TensorCircuit:NISQ时代的量子软件框架

TensorCircuit: a Quantum Software Framework for the NISQ Era

&

目录

1

量子线路模拟参考文献: 量子电路模拟 TensorCircuit: a Quantum Software Framework for the NISQ Era

1.1. 概述

TensorCircuit 是一个基于张量网络收缩的开源量子电路仿真器，旨在提高速度、灵活性和代码效率。纯粹用 Python 编写，并建立在行业标准的机器学习框架之上，TensorCircuit 支持自动分化、及时编译、矢量并行和硬件加速。这些功能使 TensorCircuit 可以模拟比现有模拟器更大、更复杂的量子电路，特别适合基于参数化量子电路的变异算法。与其他常见的量子软件相比，TensorCircuit 使各种量子模拟任务的速度提高了几个数量级，并能以适度的电路深度和低维连接性模拟多达 600 个量子比特。凭借其时间和空间效率、灵活和可扩展的架构以及紧凑、用户友好的 API，TensorCircuit 的建立是为了促进嘈杂的中尺度量子（NISQ）时代的量子算法的设计、仿真和分析。

1.1.1. 简介

近年来，用于模拟量子计算机的开源和专有软件 [1] 不断增加。虽然各软件包的特点和功能各不相同，但从整体上看，用户现在有许多高质量的选择，可用于构建和模拟量子电路。然而，为了加深我们对量子算法性能的理解，研究人员越来越需要模拟更大、更复杂的量子电路，并优化可能包含大量可调整参数的量子电路。尽管现有的量子软件很有前途，但在运行大规模、复杂的模拟时仍有许多挑战。这里我们介绍 TensorCircuit，一个新的基于张量网络的开源量子电路模拟器，为解决这些挑战而建立。TensorCircuit 是用 Python 编写的，并为速度、灵活性和易用性而设计，它建立在一些业界领先的库之上。通过 TensorFlow[2]、JAX[3] 和 PyTorch[4] 机器学习库，为自动区分、及时编译、矢量并行和硬件加速提供了方便的访问。最先进的 cotengra[5, 6] 包实现了快速张

量网络收缩，它还为用户提供了对张量网络收缩过程的可定制控制。这些功能使参数化的量子电路得到有效的优化，允许对更复杂的情况进行建模。此外，TensorCircuit 的语法旨在允许复杂的任务以最小的代码量来实现，节省了编码和仿真的时间。

1.1.2. 模拟量子电路的挑战

由于能够运行大规模量子算法的完全容错的量子计算机可能还需要很多年的时间，相当多的研究工作已经被用于研究近期内量子优势的前景。一些针对噪声中等规模量子（NISQ）[7] 量子计算机的算法旨在利用经典计算能力来补充量子计算机，而量子计算机可能只有有限数量的易错量子比特受到控制。特别是，混合量子-经典算法 [8, 9]，如变异量子解算器（VQE）[10] 和量子近似优化算法（QAOA）[11] 是围绕参数化量子电路（PQC）的概念。这些电路包含具有可调整参数的量子门（例如，具有可变旋转角度的单量子门），被嵌入到一个经典的优化循环中。通过优化电路中的参数值，人们旨在推动量子电路的输出状态向给定问题的解决方案发展。

在一个典型的 VQE 例子中，人们希望找到具有哈密顿 H 的量子系统的基态能量 E_0。PQC 的输出是一个量子态 $|\psi(\theta)\rangle$，其中 θ 是一个可调整参数的矢量。这个试验状态–被称为" 答案"–形成对基态波函数的猜测。通过对 $|\psi(\theta)\rangle$ 进行适当的测量，可以估计出预期能量 $\langle H\rangle_\theta = \langle\psi(\theta)|H|\psi(\theta)\rangle$。通过最小化

$$H_\theta$$

与参数 θ 的关系，可以得到基态能量的上界估计。

$$E_0 \leq \min_\theta \langle H\rangle_{\theta.} \tag{1.1}$$

1.1 有一些问题影响了这种方法的效果和效率。首先，为了准确估计 E_0，对于某些 θ 值来说，定理 $|\psi(\theta)\rangle$ 应该是对真实基态的良好近似。是否如此，取决于问题的性质和构建答案的 PQC 的复杂性。一方面，简单的答案–例如 [12] 的" 硬件高效" 答案–可能更容易在真实或模拟的量子计算机上实现，但可能不够准确（例如，由于缺乏物理相关的结构或短深度），或者受到其他问题的影响，例如参数空间中的所谓贫瘠高原 [13] 或能量景观中的局部最小值 [14]。另一方面，更复杂的答案，如为量子化学问题提出的单元耦合集群（UCC）[15] 方法，可能需要量子电路的复杂性和深度超过目前可以实现或模拟的，相关的电路必须被截断或简化。如果能够模拟更大更深的量子电路，就可以对更大类的答案进行系统研究。

其次，对于一个给定的答案，$\langle H\rangle_\theta$ 的评估可能是一个复杂的过程。例如，考虑 H 是一个 n 比特哈密顿的情况，它可以被表达为保利算子的张量积的加权和，即。

$$H = \sum_{j=1}^{K} a_j P_j \tag{1.2}$$

其中 a_j 是实系数，每个保利串[1] P_j 的形式为 $P_j = \sigma i_1 \oplus \sigma i_2 \oplus \cdots \sigma i_n$，而 a_j 是单比特保利算子或同一性。在最直接的方法中，估计 $\langle H\rangle_\theta$ 是通过首先估计每个项 $\langle P_j\rangle_\theta$ 的期望值，然后把这些单独的贡献加在一起来进行的。如果 Hamilonian 由许多 Pauli 项组成，那么加速评估公式1.2的方法–例如通过利用 H 的有效表示（例如作为稀疏矩阵或矩阵乘积算子（MPO）[16]），或者能够并行计算多个项，可以对计算时间产生很大影响。

[1] 一个-qubit Pauli 可以用一个由 ['I', 'X', 'Y', 'Z'] 的字符组成的字符串来表示，也可以选择相位系数。例如。XYZ 或'-iZIZ' 。在字符串表示中，qubit-0 对应于最右边的 Pauli 字符，而 qubit-对应于最左边的 Pauli 字符。
维基百科解释 谷歌 Quantum AI 使用说明 IBM Qiskit 使用说明

第三，公式1.1中的优化问题，一般来说，是非凸的。因此，寻找全局最小值一般来说在计算上是难以实现的。然而，如果潜在的复杂表达式的梯度可以被有效地评估，我们可以使用梯度下降方法来寻找局部最小值，这可能会产生一个体面的近似解。通过从一些不同的位置（即不同的 θ 值）初始化优化，可以得到多个局部最小值，提高其中一个给出好的解决方案的机会。如果这些多个解决方案可以并行优化，就可以实现巨大的效率提升。

1.1.3. 机器学习库

上一节讨论的挑战在很大程度上与机器学习，特别是深度学习 [17] 中面临的问题重叠。幸运的是，解决日益复杂的机器学习问题和处理规模越来越大的数据集的需要，导致了令人印象深刻的软件的发展，现在有许多框架，将强大的功能和易于使用的语法结合起来。特别是:

快速梯度: 所有高级机器学习包的核心是执行自动分化（AD）的能力 [18, 19]，其中用于训练神经网络的反向传播算法是一个特殊的例子。AD 能够有效地计算代码中定义的函数的梯度，对许多机器学习模型的优化至关重要。

JIT: 及时编译是一种在程序执行过程中对代码的某些部分进行编译的方式。对于像 Python 这样的解释型语言，"jitted " 函数可以带来巨大的性能提升，执行 jitted 函数所需的时间往往只是该函数被解释时的一小部分。虽然在第一次调用函数时，可能会有编译所需的开销（staging time），但如果该函数随后被多次调用，这一成本可以忽略不计。

矢量化 (VMAP): 这个功能允许在多个点上并行评估一个函数，与使用天真的 for 循环相比，速度明显提高。在机器学习中，这允许人们，例如，同时对成批的数据进行计算。

硬件加速: 对于复杂的机器学习模型，在多个 CPU、GPU 和 TPU 上执行代码的能力对于在合理的时间内完成训练可能是必要的。

这些强大的功能也有利于模拟量子计算机，特别适合于变量量子算法。在这里，通过自动微分的快速梯度评估给了模拟器一个固有的优势，而真实的量子计算机必须以不太直接的方式来估计梯度；例如，通过对各种输入参数选择的输出进行采样并计算有限差分 [20, 21] 。矢量化可以（除其他外）用于评估具有多个参数选择的 PQC 电路，或者同时计算多个保利串的期望值，而 JIT 和硬件加速则可以进一步节省时间。

此外，人们对通过量子计算解决机器学习问题，以及结合经典和量子机器学习算法的兴趣越来越大。经典 ML 框架和量子电路模拟器之间更好的整合可以促进这两方面的发展，并且可以从两者的无缝整合中获得巨大价值。

1.1.4. 量子软件的下一阶段

虽然量子软件在过去几年中取得了很大的进步–Qiskit[22]、Cirq[23]、HiQ[24]、Q[25] 和 qulacs[26] 等软件包都提供了强大的功能和特性，但高效的量子模拟软件辅以最先进的机器学习的能力和特性，仍然可以获得巨大的优势。最近，新一代的量子软件开始出现，Python 生态系统中的 TensorFlow Quantum[27]、Pennylane[28]、Paddle Quantum[29] 和 MindQuantum[30] 在这里取得了进展，并不同程度地提供了这些功能。然而，到目前为止，这些都没有完全结合上一节的所有关键功能和快速量子电路仿真。TensorCircuit 的设计就是为了填补这一空白，给用户提供一个更快、更灵活、更方便的方式来模拟量子电路。

1.2. TENSORCIRCUIT 概述

	AD	JIT	VMAP	GPU
TensorFlow	✓✓	✓✓✓	✓✓	✓✓✓
JAX	✓✓✓	✓✓✓	✓✓✓	✓✓✓
PyTorch	✓✓	✓	✓	✓✓✓
NumPy	·	·	·	·

图 1.1: 后端 TensorFlow、JAX 和 PyTorch 与中间件 TENSORCIRCUIT

我们开发 TensorCircuit 的目的是提供第一个高效的、基于张量网络的量子模拟器，它与现代机器学习框架的主要特征完全兼容，特别是自动分化、矢量并行和即时编译的编程范式。这些功能是通过一些流行的机器学习后端提供的，在撰写本文时，这些后端是 TensorFlow、JAX 和 PyTorch（见表1.1）。

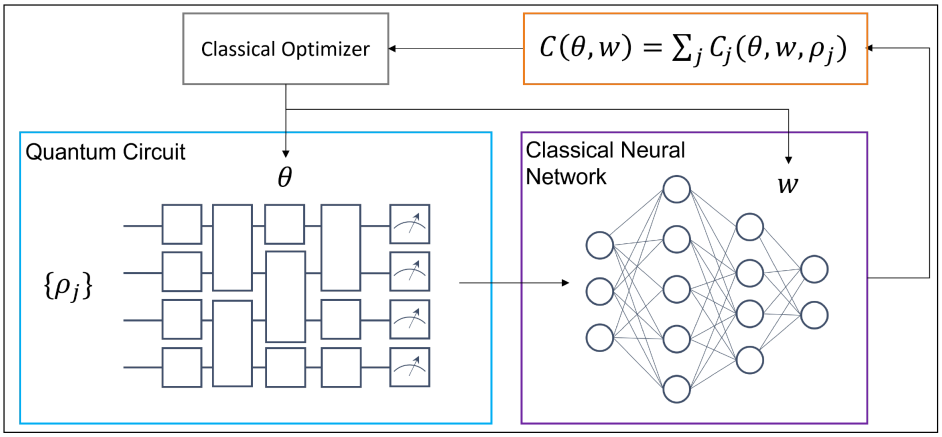

图 1.2: 一个一般的混合量子-经典神经网络，其中成本函数 C 在一批输入状态 ρj 上求和，并依赖于量子电路和经典神经网络的参数。通过使用经典优化器，两个网络的参数可以被反复改进。与经典机器学习后端集成，可以在 TensorCircuit 中无缝模拟整个端到端过程，vmap、jit 和自动分化使整个优化过程的效率提高。

与这些后端的集成允许一般的混合量子-经典模型，其中参数化的量子电路的输出可以被输入到经典的神经网络中，或者反之，并进行无缝模拟（见图1.2）。这种整合也是量子机器学习相关研究的关键 [31]。目前，基于张量网络的量子模拟器的选择非常有限，大多数流行的软件都是使用状态矢量模拟器。状态矢量模拟器受到内存的强烈限制，因为波函数振幅是完全存储的，因此在模拟具有较大数量的量子比特的电路时可能会遇到困难。另一方面，具有大量（可能是嘈杂的）量子比特但电路深度相对较短的量子电路–例如那些对应于 NISQ 设备的量子比特具有较短的相干时间–属于张量网络模拟器的适用区域。

1.2.1. 设计理念

在高效的基于张量网络的仿真引擎支持下，与现代机器学习范式无缝集成，TensorCircuit 的设计从一开始就以下列原则为基础。**速度**:TensorCircuit 使用张量网络收缩来模拟量子电路，并与最先进的第三方收缩引擎兼容。相比之下，目前大多数流行的量子模拟器是基于状态矢量的。张量收缩框架允许 TensorCircuit ，在许多情况下。

与其他模拟器相比，TensorCircuit 可以在时间和空间上提高模拟量子电路的效率。JIT、AD、VMAP 和 GPU 支持也都可以在许多情况下提供明显的加速（通常是几个数量级）。

灵活性:TensorCircuit 的设计是与后端无关的，因此可以在任何机器学习后端之间轻松切换，而不会改变语法或功能。通过不同的 ML 后端，可以灵活地模拟混合量子-经典神经网络，在 CPU、GPU 和 TPU 上运行代码，并在 32 位和 64 位精度数据类型之间切换。

代码效率: 现代机器学习框架，如 TensorFlow、PyTorch 和 JAX，具有用户友好的语法，允许以最小的代码量执行强大的任务。通过 TensorCircuit，我们同样专注于一个紧凑和易于使用的 API，以提高可读性和生产力。与其他流行的量子模拟软件相比，TensorCircuit 通常可以用少得多的代码执行类似的任务（见 tc 与 tfq 的 VQE 的例子）。此外，TensorCircuit 的后端不可知的语法使其很容易在 ML 框架之间切换，只需一行设置代码。

关注社区 TensorCircuit 是开源软件，我们关注代码库的可读性、可维护性和可扩展性。我们邀请量子计算社区的所有成员参与其持续发展和使用。

1.2.2. 张量网络引擎

TensorCircuit 使用张量网络形式主义来模拟量子电路，这种形式主义在计算物理学中有着悠久的历史，最近被率先用于量子电路模拟 [32] 。事实上，量子电路和张量网络的图形表示是一致的，这使得量子电路模拟与张量网络收缩的转换变得直接而简单。在这幅图中，量子电路由对应于单个量子门的低等级张量网络表示，振幅、期望值或其他标量的计算是通过收缩网络的边缘直到只剩下一个节点来进行的。边缘收缩的顺序或路径很重要，对收缩网络所需的时间和空间有很大影响 [6, 33, 34]。参见 [35]，从物理学角度对张量网络做了很好的介绍。

在最一般的情况下，与其他类型的量子模拟器一样，通过张量网络收缩来模拟量子电路所需的时间与量子比特的数量成指数关系。然而，在特殊情况下–包括许多具有实际意义的情况–张量网络收缩可以提供显著的优势，因为它避免了困扰全状态模拟器的内存瓶颈，到目前为止，最大规模的量子计算模拟，如量子超限实验中使用的随机电路的模拟 [36, 37] 都是通过这种方法进行的 [38-43]。

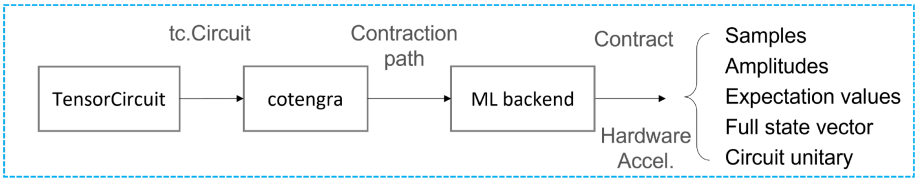

图 1.3: 图 1 中的量子电路组件示意图。在 TensorCircuit 中，构成量子电路的门被包含在 tc.Circuit 对象中。电路输出的计算分两步执行。首先，张量收缩路径由收缩引擎确定，例如 cotengra，后端负责实际收缩。

TensorCircuit 中使用的张量网络数据结构由 TensorNetwork[44] 软件包提供。此外，TensorCircuit 可以利用最先进的外部 Python 包，如 cotengra 来选择有效的收缩路径，然后由用户选择的机器学习后端通过 einsum 和 matmul 进行收缩（见图1.5）。

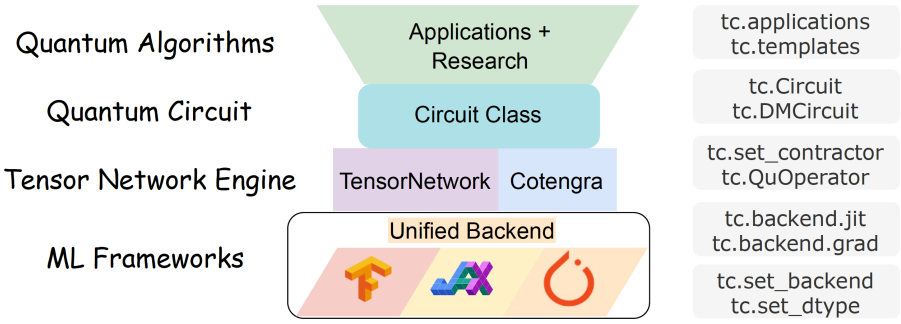

图 1.4: TensorCircuit 的软件架构。抽象层显示在左边，而代表性的 API 显示在右边。在底部，不同的机器学习框架，如 TensorFlow、JAX 和 PyTorch，通过一组与后端无关的 API 被统一起来。这些后端与张量网络引擎一起，实现了高效的量子电路模拟和量子-经典混合算法和应用的实施。

架构量子电路模拟的张量网络引擎是建立在各种机器学习框架之上的，中间有一个抽象层，统一了不同的后端。在应用层，TensorCircuit 还包括基于我们最新研究 [45-49] 的各种高级量子算法。整体软件架构如图 3 所示。

1.2.3. 安装和贡献于 TensorCircuit

TensorCircuitis open-sourced under the Apache 2.0 license. The software is available on the Python Package Index (PyPI) and can be installed using the command pip install tensorcircuit.

TensorCircuit 是在 Apache 2.0 许可下开源的。该软件可在 Python 软件包索引（PyPI）上找到，可以使用 pip install tensorcircuit 命令进行安装。

The development of TensorCircuit is open-sourced and centered on GitHub: welcome all members of the quantum community to contribute, whether it is • Answering questions on the discussion page or issues page. • Raising issues such as bug reports or feature requests on the issues page. • Improving the documentation (docstrings/tutorials) by pull requests. • Contributing to the codebase by pull requests. For more details, please refer to the contribution guide: contribution.html .

TensorCircuit 的开发是开源的，以 GitHub 为中心：欢迎量子社区的所有成员作出贡献，无论是

- 在讨论页或问题页上回答问题。
- 在问题页上提出问题，如错误报告或功能请求。通过拉动请求改进文档（docstrings/tutorials）。
- 通过拉动请求对代码库进行贡献。

更多细节，请参考贡献指南：contribution.html 。

1.3. 电路和闸门

Jupyter notebook: 3-circuits-gates.ipynb

在 TensorCircuit 中，通过 tc.Circuit(n)API 创建 n 个量子比特上的量子电路–它支持通过蒙特卡洛轨迹方法进行无噪声和噪声模拟。这里我们展示了如何创建基本电路，对其应用门，并计算各种输出。

1.3.1. 预备工作

在本文的其余部分，我们假设我们同时导入了 TensorCircuit 和 NumPy 作为

Listing 1.1: 导入 TensorCircuit 和 NumPy

```
import tensorcircuit as tc
import numpy as np
```

此外，我们假设已经设置了一个 TensorCircuit 的后端，如：

Listing 1.2: 导入 TensorCircuit 和 NumPy

```
K = tc.set_backend("tensorflow")
```

和代码片段中出现的符号 K（例如 K.real()）指的是该后端。set_backend 方法的其他选项是”jax”、”pytorch ” 和”numpy”（默认后端）。

在 TensorCircuit 中，量子比特从 0 开始编号，多量子比特寄存器从左边的第 4 个量子比特开始编号，例如：$|0\rangle_{q0}|1\rangle_{q1}|0\rangle_{q1}$。除非需要，我们将省略下标，并使用紧凑的符号，例如：$|010\rangle$ 表示多比特状态。X,Y,Z 表示标准的单量子位保利算子，用下标表示，例如 X_3，以明确哪个量子位被作用。相对于一个状态 $|\psi$ 的算子的期望值，例如 $\langle\psi|Z\oplus|\oplus X|\psi\rangle$ 将被简记为 $\langle Z_0X_2\rangle$。除非说明，期望值总是相对于一个给定的量子电路的输出状态而言的。如果该电路由一组角度 ψ 参数化，那么与参数有关的期望值可以用 $\langle\cdot\rangle$ 来表示。在 TensorCircuit 中，默认的运行时数据类型是 complex64，但如果需要更高的精度，可以按如下方式设置。

Listing 1.3: 更高的精度

```
tc.set_dtype("complex128")
```

1.3.2. 基本电路和输出

Consider the following two-qubit quantum circuit consisting of a Hadamard gate on qubit q0, a CNOT on qubits q0 and q1, and a single qubit rotation RX (0.2) of qubit q1 by angle 0.2 about the X axis (see Figure 4). Qubits are numbered from 0 with q0 on the top row. This circuit can be implemented in TensorCircuit as

考虑下面的双量子电路，包括在量子位 q0 上的哈达玛德门，在量子位 q0 和 q1 上的 CNOT，以及量子位 q1 的单量子位旋转 $R_X(0.2)$，围绕 X 轴的角度为 0.2（见图 4）。量子位从 0 开始编号，$q0$ 在最上面一行。这个电路可以在 TensorCircuit 中实现为

Listing 1.4: 围绕 X 轴的角度为 0.2 的状态

```
c = tc.Circuit(2)
c.h(0)
c.cnot(0, 1)
c.rx(1, theta=0.2)
```

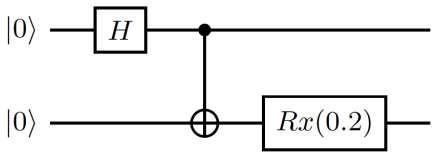

图 1.5: 一个由 Hadamard、CNOT 和单量级 RX 旋转组成的双量级电路。

电路需要用 Latex 重画

由此，可以计算出各种输出。

基本输出. 完整的波函数可以通过以下方式获得

Listing 1.5: 基本输出

```
c.state()
```

这将输出一个数组 $[a_{00}, a_{01}, a_{10}, a_{11}]$，对应于 $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ 基态的振幅。完整的波函数也可以用来生成量子比特子集的还原密度矩阵，例如;

Listing 1.6: 生成量子比特子集的还原密度矩阵

```
# reduced density matrix for qubit 1
s = c.state()
tc.quantum.reduced_density_matrix(s, cut=[0]) # cut: list of qubit indices to trace out
```

单个基向量的振幅是通过向振幅函数传递相应的位串值来计算的。例如，$|0\rangle$ 基向量的振幅是通过以下方式计算的

Listing 1.7: 单个基向量输出

```
amplitude("10")
```

对应于整个量子电路的单元矩阵也可以被输出。

Listing 1.8: 对于整个量子电路的单元矩阵输出

```
c.matrix()
```

测量和样本

对应于 Z-测量的随机样本，即在 $\{|0\rangle, |1\rangle\}$ 的基础上，对所有量子比特的测量可以用以下方法产生

Listing 1.9: 对于整个量子电路的单元矩阵输出

```
 c.sample()
```

它将输出一个 (bitstring,probability) 元组，包括一个二进制字符串，对应于对所有量子比特进行 Z 测量的测量结果以及获得该结果的相关概率。对一个子集的量子比特的 Z 测量可以用测量命令来执行

Listing 1.10: 子集的量子比特的 Z 测量输出

```
 # return (outcome, probability) of measuring qubit 0 in Z basis
print(c.measure(0, with_prob=True))
```

对于多个量子位的测量，只需提供一个要测量的指数列表，例如，如果 c 是一个 4 量子位电路，对量子位 1、3 的测量可以通过以下方式完成

Listing 1.11: c 是一个 4 量子位电路，对量子位 1、3 的测量

```
c.measure(1, 3, with_prob=True)
```

请注意，为了计算这些结果，测量门不需要明确地添加到电路中，而且测量和采样命令不会使电路输出状态崩溃。测量门可以被添加和使用，例如，当门必须以电路中间的测量结果为条件来应用时（见第六节 A）。

期望值。诸如 $\langle X_0\rangle, \langle X_1 + Z_1\rangle$ 或 $\langle Z_0 Z_1\rangle$ 这样的期望值可以通过电路对象的期望方法计算，其中运算符通过门对象或简单的数组定义。 什么意思?

Listing 1.12: 运算符通过门对象或简单的数组定义

```
c.expectation([tc.gates.x(), [0]]) # <X0>
c.expectation([tc.gates.x() + tc.gates.z(), [1]]) # <X1 + Z1>
c.expectation([tc.gates.z(), [0]], [tc.gates.z(), [1]]) # <Z0 Z1>
```

和用户定义的运算符的期望值也可以通过提供相应的矩阵元素阵列来计算。例如，运算符 $2X + 3Z$ 可以用矩阵表示为

$$\begin{pmatrix} 3 & 2 \\ 2 & -3 \end{pmatrix}$$

并实现（假设观测点是在 0 号量子位上测量）为

Listing 1.13: 在 0 号量子位上测量

```
c.expectation([np.array([[3, 2], [2, -3]]), [0]])
```

泡利串的期望值.

虽然保利算子的乘积的期望值，例如 ${}_0X_1$ 可以使用上述的 c.expectation 来计算，但 TensorCircuit 提供了另一种计算这种表达式的方法，对于较长的保利串可能更方便。

Listing 1.14: 较长的保利串表达

```
c.expectation_ps(x=[1], z=[0])
```

而更长的保利串也同样可以通过提供与 X, Y, Z 算子所作用的量子比特相对应的索引列表来计算。例如，对于一个 $n = 5$ 的量子比特电路，期望值 ${}_0X_1Z_2Y_4$ 可计算为

Listing 1.15: 对于一个 n

```
expectation_ps(x=[1], y=[4], z=[0, 2])
```

标准量子门。除了我们目前遇到的 CNOT、Hadamard 和 RX 门之外，TensorCircuit 还为各种常见的量子门提供支持。全部门的列表可以通过查询找到

Listing 1.16: 查询全部门的列表

```
tc.Circuit.sgates # non-parameterized gates
tc.Circuit.vgates # parameterized gates
```

对应于某个门的矩阵，例如 Hadamard h 门，可以通过以下方式访问

Listing 1.17: Hadamard h 门

```
tc.gates.matrix_for_gate(tc.gates.h())
```

任意的单元门

用户定义的单元门可以通过将其矩阵元素指定为数组来实现。例如，单元门

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} -$$

也可以通过调用 c.s() 直接添加 - 0i 可以被实现为

Listing 1.18: 用户定义的单元门

```
c.unitary(0, unitary = np.array([[1,0],[0,1j]]), name='S')
```

其中，可选的名称参数指定了当电路输出到 LATEX 时如何显示这个门。

指数门

形式为 $e^{i\psi G}$ 的门，其中矩阵 G 满足 $G^2 = I$，允许通过 exp1 命令快速实现，例如，作用于量子位 0 和 1 的双量子位门 $e^{i\psi Z\oplus Z}$

Listing 1.19: 双量子位门

```
c.exp1(0, 1, theta=0.2, unitary=tc.gates._zz_matrix)
```

其中 tc.gates._zz_matrix 创建了一个对应于矩阵的 numpy 数组。

$$Z \oplus Z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \tag{1.4}$$

一般的指数门，其中 $G^2 \neq I$ 可以通过 exp 命令实现。

Listing 1.20: 指数门

```
c.exp(0, theta=0.2, unitary=np.array([[2, 0],[0, 1]]))
```

非单片机门

TensorCircuit 也支持非单片门的应用，方法是提供一个非单片矩阵作为 c.unitary 的参数，例如:

Listing 1.21: 非单片机门

```
c.unitary(0, unitary=np.array([[1,2],[2,3]]), name='non_unitary')
```

请注意，非单片机门将导致输出状态不再被归一化，因为归一化往往是不必要的，并且需要额外的计算时间。

1.3.3. 指定输入状态和组成电路

默认情况下，量子电路被应用于初始的全零乘积状态。可以通过向 tc.Circuit 的可选输入参数传递一个包含输入状态振幅的数组来设置任意的初始状态。

例如，最大纠缠状态 $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$ 可以被输入如下。

Listing 1.22: 最大纠缠状态 $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$

```
c1 = tc.Circuit(2, inputs=np.array([1, 0, 0, 1] / np.sqrt(2)))
```

矩阵产品状态（MPS）形式的输入状态也可以通过 tc.Circuit 的可选 mps_inputs 参数输入。详见第六节 D。

作用于相同数量量子比特的电路可以通过 c.append() 或 c.predend() 命令组合在一起。有了上面定义的 c1，我们可以创建一个新的电路 c2，然后将它们组合在一起。

Listing 1.23: 组合 $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$

```
c2 = tc.Circuit(2)
c2.cnot(0, 1)

c3 = c1.append(c2)
```

这导致了电路 C_3，它相当于首先应用 C_1，然后是 C_2。

1.3.4. 电路转换和可视化

tc.Circuit 对象可以与 Qiskit QuantumCircuit 对象进行转换。输出到 Qiskit 的方法是

Listing 1.24: 对象进行转换

```
c.to_qiskit()
```

然后产生的 QuantumCircuit 对象可以被编译并在兼容的物理量子处理器和模拟器上运行。反之，从 Qiskit 导入一个 QuantumCircuit 对象是通过

Listing 1.25: 从 Qiskit 导入一个 QuantumCircuit 对象

```
c = tc.Circuit.from_qiskit(QuantumCircuit)
```

有两种方法可以将 TensorCircuit 中生成的量子电路可视化。第一种是使用

Listing 1.26: 可视化输出

```
print(c.tex())
```

其中，使用 LATEX quantikz 软件包 [50] 输出绘制相关量子电路的代码。

第二种方法是使用绘制函数。

Listing 1.27: 绘制函数

```
c.draw()
```

的捷径，它是

Listing 1.28: 绘制函数长

```
qc = c.to_qiskit()
qc.draw()
```

支撑电路转换和可视化工具的是 TensorCircuitCircuit 对象的量子中间表示（IR），它可以通过以下方式获得

Listing 1.29: 绘制函数长

```
c.to_qir()
```

1.4. 梯度、优化和变异算法

TensorCircuit 旨在使参数化量子门的优化变得简单、快速和方便。考虑一个作用于 n 量子比特的变量电路，由 k 层组成，其中每一层包括相邻量子比特之间的参数化 $e^{iX\oplus X}$ 门，然后是一串单量子比特的参数化 Z 和 X 旋转。

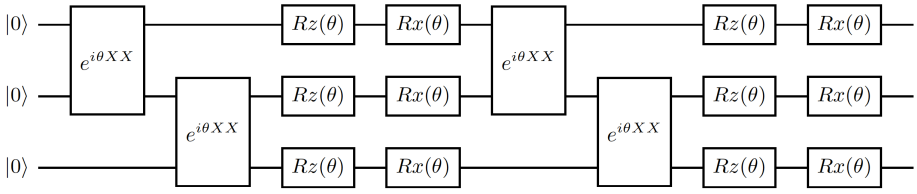

图 1.6: 一个参数化、分层的量子电路。每个门都依赖于一个单独的参数，这里都示意性地表示为ψ□

我们现在展示如何在 TensorCircuit 中实现这样的电路，以及如何使用机器学习后端之一来轻松有效地计算成本函数和梯度。一般 n、k 和一组参数的电路可以定义如下。

Listing 1.30: n、k 和一组参数的电路

```
def qcircuit(n, k, params):
        c = tc.Circuit(n) for j in range(k):
        for i in range(n - 1):
                c.exp1(
                )
        for i in range(n):
return c
```

作为一个例子，我们把 $n = 3$，$k = 2$，设置 TensorFlow 作为我们的后端，并定义一个能量成本函数来最小化

$$E = \langle X_0X_1\rangle_\theta + \langle X_1X_2\rangle_\theta \tag{1.5}$$

Listing 1.31: E

```
n=3
k=2

K = tc.set_backend("tensorflow")

def energy(params):
        return K.real(e)
```

K.grad 和 K.value_and_grad。利用 ML 后台对自动微分的支持，我们现在可以快速计算能量和能量的梯度（相对于参数而言）。

Listing 1.32: 快速计算能量和能量的梯度

```
energy_val_grad = K.value_and_grad(energy)
```

这创建了一个函数，给定一组参数作为输入，它同时返回能量和能量的梯度。如果只需要梯度，那么可以通过 K.grad(energy) 来计算。虽然我们可以在一组参数上直接运行上述代码，但如果要对能量进行多次评估，使用该函数的即时编译版本可以大大节省时间。

Listing 1.33: 函数的即时编译版本可以大大节省时间

```
energy_val_grad_jit = K.jit(energy_val_grad)
```

使用 K.jit，能量和梯度的初始评估可能需要更长的时间，但随后的评估将明显比非 jit 的代码快。我们建议一直使用 jit，只要函数是' 张量输入，张量输出'，我们努力使电路模拟器的所有方面与 JIT 兼容。

1.4.1. 通过 ML 后端进行优化

有了能量函数和梯度，参数的优化就很简单了。下面是一个如何通过随机梯度下降进行优化的例子。

Listing 1.34: 通过随机梯度下降进行优化的例子

```
learning_rate = 2e-2
opt = K.optimizer(tf.keras.optimizers.SGD(learning_rate))

def grad_descent(params , i):
        params = opt.update(grad, params)
        if i % 10 == 0:
params = K.implicit_randn(k * (3 * n - 1))
for i in range(200):
        params = grad_descent(params , i)
```

虽然这个例子是用 TensorFlow 后端完成的，但切换到 JAX 也可以很容易做到。只需要使用 JAX 优化库 optax[51] 来重新定义优化器 opt。

Listing 1.35: 通过 JAX 随机梯度下降进行优化的例子

```
import optax
opt = tc.backend.optimizer(optax.sgd(learning_rate)
```

然后，通过以下方式选择 JAX 作为后端

Listing 1.36: JAX 后端

```
K = tc.set_backend("jax")
```

并完全按照上述方法执行梯度下降。注意，如果没有明确设置后端，TensorCircuit 默认使用 NumPy 作为后端，这不允许自动区分。

1.4.2. 通过 SciPy 进行优化

使用机器学习后端进行优化的另一个选择是使用 SciPy。这可以通过 scipy_interface API 调用完成，并允许使用基于梯度（如 BFGS）和非梯度（如 COBYLA）的优化器，这些优化器无法通过 ML 后端获得。

Listing 1.37: SciPy 优化

```
import scipy.optimize as optimize

f_scipy = tc.interfaces.scipy_interface(energy, shape=[k * (3 * n - 1)], jit=True)
params = K.implicit_randn(k * (3 * n - 1))
```

上面的第一行指定了提供给要最小化的函数的参数形状，这里是能量函数。jit=True 参数自动处理了能量函数的 jitting。同样，通过向 scipy_interface 提供 gradient=False 参数，可以有效地进行无梯度优化。

Listing 1.38: SciPy 优化

```
f_scipy = tc.interfaces.scipy_interface(
        energy, shape=[k * (3 * n - 1)],
        jit=True, gradient=False
)
r = optimize.minimize(f_scipy , params , method="COBYLA")
```

1.5. 密度矩阵和混合状态演变

TensorCircuit 提供了两种模拟噪声、混合态量子演化的方法。通过使用 tc.DMCircuit(n) 提供 n 个量子比特的全密度矩阵模拟，然后在电路中加入量子操作–包括单元门以及由 Kraus 算子指定的一般量子操作。相对于通过 tc.Circuit 对 n 个量子比特进行的纯状态模拟，全密度矩阵模拟的内存消耗是两倍，因此可模拟的最大系统规模将是纯状态模拟的一半。一个内存密集度较低的选择是使用标准的 tc.Circuit(n) 对象，通过蒙特卡洛方法随机地模拟开放系统的演化。

1.5.1. 用 tc.DMCircuit 进行密度矩阵模拟

下面我们通过考虑一个单量子位上的简单电路来说明这种方法，该电路的输入为 $|0\rangle$ 状态和最大混合状态的概率混合物所对应的混合状态

$$\rho(\alpha) = \alpha|0\rangle\langle 0| + (1-\alpha)I/2 \tag{1.6}$$

这个状态然后通过一个应用 X 门的电路，接着是对应于参数为 ε_γ 的振幅阻尼通道 γ 的量子操作。这有克劳斯算子

$$k_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}, k_1 = \begin{pmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{pmatrix} \tag{1.7}$$

因此，该电路诱发了以下演变

$$\rho(\alpha) \xrightarrow{X} X\rho(\alpha)X \xrightarrow{\varepsilon_\gamma} \sum_{i=0}^{1} K_i X \rho(\alpha) X K_i^\dagger \tag{1.8}$$

为了在 TensorCircuit 中进行模拟，我们首先创建一个 tc.DMCircuit（密度矩阵电路）对象，并使用 dminputs 可选参数设置输入状态（注意，如果向 tc.DMCircuit 提供纯状态输入，应该通过 inputs 可选参数而不是 dminputs 完成）。$\rho(\alpha)$ 的矩阵形式为

$$\rho(\alpha) = \begin{pmatrix} \frac{1+\alpha}{2} & \\ & \frac{1-\alpha}{2} \end{pmatrix} \tag{1.9}$$

因此（取 $(\alpha) = 0.6$），我们初始化密度矩阵电路如下

Listing 1.39: 密度矩阵电路

```
def rho(alpha):
```

```
        return np.array([[(1 + alpha) / 2, 0], [0, (1 - alpha) / 2]])

input_state = rho (0.6)
```

添加 X 门（和其他单元门）的方法与纯状态电路的方法相同。

Listing 1.40: 添加 X 门

```
dmc.x(0)
```

为了实现一般的量子操作，如振幅阻尼通道，我们使用 General_kraus，提供相应的 Kraus 算子列表。

Listing 1.41: Kraus 算子列表

```
def amp_damp_kraus(gamma):
        K0 = np.array([[1, 0], [0, np.sqrt(1 - gamma)]])
        K1 = np.array([[0, np.sqrt(gamma)], [0, 0]])
        return K0 , K1

K0, K1 = amp_damp_kraus(0.3)
```

而完整的密度矩阵输出可以通过以下方式返回

Listing 1.42: 完整的密度矩阵输出可返回

```
dmc.state()
```

在这个例子中，我们手动输入了振幅阻尼通道的 Kraus 算子，以说明用 tc.DMCircuit 方法实现量子通道的一般方法。事实上，TensorCircuit 包括内置的方法来返回一些常见通道的 Kraus 算子，包括振幅阻尼、去极化、相位阻尼和复位通道。例如，参数为 γ 的相位阻尼通道的 Kraus 算子

$$k_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}, k_1 = \begin{pmatrix} 0 & 0 \\ 0 & \sqrt{\gamma} \end{pmatrix} \tag{1.10}$$

可以通过调用

Listing 1.43: 调用方法

```
gamma = 0.3
K0, K1 = tc.channels.phasedampingchannel(gamma)
```

和相位阻尼通道加入到电路中，通过

Listing 1.44: 调用方法

```
dmc.general_kraus([K0, K1], 0)
```

上述操作可以通过一个 API 调用进一步简化。

Listing 1.45: 通过一个 API 调用进一步简化

```
dmc.phasedamping(0, gamma=0.3)
```

1.5.2. 用 tc.Circuit 进行蒙特卡洛模拟

蒙特卡洛方法可用于使用 tc.Circuit 而不是 tc.DMCircuit 对嘈杂的量子演化进行采样，其中混合态有效地被纯态的集合所模拟。与穴位矩阵模拟一样，量子通道 E 可以通过提供其相关的克劳斯算子 Ki 的列表添加到电路对象中。该 API 与全密度矩阵模拟相同。

Listing 1.46: 该 API 与全密度矩阵模拟相同

```
input_state = np.array([1, 1] / np.sqrt(2))
c = tc.Circuit(1, inputs=input_state)
c.general_kraus(tc.channels.phasedampingchannel(0.5), 0)
c.state()
```

但在这个框架中，作用于 $|\psi\rangle$ 的通道的输出，即

$$\varepsilon(|\psi\rangle\langle\psi|) = \sum_i K_i|\psi\rangle\langle\psi|K_i^\dagger \tag{1.11}$$

被看作是一个状态 $\frac{K_i|\psi\rangle}{\sqrt{\langle\psi|K_i^\dagger K_i|\psi\rangle}}$ 的集合，每个状态发生的概率 $p_i = \langle\psi|K_i^\dagger K_i|\psi\rangle$。因此，上述代码随机地产生了初始化为状态 $|\psi\rangle = \frac{|0\rangle+|1\rangle}{2}$ 的单一量子位的输出，并通过参数 $\gamma = 0.5$ 的相位阻尼通道。

使用 unitary_kraus 而不是 general_kraus 可以更有效地处理 Kraus 算子都是单元矩阵的通道的蒙特卡洛模拟（最多是一个常数）。例如，用 p_x,p_y,p_z 作为参数的 Kraus 算子的去极化通道:

$$K_0 = (1-p_x-p_y-p_z)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, K_1 = p_x\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, K_2 = p_y\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, K_3 = p_z\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{1.12}$$

可以通过以下方式实现

Listing 1.47: 方式实现

```
px, py, pz = 0.1, 0.2, 0.3
c.unitary_kraus(tc.channels.depolarizingchannel(px, py, pz), 0)
```

其中，在第二行中，tc.channel.depolarizingchannel(px, py, pz) 返回所需的 Kraus 算子。

随机性的外部化

General_kraus 和 unitary_kraus 的例子都是从各自的方法内部处理随机性的生成。也就是说，当 Kraus 算子的列表 $[K_0, K_1, \cdots, K_{m-1}]$ 被提供给 general_kraus 或 unitary_kraus 时，该方法将区间 $[0, 1]$ 划分为 m 个连续的区间 [0, 1] = I0 □ I1 □. . . Im-1，其中 Ii 的长度与获得结果 i 的相对概率成正比。然后从该方法中抽取 [0,1] 中的均匀随机变量 x，并根据 x 所在的区间选择结果 i。在 TensorCircuit 中，我们有完整的后端不可知的基础设施来生成和管理随机数。然而，如果我们依赖这些方法内部的随机数生成，jit、随机数和后端切换之间的相互作用往往是微妙的。详见 advance.htmlrandoms-jit-backend-agnostic-and-their-interplay。

在某些情况下，最好是先从方法外部生成随机变量，然后将生成的值传递给 general_kraus 或 unitary_kraus。这可以通过可选的状态参数来实现。

Listing 1.48: 从方法外部生成随机变量

```
px, py, pz = 0.1, 0.2, 0.3
```

```
x = K.implicit_randn()
c.unitary_kraus(tc.channels.depolarizingchannel(px, py, pz), 0, status=x)
```

This is useful, for instance, when one wishes to use vmap to batch compute multiple runs of a Monte Carlo simulation. This is illustrated in the example below, where vmap is used to compute 10 runs of the simulation in parallel:

这很有用，例如，当人们希望使用 vmap 来批量计算蒙特卡洛模拟的多次运行时。下面的例子说明了这一点，vmap 被用来并行计算 10 次模拟运行。

Listing 1.49: vmap 被用来并行计算 10 次模拟运行

```
def f(x):
        c = tc.Circuit(1)
        c.h(0)
        c.unitary_kraus(tc.channels.depolarizingchannel(0.1, 0.2, 0.3), 0, status=x) return c.state
            ()
X = K.implicit_randn(10)
f_vmap(X)
```

从概念上讲，线

Listing 1.50: 线

```
f_vmap = K.vmap(f, vectorized_argnums=0)
```

创建一个函数，作为

$$f_vmap\begin{pmatrix} x_0 \\ x_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \end{pmatrix} \tag{1.13}$$

而参数 vectorized_argnums=0 表示这是我们希望以并行方式批量计算的 f 的第 2 个参数（在本例中是唯一参数）。

1.6. 高级功能

1.6.1. 有条件的测量和后选择

TensorCircuit 允许执行两种与测量结果有关的操作。这两种操作是：(i) 条件性测量，其结果可用于控制下游的条件性量子门，以及 (ii) 后期选择，允许用户选择与特定测量结果相对应的后期测量状态。

有条件的测量

cond_measure 命令用于模拟对一个量子比特进行 Z 测量的过程，产生一个由 Born 规则给出的概率的测量结果，并根据测量结果折叠波函数。获得的经典测量结果可以作为后续量子操作的控制，通过 conditional/$_gateAPI$□□□□□□□□□□□□□□□□□□□□.

Listing 1.51: 远程传输电路

```
# quantum teleportation of state |psi> = a|0> + sqrt(1-a^2)|1>
a = 0.3
input_state = np.kron(np.array([a, np.sqrt(1 - a ** 2)]), np.array([1, 0, 0, 0]))
```

```
c = tc.Circuit(3, inputs=input_state) c.h(2)
c.h(0)
# mid-circuit measurements
# if x = 0 apply I, if x = 1 apply X (to qubit 2)
c.conditional_gate(x, [tc.gates.i(), tc.gates.x()], 2)
# if z = 0 apply I, if z = 1 apply Z (to qubit 2)
c.conditional_gate(z, [tc.gates.i(), tc.gates.z()], 2)
```

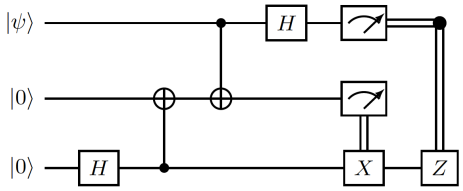

图 1.7: 用 c.cond_measure 和 c.conditional_gate 实现的远程传输电路。

后期选择

在 TensorCircuit 中通过 post_select 方法启用后选择。这允许用户通过 keep 参数来选择一个量子比特的后 Z 测量状态。与 cond_measure 不同的是，post_select 返回的状态没有被规范化。作为一个例子，考虑

Listing 1.52: 后期选择

```
c = tc.Circuit(2, inputs=np.array([1, 0, 0, 1] / np.sqrt(2)))
c.post_select(0, keep=1) # measure qubit 0, post-select on outcome 1
c.state()
```

它初始化了一个 2 比特的最大纠缠状态 $|\psi\rangle = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$。然后，第一个量子比特 (q_0) 在 Z 基中被测量，与测量结果 1 相对应的非正常化状态 $|11\rangle/\sqrt{2}$ 被后选。

这种带有非正常化状态的后选方案速度很快，例如可以用来探索各种量子算法和非微观的量子物理学，如测量诱导的纠缠相变 [52-55]。

B. Pauli string expectation

1.6.2. 保利期望

最小化保利弦之和的期望值是量子算法中的一项常见任务。例如，在 VQE 基态制备的 n 位横场 Ising 模型（TFIM）中，哈密顿的

$$H = \sum_{i=0}^{n-2} J_i X_i X_{i+1} - \sum_{i=0}^{n-1} h_i Z_i \tag{1.14}$$

其中 J_i,h_i 是模型参数，我们希望最小化

$$E(\theta) = \langle H \rangle_\theta = \sum_{i=0}^{n-2} J_i \langle X_i X_{i+1} \rangle_\theta - \sum_{i=0}^{n-1} h_i \langle Z_i \rangle_\theta \tag{1.15}$$

与电路参数 $theta$ 有关。TensorCircuit 提供了许多方法来计算这种形式的表达式。这种形式的表达式，在不同的情况下都很有用。在高层次上，这些方法是

- 循环计算条款，每个条款由 c.expectation_ps 计算。

- 向 operator_expectation 函数提供密集矩阵、稀疏矩阵或汉密尔顿的 MPO 表示。
- 使用 vmap 以矢量并行方式计算每个项，其中每个项作为结构矢量输入。

这些方法的基础是各种表示 Pauli 算子串的方法和这些表示法之间的转换。在详细介绍上述方法之前，让我们介绍一下 TensorCircuit 中使用的 Pauli 结构向量表示法。

泡利结构和权重

一串作用于 n 个量子位的保利算子可以表示为一个长度为 n 的矢量 $v \in \{0,1,2,3\}^n$，其中 $v_i = j$ 的值对应于 σ_i^j，即作用于量子位 i 的保利算子 σ^j（其中 $\sigma^0 = I, \sigma^1 = X, \sigma^2 = Y, \sigma^3 = Z$。例如，在这个符号中，如果 $n = 3$，术语 X_1X_2 对应于 $v = [0,1,1]$。我们把保利弦的这种矢量表示称为结构，而结构的列表，即哈密顿中的每个保利弦项都有一个结构，被用来作为输入，以多种方式计算期望值之和。

Listing 1.53: 远程传输电路

```
# Pauli structures for Transverse Field Ising Model
structures = []
for i in range(n - 1):
        s = [0 for _ in range(n)]
        s[i + 1] = 1 structures.append(s)
for i in range(n):
        structures.append(s)
```

如果每个结构都有一个相关的权重，例如 X_iX_{i+1} 项在哈密顿（3）中具有权重 J_i，那么我们定义一个相应的权重张量

Listing 1.54: 远程传输电路

```
# Weights, taking J_i = 1.0, all h_i = -1.0
J_vec = [1.0 for _ in range(n - 1)]
h_vec = [-1.0 for _ in range(n)]
weights = tc.array_to_tensor(np.array(J_vec + h_vec))
```

使用 c.expectation_ps 的明确循环

正如第三节 B 所介绍的，给定一个 TensorCircuit 量子电路 c，可以通过向 c.expectation_ps 提供一个索引列表来计算单个 Pauli 弦的期望值。然后可以通过一个简单的循环来计算总和。

Listing 1.55: 明确循环

```
def tfim_energy(c,J_vec, h_vec)
        e = 0.0
        n = c._nqubits
        for i in range(n):
        for i in range(n-1):
                e+= J_vec[i] * c.expectation_ps(x=[i,i+1])
```

通过 operator_expectation 的哈密顿人的期望值

给定一个 TensorCircuit 量子电路 c 和代表哈密顿的算子 op，能量的期望值也可以通过 TensorCircuit 模板库的 operator_expectation API 计算出来

Listing 1.56: 哈密顿人的期望值

```
e = tc.templates.measurements.operator_expectation(c, op)
```

运算器 op 本身可以用三种形式之一表示：(i) 密集矩阵，(ii) 稀疏矩阵，和 (iii) 矩阵乘积运算器（MPO）。密集矩阵的输入。作为一个简单的例子，以哈密尔顿为例

$$X_0X_1 - Z_0 - Z_1 \tag{1.16}$$

这有密集的矩阵表示

$$\begin{pmatrix} -2 & & & 1 \\ & & 1 & \\ & 1 & & \\ 1 & & & 2 \end{pmatrix} \tag{1.17}$$

而哈密尔顿的期望值（关于电路 c）可以通过以下方式计算出来

Listing 1.57: 哈密顿人的期望值

```
op = tc.array_to_tensor([[-2, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]])
e = tc.templates.measurements.operator_expectation(c, op)
```

上面的矩阵元素是手工输入的。TensorCircuit 还提供了一种从相关的 Pauli 结构和权重中生成矩阵元素的方法，例如。

Listing 1.58: 相关的 Pauli 结构和权重中生成矩阵元素的方法

```
structure = [[1, 1], [0, 3], [3, 0]]
weights = [1.0, -1.0, -1.0]
H_dense = tc.quantum.PauliStringSum2Dense(structure , weights)
```

稀疏矩阵输入

如果哈密顿是稀疏的，可以在空间和时间上获得明显的计算优势，在这种情况下，运算符的稀疏表示是最好的。这可以通过一个两阶段的过程从 Pauli 结构列表中转换，以一种与后端无关的方式实现。首先我们转换为 COO（COOrdinate）格式的稀疏 numpy 矩阵。例如，在第 VI B 1 节中定义了结构和权重后，我们调用

Listing 1.59: 转换为 COO（COOrdinate）格式的稀疏 numpy 矩阵

```
H_sparse_numpy = tc.quantum.PauliStringSum2COO_numpy(structures , weights)
```

然后我们可以转换为与所选后端 K 兼容的稀疏张量。

Listing 1.60: 后端 K 兼容的稀疏张量

```
H_sparse = K.coo_sparse_matrix(
        np.transpose(np.stack([H_sparse_numpy.row,
        H_sparse_numpy.col])), H_sparse_numpy.data,
shape=(2 ** n, 2 ** n),
```

然后对这个稀疏张量调用 operator_expectation

Listing 1.61: 后端 K 兼容的稀疏张量

```
e = tc.templates.measurements.operator_expectation(c, H_sparse)
```

MPO 输入。TFIM 哈密顿，作为一个短程自旋哈密顿，承认一个有效的矩阵积算子表示。同样，这是一个使用 TensorCircuit 的两阶段过程。我们首先通过 TensorNetwork[44] 或 Quimb 软件包 [56] 将哈密尔顿转换为 MPO 表示。

Listing 1.62: 后端 K 兼容的稀疏张量

```
# generate the corresponding MPO by converting the MPO in tensornetwork package

Jx = np.array([1.0 for _ in range(n - 1)]) # strength of xx interaction (OBC)
Bz = np.array([1.0 for _ in range(n)]) # strength of transverse field
hamiltonian_mpo = tn.matrixproductstates.mpo.FiniteTFI(Jx, Bz, dtype=np.complex64)
```

然后将 MPO 转换成与 TensorCircuit 兼容的 QuOperator 对象。

Listing 1.63: 后端 K 兼容的稀疏张量

```
hamiltonian_mpo = tc.quantum.tn2qop(hamiltonian_mpo) # QuOperator in TensorCircuit
```

然后可以通过 operator_expectation 来计算能量的期望值

Listing 1.64: 后端 K 兼容的稀疏张量

```
e = tc.templates.measurements.operator_expectation(c, hamiltonian_mpo)
```

4. 保利结构上的 vmap

给定一个状态 $|s\rangle$，具有给定结构的保利弦 P 的期望值 $\langle s|P|s\rangle$ 可以作为张量输入的函数计算为

Listing 1.65: 后端 K 兼容的稀疏张量

```
# assume the following are defined
# state: 2**n vector of coefficients corresponding to input state
# structure: Pauli structure (i.e. list of integers {0,1,2,3}**n)
state = tc.array_to_tensor(state)
```

表 1.1: Add caption

time(s)	CPU	CPU
explicit loop	65.7	119
vmap over Pauli structures	0.68	0.0018
dense matrix representation	0.26	0.0015
sparse matrix representation	0.008	0.0014

表二. 保利弦和评估方法的性能基准，哈密尔顿对应于 H2O 分子。12 个量子比特用于 STO-3G 轨道的二进制编码。该量子比特哈密尔顿总共包含 1390 个保利弦项。数据是使用 JAX 后端获得的。GPU 模拟在 Nvidia T4 GPU 上进行，而 CPU 模拟使用 Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz。在这种情况下，MPO 表示法并不适用，因为所需的债券尺寸太大。

Listing 1.66: 后端 K 兼容的稀疏张量

```
def e(state , structure):
        c = tc.Circuit(n, inputs=state)
        return tc.templates.measurements.parameterized_measurements(
                c, structure , onehot=True
```

其中参数化 _measurements 函数用于计算电路输出的期望值。然后，如果哈密顿由保利结构 $[v_1, \cdots, v_k]$ 的列表表示，$vmap$ 可以用来计算每项并联的相对于电路 c 的期望值。

Listing 1.67: 后端 K 兼容的稀疏张量

```
e_vmap = K.vmap(e, vectorized_argnums=1)
```

与第 V B 1 节中的例子类似，vmapping 创建了一个函数，它作为

$$e_{vmap}\left(s, \begin{pmatrix} \leftarrow & V_1 & \rightarrow \\ & \vdots & \\ \leftarrow & V_k & \rightarrow \end{pmatrix} = \begin{pmatrix} e(s, v_1) \\ \vdots \\ e(s, v_k) \end{pmatrix}\right. \tag{1.18}$$

即输出一个对应于哈密顿项的期望值向量，以及

Listing 1.68: 后端 K 兼容的稀疏张量

```
vectorized_argnums=1
```

表示应该并行计算 $e(s, v)$ 函数的第一个参数 v，而第 2 个参数 $(i.e, s$ 是固定的。有了第六节 B 1 中定义的结构和权重，哈密顿人相对于电路 c 的期望值就可以被计算为

Listing 1.69: 后端 K 兼容的稀疏张量

```
s = c.state()
e_terms = e_vmap(s, structures)
hamiltonian_expectation = K.sum(e_terms * K.real(weights))
```

我们以这些不同的方法为基准，对对应于 H2O 分子和 TFIM 自旋模型的哈密尔顿进行保利弦和评估，结果分别见表二和表三。详情见 examples/vqeh2o_benchmark.py 和 examples/vqet-fim_benchmark.py。在 H2O 案例中，我们观察到使用稀疏矩阵表示的 VQE 评估比天真循环加速了 85,000 倍。与其他大多数量子软件中使用的天真循环相比，我们观察到使用稀疏矩阵表示的 VQE 评估加速了 85000 倍。

1.6.3. vmap 和 vectorized_value_and_grad

Jupyter notebook: 6-3-vmap.ipynb

正如我们在 VB1 和 VIB4 节中看到的，$vmap$ 允许同时并行地进行成批的函数评估。如果需要对梯度和函数值进行批量评估，那么可以通过 $vectorized_value$and$_grad$ 来完成。在最简单的情况下，考虑一个函数 $f(x, y)$

表 1.2: Add caption

time(s)	CPU	CPU
explicit loop	1.73	0.11
vmap over Pauli structures	10.68	0.2
sparse matrix representation	0.61	0.0086
MPO matrix representation	0.0007	0.0039

表三. 在具有开放边界条件的 20 比特 TFIM 哈密顿上进行不同保利弦和评估的性能基准。该量子位哈密尔顿包含 39 个保利弦项。数据是使用 JAX 后端获得的。GPU 模拟使用 Nvidia T4 GPU，

而 CPU 模拟使用 Intel(R) Xeon(R) Platinum 8255C CPU @ 2.50GHz。由于内存要求过高，密集矩阵表示法不适用于 20 比特的系统。

其中 $x \in \mathbb{R}^p$，$y \in \mathbb{R}^q$ 都是向量，我们希望对输入 x 的一批 $x_1, x_2, \cdots, x_k$ 进行 $f(x, y)$ 和 x□yf(x,y)=∂f(x,y1),...,∂f(x,yq)□ 的评价。

1.7. 综合实例

1.8. 前景和结束语

我们介绍了 TensorCircuit，一个开源的 Python 软件包，旨在满足更大和更复杂的量子计算模拟的要求。TensorCircuit 建立在现代机器学习库的基础上，并结合了所有主要的工程范式，其灵活和可定制的张量网络引擎可以实现高性能的电路计算。**展望未来**。我们将继续开发 TensorCircuit，以提供一个更有效、更优雅、全功能和兼容 ML 的量子软件包。在我们的优先列表中，最重要的是

1. 更好的张量网络收缩路径搜索器：整合更先进的算法和机器学习技术以实现最佳收缩路径搜索。
2. 脉冲级优化和量子控制：实现端到端的可微调脉冲级优化和最优量子控制计划 [65, 66]。
3. 分布式量子电路模拟：实现张量网络并行切片和多主机上的分布式计算。
4. 基于 MPS 的近似电路仿真：引入类似 TEBD 的算法 [67, 68] 来近似仿真大尺寸和大深度的量子电路。
5. 更多的量子感知或流形感知的优化器：包括诸如 SPSA[69]、rotosolve[70] 和 Riemannian 优化器 [71] 等优化器。
6. 量子应用：为金融、材料、能源、生物、药物发现、气候预测等方面的量子计算开发应用级库。

随着量子计算理论和硬件的持续快速发展，我们希望 TensorCircuit 这个为 NISQ 时代设计的量子模拟器平台能够在这个令人兴奋的领域的学术和商业进展中发挥重要作用。

1.9. Acknowledgements:

The authors would like to thank our teammates at the Tencent Quantum Laboratory for supporting this project, and Tencent Cloud for providing computing resources. Shi-Xin Zhang would like to thank Rong-Jun Feng, Sai-Nan Huai, Dan-Yu Li, Zi-Xiang Li, Lei Wang, Hao Xie, Shuai Yin and Hao-Kai Zhang for their helpful discussions.

1.10. BIBLIOGRAPHY

[1] M. A. Nielsen and I. L. Chuang, Quantum Computation and Quantum Information: 10th Anniversary Edition, 10th ed. (Cambridge University Press, USA, 2011).

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: A system for large-scale machine learning, in 12th USENIX symposium on operating systems design and implementation (OSDI 16) (2016) pp. 265–283.

[3] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, JAX: composable transformations

of Python+NumPy programs (2018).

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, in Advances in Neural Information Processing Systems.

[5] J. Gray, cotengra, https://github.com/jcmgray/cotengra (2020).

[6] J. Gray and S. Kourtis, Hyper-optimized tensor network contraction, Quantum 5, 410 (2021).

[7] J. Preskill, Quantum computing in the nisq era and beyond, Quantum 2, 79 (2018).

[8] K. Bharti, A. Cervera-Lierta, T. H. Kyaw, T. Haug, S. Alperin-Lea, A. Anand, M. Degroote, H. Heimonen, J. S. Kottmann, T. Menke, W.-K. Mok, S. Sim, L.-C. Kwek, and A. Aspuru-Guzik, Noisy intermediate-scale quantum algorithms, Reviews of Modern Physics 94, 015004 (2022).

[9] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio, and P. J. Coles, Variational quantum algorithms, Nature Reviews Physics 3, 625 (2021).

[10] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'brien,A variational eigenvalue solver on a photonic quantum processor, Nature communications 5, 1 (2014).

[11] E. Farhi, J. Goldstone, and S. Gutmann, A quantum approximate optimization algorithm, arXiv preprintarXiv:1411.4028 (2014).

[12] A. Kandala, A. Mezzacapo, K. Temme, M. Takita, M. Brink, J. M. Chow, and J. M. Gambetta, Hardware-efficientvariational quantum eigensolver for small molecules and quantum magnets, Nature 549, 242 (2017).

[13] J. R. McClean, S. Boixo, V. N. Smelyanskiy, R. Babbush, and H. Neven, Barren plateaus in quantum neuralnetwork training landscapes, Nature communications 9, 1 (2018).

[14] E. R. Anschuetz, Critical points in quantum generative models, arXiv:2109.06957 (2021).

[15] M.-H. Yung, J. Casanova, A. Mezzacapo, J. Mcclean, L. Lamata, A. Aspuru-Guzik, and E. Solano, From transistor to trapped-ion computers for quantum chemistry, Scientific reports 4, 1 (2014).

[16] U. Schollwöck, The density-matrix renormalization group in the age of matrix product states, Annals of Physics326, 96 (2011).

[17] Y. LeCun, Y. Bengio, and G. Hinton, Deep learning, Nature 521, 436 (2015).

[18] M. Bartholomew-Biggs, S. Brown, B. Christianson, and L. Dixon, Automatic differentiation of algorithms, J.Comput. Appl. Math. 124, 171 (2000).