    test_data = test_data[:n_test]
    test_labels = test_labels[:n_test]
    test_acc_epochs = []

    model.eval()
    for x_test, y_test in dataloader(test_data, test_labels, batch_size, False):
        x_test, y_test = tensor.to_tensor(x_test), tensor.to_tensor(y_test)

        test_out = model(x_test)
        test_pred = tensor.greater(test_out, QTensor(0.5))
        test_acc = tensor.equal(test_pred, y_test)

        if len(test_acc_epochs) == 0:
            test_acc_epoch = test_acc.to_numpy()
        else:
            test_acc_epoch = np.concatenate((test_acc_epoch, test_acc.to_numpy()))
        test_acc_epochs.append(np.mean(test_acc_epoch))

    print("Acc: ")
    print(np.mean(test_acc_epoch))

eval(n_test, batch_size)
```

Listing 6: 模型测试

参考文献

[1] Tianyi Peng, Aram W Harrow, Maris Ozols, and Xiaodi Wu. Simulating large quantum circuits on a small quantum computer. *Physical review letters*, 125(15):150504, 2020.

[2] 王升斌, 窦猛汉, 吴玉椿, 郭国平, 郭光灿, et al. 分布式量子计算研究进展. *Chinese Journal of Quantum Electronics*, 41(1):1–25, 2024.

[3] Michael Horodecki, Peter W Shor, and Mary Beth Ruskai. Entanglement breaking channels. *Reviews in Mathematical Physics*, 15(06):629–641, 2003.

[4] Aram W Harrow and Angus Lowe. Optimal quantum circuit cuts with application to clustered hamiltonian simulation. *arXiv preprint arXiv:2403.01018*, 2024.

[5] Daniel T Chen, Ethan H Hansen, Xinpeng Li, Vinooth Kulkarni, Vipin Chaudhary, Bin Ren, Qiang Guan, Sanmukh Kuppannagari, Ji Liu, and Shuai Xu. Efficient quantum circuit cutting by neglecting basis elements. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 517–523. IEEE, 2023.