

PADDLE QUANTUM 入门手册

Copyright (c) 2020 Institute for Quantum Computing, Baidu Inc. All Rights Reserved.

总览

这是一份简洁、实用的关于量子机器学习 (Quantum Machine Learnig, QML) 的介绍, 面向读者包括但不限于物理、数学和计算机背景。本手册主要采用 Jupyter Notebook 的交互形式 (调用 Numpy, Matplotlib等 Python包以及飞桨Paddlepaddle深度学习框架来实现基于线性代数的量子运算和机器学习优化问题)。我们不仅提供了关于量子计算的一些基础教程同时还能手把手带你完成属于你自己的第一份量子机器学习算法。这并不是关于量子计算的百科全书, 但我们涉及的案例经常出现在教科书中以及文献中。如果你想深入挖掘一些相关的基础知识, 我们也提供了一些外部链接方便自己学习。

最后修改于: 2020年9月9日 由量浆 Paddle Quantum开发小组共同完成。

目录

- 入门手册总览
- 安装:
 - (Conda 与环境配置)
 - (安装 Paddle Quantum包)
- 量子计算基础:
 - (量子比特)
 - (量子门)
 - (测量)
 - (示例和练习)
- 量子电路模板的搭建:
 - (量子神经网络QNN)
 - (内置电路模板)
- 量子的运算模式:
 - (波函数向量模式)
 - (密度矩阵模式)
 - (练习: 贝尔态)
- 飞桨优化器的使用:
 - (简单案例)
 - (应用与练习)
- 量子机器学习案例:
 - (无监督学习 - VQE)
- 参考文献

安装教程

Conda 与 Python 环境安装

我们推荐使用 [Anaconda](#) 作为 Python3 的开发环境管理工具，Anaconda 支持多种主流操作系统 (Windows, MacOS, 以及 Linux) 。Anaconda本身提供 Scipy, Numpy, Matplotlib等科学计算、作图包，最主要的是其自带 Python开发环境的管理器 conda，可以用来安装或者更新主流 Python包。这里我们提供一个例子来学习使用 conda创建和管理环境：

1. 首先进入命令行 (Terminal) 界面：Windows用户可以使用组合键 `win + R` 打开运行程序再输入 `cmd` / Mac用户可以使用组合键 `command* + 空格` 再输入 `Terminal`。
2. 进入 Terminal 后输入 `conda create --name paddle_quantum_env python=3.6` 创建名为 `paddle_quantum_env` 的 Python3.6 环境。在 Terminal 内通过 `conda env list` 查看已有的环境，然后通过 `conda activate paddle_quantum_env` 进入我们刚建立的环境。
3. 为了能正确运行 Jupyter Notebook 我们还需要安装 `conda install jupyter notebook`。安装完成之后，如果你想开启 Jupyter 只需要在Terminal内激活正确的环境然后输入 `jupyter notebook` 即可。

```
(base) [redacted]$ conda create --name paddle_quantum_env python=3.6
Collecting package metadata (current_repodata.json): done
Solving environment: done

==> WARNING: A newer version of conda exists. <==
  current version: 4.8.2
  latest version: 4.8.4

Please update conda by running

  $ conda update -n base -c defaults conda

## Package Plan ##

environment location: /Users/[redacted]/.conda/envs/paddle_quantum_env

added / updated specs:
- python=3.6

The following packages will be downloaded:

package | build | size | url
-----|-----|-----|-----
pip-20.2.2 | | 1.8 MB | https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
setuptools-49.6.0 | py36_0 | 756 KB | https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
sqlite-3.33.0 | hffcf06c_0 | 1.3 MB | https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main

Total: 3.8 MB

The following NEW packages will be INSTALLED:
```

关于 conda 更多的本地指令请参考 [官方教程](#)。

此外，你也可以通过使用 [Anaconda Navigator](#) 开启 jupyter notebook。

以下是这个教程中你需要使用的包：

- Numpy
 - Paddlepaddle 1.8.3 以上
 - Paddle Quantum
-

安装 Paddle和 Paddle Quantum

接着我们先讲解下如何安装飞桨Paddlepaddle深度学习框架。用户需要再次打开 Terminal 界面，输入 `conda activate paddle_quantum_env` 进入我们新建的Python 环境。接着输入 `conda install paddlepaddle` 安装最新的飞桨Paddle。如果你在按照以上方案安装 `paddle-cpu` 或者 `paddle-gpu` 遇到问题时参考 [官方链接](#)。或者考虑以下备选方案：在 Terminal 界面输入 `pip install paddlepaddle` 或者开启 jupyter notebook 后运行以下的命令。

```
1 from IPython.display import clear_output
2
3 # 备用方法： 通过PyPI安装最新的飞桨Paddle
4 !pip install paddlepaddle
5 clear_output()
```

在安装完成后，请通过以下代码段来检验是否成功安装 Paddle。

```
1 from paddle import fluid
2
3 # 检查飞桨Paddle是否成功安装
4 fluid.install_check.run_check()
```

```
1 Running Verify Fluid Program ...
2 Your Paddle Fluid works well on SINGLE GPU or CPU.
3 Your Paddle Fluid works well on MUTIPLE GPU or CPU.
4 Your Paddle Fluid is installed successfully! Let's start deep
  Learning with Paddle Fluid now
```

接着我们安装 Paddle Quantum包，很遗憾目前我们还不支持通过 pip安装。需要用户在 Terminal 界面通过 git指令 `git clone http://github.com/PaddlePaddle/quantum` 下载文件，然后输入 `cd quantum` 和 `pip install -e .` 完成安装。接着在 Terminal 界面输入 `pip list` 查看是否在正确的环境中安装完成。关于 git的使用和安装，请参考这篇 [教程](#)。此外，如果你需要更多的关于安装 Paddle-Quantum 的帮助，可以参考我们的 [Github链接](#) 或者通过 Github Issues联系我们。

```
1 import numpy as np
2 from paddle.complex import matmul, transpose, trace
3 from paddle_quantum.circuit import UAnsatz
4 from paddle_quantum.utils import dagger, random_pauli_str_generator,
  pauli_str_to_matrix
5 from paddle_quantum.state import vec, vec_random, density_op,
  density_op_random
```

以上的几个代码块没有任何报错的话，恭喜你！接着就可以顺利运行全部的教程了！

(回到 [目录](#))

量子计算基础

量子计算 (Quantum Computing QC) 是利用量子物理中特有的现象 (量子叠加态, 量子纠缠) 来设计相应的量子算法以解决 (物理、化学、计算机等领域) 特定的任务。现有的量子计算有好几种模型, 比如有基于绝热定理的绝热计算模型以及基于测量的 MBQC 模型等等。在本手册中, 我们主要讨论目前影响力最大、使用最广泛的量子电路 (Quantum circuit) 模型。在量子电路的框架下, 运算最基本的组成单元是量子比特 (qubit)。这与经典计算机中比特 (bit) 的概念很相似。经典比特只能处于 0 和 1 两种状态中的某一种 (物理图景上可以对应高低电位)。与之不同的是, 量子比特不仅可以处于两个状态 $|0\rangle$ 还有 $|1\rangle$ 还可以处于两者的叠加态 (稍后我们来具体讲解下这一概念)。而常见的量子计算模型, 就是利用量子逻辑门操控这些量子比特。逻辑门运算的基本理论是线性代数, 在此我们假定读者已经具备一定的线性代数基础。

什么是量子比特?

数学表示

在量子力学中, 一个微观粒子的量子态可以表示为由两个正规正交基线性组合得到的向量, 这些基向量一般可以写为:

$$|0\rangle := \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle := \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

我们这里向量的表示方法采用了量子物理上传统的狄拉克表示 (bra-ket)。这两个单位正交向量 $\{|0\rangle, |1\rangle\}$, 它俩一般被称为**计算基** (computational basis)。物理图景中我们可以认为 $|0\rangle$ 和 $|1\rangle$ 分别对应一个原子的能量基态和激发态或者其他一些二分类状态。一个量子比特所有可能的态可以看作是二维希尔伯特空间中所有的归一化向量, 这个希尔伯特空间的一组正规正交基正是 $\{|0\rangle, |1\rangle\}$ 。而更多的量子比特系统也同样可以由高维度的希尔伯特空间中的单位向量表示, 而这个高维希尔伯特空间的正交基就是 $\{|0\rangle, |1\rangle\}$ 的张量积。比如说, 一个两量子比特 (2-qubit) 系统可以被一个 4 维的希尔伯特空间里的单位复数向量表示, 而这个希尔伯特空间的正规正交基是:

$$|00\rangle = |0\rangle \otimes |0\rangle := \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad |01\rangle = |0\rangle \otimes |1\rangle := \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad |10\rangle = |1\rangle \otimes |0\rangle := \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad |11\rangle = |1\rangle \otimes |1\rangle := \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

我们默认最左边的位置代表第一个量子比特, 依此类推。其中符号 \otimes 是张量积运算。其工作原理大概如下: 给定两个矩阵 $A_{m \times n}$ 还有 $B_{p \times q}$, 那么 A, B 的张量积为

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}_{(mp) \times (nq)}$$

一个单量子比特所处的任意量子态 $|\psi\rangle$ 可以写成基向量 $|0\rangle$ 和 $|1\rangle$ 的线性叠加, 也就是说, 它可以被描述成一个 $|0\rangle$ 和 $|1\rangle$ 的线性组合

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle := \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \quad \text{其中 } \alpha, \beta \in \mathbb{C}$$

其中 α 和 β 可以是复数，他们表示概率振幅。这意味着当我们测量这个量子比特时，根据波恩法则，测量得到量子比特处于 $|0\rangle$ 状态的概率是 $|\alpha|^2$ ；而测量得到 $|1\rangle$ 的概率是 $|\beta|^2$ 。由于概率相加等于1，我们必须加入如下的限制条件： $|\alpha|^2 + |\beta|^2 = 1$ 。

布洛赫球面 (Bloch Sphere) 表示

我们用一个球面上的点来表示一个量子比特可能处于的量子态，这个球面被称为**布洛赫球面**，(见图1)

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle$$

注意：多个量子系统的状态就无法用布洛赫球面来表示。如果是一个经典比特的话，那么它只有两个状态0和1，也就是布洛赫球面的北极和南极。这两个位置恰好对应着 $|0\rangle$ 和 $|1\rangle$ 。而一个量子比特不光可以处于两极，它可以在球面上任意一点，这样一种叠加的状态是经典比特做不到的。举例来说，量子态 $\frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$ 就处于球面赤道和y-正半轴的交界处。

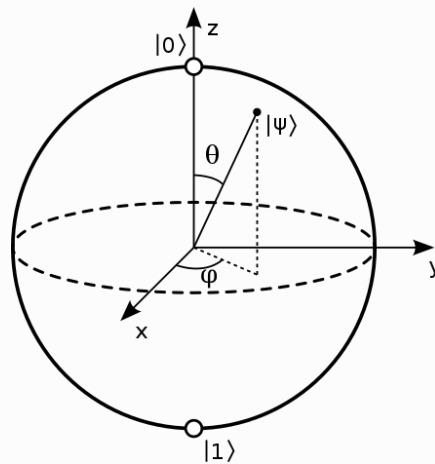


图 1. 单量子比特的布洛赫球面表示. (图片来源)

下面的内容面向对量子计算更熟悉的读者。如果你阅读这段感到困难，不用担心，您可以选择略过这一节，这不会对理解接下的内容产生影响。由于量子比特之间的交互以及去相干问题 (Decoherence)，因此，对于一个具有多量子比特的系统来说，它的单量子比特子系统将不再处于纯态 (pure state)，而是演变成混合态 (mixed state)。混合态可以看成不同纯态的按照一定概率的混合。单比特的混合态可以看成是布洛赫球内部的点，而不是存在于球表面。通常来说，混合态需要用到量子力学的密度矩阵形式来描述，比如

$$\rho_{\text{mixed}} = \sum_i P_i |\psi_i\rangle\langle\psi_i| = \frac{1}{2}|0\rangle\langle 0| + \frac{1}{2}|1\rangle\langle 1| := \frac{1}{2} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

其中行向量 (bra) $\langle 0|$ 是列向量 (ket) $|0\rangle$ 的复共轭转置。

注：如需更多信息，可参考维基百科 [链接](#)

什么是量子逻辑门？

在经典计算机中，我们可以在经典比特上施加基本的逻辑运算(非门 NOT, 与非门 NAND, 异或门 XOR, 与门 AND, 或门 OR)并组合成更复杂的运算。而量子计算则有完全不同的一套逻辑运算，它们被称为量子门 (quantum gate)。我们并不能在一个量子计算机上编译现有的C++程序。因为经典计算机和量子计算机有不同的逻辑门构造，所以量子算法是需要利用这些量子门的特殊性来构造的。量子门在数学上可以被表示成酉矩阵 (unitary matrix)。酉矩阵操作可以保证向量的长度不变，这是个很好的性质。不然我们对一个纯态量子比特进行操作，会让它劣化成混合态导致其无法接着使用。酉矩阵定义为：

$$U^\dagger U = U U^\dagger = I, \quad \text{并且} \quad \|\psi\| = \|U\psi\| = 1$$

其中 U^\dagger 是 U 的埃尔米特转置， I 表示单位矩阵。但是酉矩阵作为量子门的物理意义是什么？这意味着所有的量子门都必须是可逆的。对于任何一个量子门运算，都可以找到一个与其对应的反向运算。除此之外，酉矩阵必须是个方阵。因为量子门的输入和输出要求有同样数量的量子比特。一个作用在 n 量子比特的量子门可以写成一个 $2^n \times 2^n$ 的酉矩阵。最常见的（也是物理上最容易实现的）量子门作用在一个或两个量子比特上，就像经典逻辑门那样。

单量子比特门

接下来，我们介绍在量子计算中非常重要的单量子比特门，包括泡利矩阵 $\{X, Y, Z\}$ 、单比特旋转门 $\{R_x, R_y, R_z\}$ 和哈达玛门 H 。其中非门 (NOT gate) 对于经典或量子计算都很重要，酉矩阵表示为：

$$X := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

这个量子门（酉矩阵）作用在单量子比特（一个复向量）上本质上的运算是矩阵乘以向量

$$X|0\rangle := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle \quad \text{and} \quad X|1\rangle := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

回忆起前面的布洛赫球面表示，这个矩阵 X 作用在一个量子比特（布洛赫球面上的一点）就相当于关于布洛赫球的X轴旋转角度 π 。这就是为什么 X 可以表示成 $R_x(\pi)$ （相差全局相位 $e^{-i\pi/2} = -i$ ）。其他两个泡利矩阵 Y 和 Z 在这一点上也非常相似（代表绕 Y 和 Z 轴旋转 π 运算）：

$$Y := \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad \text{and} \quad Z := \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

一般来说，任何一个在布洛赫球关于相应的轴旋转 θ 角度的量子门可以表示为：

$$R_x(\theta) := \begin{bmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}, \quad R_y(\theta) := \begin{bmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}, \quad R_z(\theta) := \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}$$

除了旋转门之外，最重要的单比特门就是哈达玛门了。对应的布洛赫球面解释是两个旋转组成的，先是按Z轴旋转 π ，然后按Y轴旋转 $\pi/2$ 。它的矩阵表示是：

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

两比特量子门

从单量子比特门我们可以拓展到两量子比特门。有两种拓展方式，第一种是只挑选出一个量子比特，在上面施加单量子比特门，其他的量子比特则不受影响。有的时候，您会见到如下图所示的量子电路：

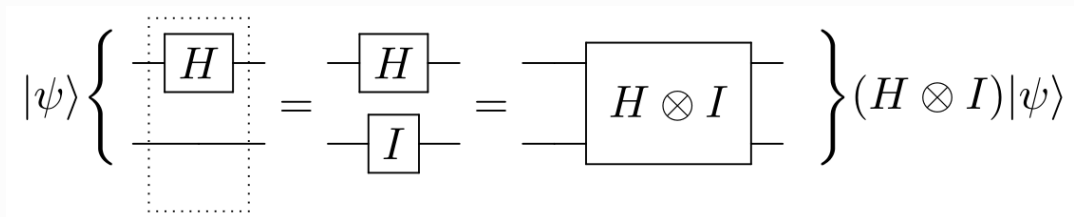


图 2. 两量子比特逻辑运算的电路表示和解读。(图片来源)

作用在两量子比特上的量子门可以表示成一个 4×4 酉矩阵

$$U = H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

另一种拓展方式是作用在全部两个量子比特上。比如 CNOT，这个门会使得一个量子比特的状态影响到另一个量子比特的状态

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

我们观察一下它作用在不同的初始量子态上

$$CNOT|00\rangle = |00\rangle, \quad CNOT|01\rangle = |01\rangle, \quad CNOT|10\rangle = |11\rangle, \quad CNOT|11\rangle = |10\rangle$$

也就是说，当第一个量子比特处于 $|1\rangle$ 状态时，CNOT 会在第二个量子比特上施加 X 门，如果第一个量子比特处于 $|0\rangle$ 状态，那么第二个量子比特则不受任何影响。这也是为什么 CNOT 会被称为受控非门。

下面是一些常见的量子门及它们的矩阵表示，这些量子门都可以在量浆内被调用。

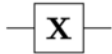

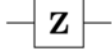
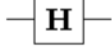
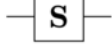
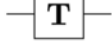
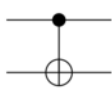
Operator	Gate(s)	Matrix
Pauli-X (X)	 \oplus	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

图 3. 常见的量子门. (图片来源)

注：更多信息可见如下[维基百科链接](#)

什么是量子力学中的测量？

对于一个两分类的量子态，比如电子的自旋 (Spin) ，可以自旋向上。这时我们规定该电子处于 $|0\rangle$ 态。当然电子也可以自旋向下，这时我们规定它处于 $|1\rangle$ 态。神奇的是，电子等微观粒子在被观测之前可以同时处于自旋向上和自旋向下的叠加态 $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ 。那么这个奇怪的叠加态到底指的是什么呢？答案很简单，我们可以去测量一下这个处于“叠加态”的电子。值得注意的是，量子力学中的测量通常指的是一个统计结果而不是单次测量。这是由于测量本身的特性会使得观察后的量子态塌缩。就拿我们前面提到的处于 $|\psi\rangle$ 态的这个电子来举例，如果我们测量这一个电子的自旋，我们会有 $|\alpha|^2$ 的概率观测到自旋向上并且观测后量子态塌缩成 $|0\rangle$ 。同样的，我们也有 $|\beta|^2$ 的概率测量得到自旋向下 $|1\rangle$ 。那么想要精确的得到 α 的数值，一次实验显然是不够的。我们需要拜托物理学家朋友准备了好多好多处于叠加态 $\alpha|0\rangle + \beta|1\rangle$ 的电子，把每一个电子的自旋都测量了一下再统计频率。测量在量子力学中地位比较特殊，如果读者觉得难理解。请参阅 [维基百科-量子力学中的测量](#) 获取更多知识。

示例以及练习

示例: 用Paddle Quantum创建 X 门

注意: 所有的单比特旋转门都按如下规定建立:

$$R_x(\theta) := \begin{bmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}, \quad R_y(\theta) := \begin{bmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix}, \quad R_z(\theta) := \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}$$

因此, 我们不难看出 X 门可以表示为 $R_x(\pi)$ 。以下是代码展示:

```
1 # 设置角度参数 theta = pi
2 theta = np.array([np.pi])
3
4 # 启动 Paddle 动态图模式
5 with fluid.dygraph.guard():
6
7     # 我们需要将 Numpy array 转换成 Paddle 动态图模式中支持的 variable
8     theta = fluid.dygraph.to_variable(theta)
9
10    # 设置计算所需的量子比特数量
11    num_qubits = 1
12
13    # 初始化我们的单比特量子电路
14    cir = UAnsatz(num_qubits)
15
16    # 在第一个量子比特(第0号量子比特)的位置上施加一个 Rx 旋转门, 角度为 pi
17    which_qubit = 0
18    cir.rx(theta, which_qubit)
19
20    # 打印出这个量子门
21    # 转换成 numpy
22    print('量子门的矩阵表达式为: ')
23    print(cir.U.numpy())
```

```
1 量子门的矩阵表达式为:
2  [[ 6.123234e-17+0.j -6.123234e-17-1.j]
3  [ 6.123234e-17-1.j  6.123234e-17+0.j]]
```

结果和 X (NOT) 门只相差一个全局相位 $-i$

$$\text{output} = \begin{bmatrix} 0 & -i \\ -i & 0 \end{bmatrix} = -i \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = -iX$$

有兴趣的话, 你可以仔细思考一下为什么在量子计算中, 全局相位并不重要。

练习: 创建 Y 门

那么按照以上的例子依葫芦画瓢，你是否可以试着自己创建一个 Y 门？试着补全下面的代码

```
1 theta = "your code"
2 with fluid.dygraph.guard():
3
4     theta = fluid.dygraph.to_variable(theta)
5     num_qubits = 1
6     cir = UAnsatz("your code")
7     cir.ry("your code")
8     print(cir.U.numpy())
```

和原来一样，我们还是多了一个全局相位

$$\text{output} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = -i \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = -iY$$

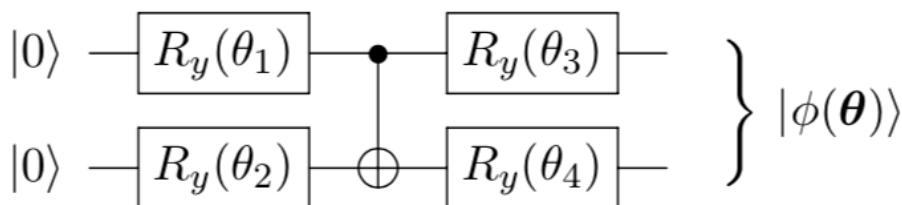
(回到 [目录](#))

量子电路模板/量子神经网络

经过上面的准备，你现在有一定的知识基础可以了解量子机器学习了。简单来说我们要做的就是利用量子电路来替代传统的神经网络来完成机器学习的任务。我们一般会准备一个可调节参数的量子电路（量子神经网络 QNN），很多时候也被称为模板 (Ansatz)，里面的参数是人为可调节的（这些参数其实就是旋转门的角度 θ ）。如果再加上一个精心设计的损失函数，就可以将一个量子计算问题转化为经典的寻找损失函数最小值问题。然后不断调节电路中的参数直到损失函数下降至收敛（此时损失函数达到最优值或次优值），我们就完成优化了。

示例: 如何创建量子神经网络 QNN?

QNN通常可以表示为一些单比特量子旋转门和双比特门的组合。其中一个可以高效利用硬件的架构是只包含 $\{R_x, R_y, R_z, \text{CNOT}\}$ 这四种量子门的模板。它们很容易在NISQ设备（通常是超导量子比特）上实现，因为CNOT只需要实施在相邻量子比特上。一个例子可见下图：



通常来说，每条线代表一个量子比特。我们把图最上端的认为是第一个量子比特，依次往下。从左到右代表我们施加门的时间顺序，先施加最左边的量子门。接下来，我们来看看如何在量浆上建造这个简单的两比特量子神经网络

```
1 # 设置角度参数 theta
2 theta = np.full([4], np.pi)
3
4 # 启动 Paddle 动态图模式
5 with fluid.dygraph.guard():
6
7     # 我们需要将 Numpy array 转换成 Paddle 动态图模式中支持的 variable
8     theta = fluid.dygraph.to_variable(theta)
9
10    # 初始化量子电路
11    num_qubits = 2
12    cir = UAnsatz(num_qubits)
13
14    # 添加单比特旋转门
15    cir.ry(theta[0], 0)
16    cir.ry(theta[1], 1)
17
18    # 添加两比特门
19    cir.cnot([0, 1])
20
21    # 添加单比特旋转门
22    cir.ry(theta[2], 0)
23    cir.ry(theta[3], 1)
24
25    print('图中量子神经网络 U(theta=pi) 的矩阵表达式是:')
26    print(cir.U.numpy().real)
```

```
1 图中量子神经网络 U(theta=pi) 的矩阵表达式是：
2 [[ 0.00000000e+00 -1.00000000e+00  6.1232340e-17 -6.1232340e-17]
3  [-1.00000000e+00  0.00000000e+00 -6.1232340e-17  6.1232340e-17]
4  [-6.1232340e-17  6.1232340e-17  1.00000000e+00  1.2246468e-16]
5  [ 6.1232340e-17 -6.1232340e-17 -1.2246468e-16  1.00000000e+00]]
```

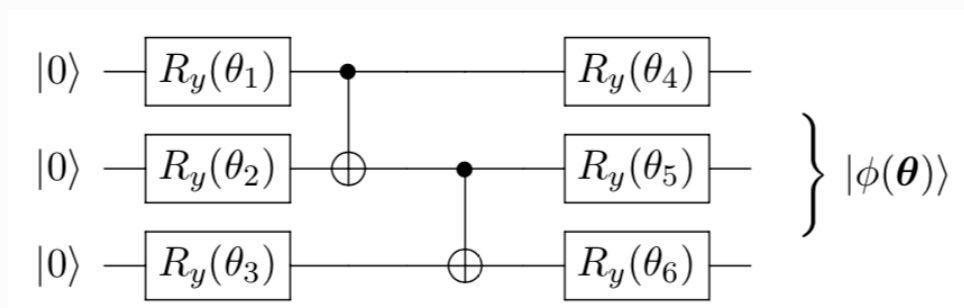
$$\text{output} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

练习:

给你如下代码，你能想象出对应的电路吗？

```
1 theta = np.full([6], np.pi)
2 with fluid.dygraph.guard():
3
4     theta = fluid.dygraph.to_variable(theta)
5
6     num_qubits = 3
7     cir = UAnsatz(num_qubits)
8
9     cir.ry(theta[0], 0)
10    cir.ry(theta[1], 1)
11    cir.ry(theta[2], 2)
12
13    cir.cnot([0, 1])
14    cir.cnot([1, 2])
15
16    cir.ry(theta[3], 0)
17    cir.ry(theta[4], 1)
18    cir.ry(theta[5], 2)
19
```

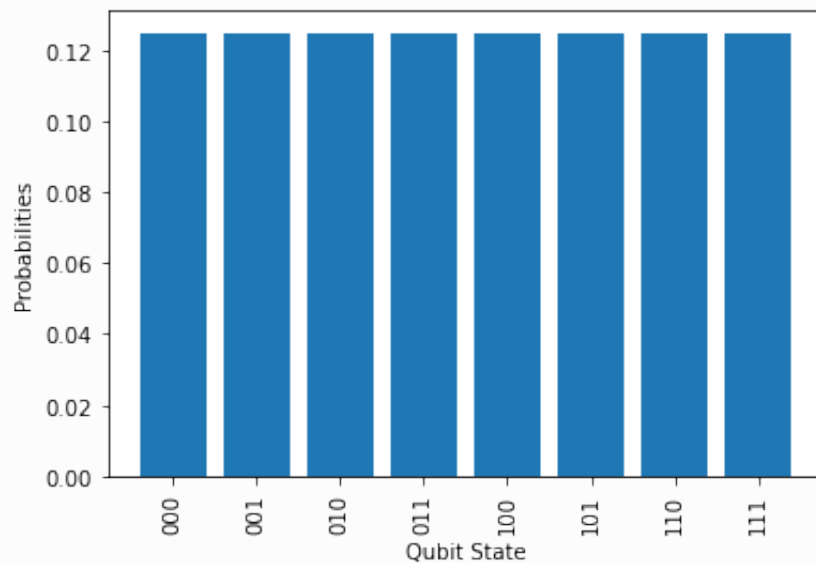
答案如下:



内置的电路模板

在最新版本的 Paddle Quantum中，我们提供了一些内置的电路模板方便场景部署。

```
1 N = 3 # 设置量子比特数
2
3 # 启动 Paddle 动态图模式
4 with fluid.dygraph.guard():
5
6     # 初始化量子电路
7     cir = UAnsatz(N)
8
9     # 给每一个量子比特施加哈达玛门 H
10    cir.superposition_layer()
11
12    # 制备输出态
13    # 如果用户不输入初始量子态，默认初始为 |00..0>
14    final_state = cir.run_state_vector()
15
16    # 获取概率分布的理论值，令 shots = 0
17    cir.measure(shots = 0, plot = True)
```

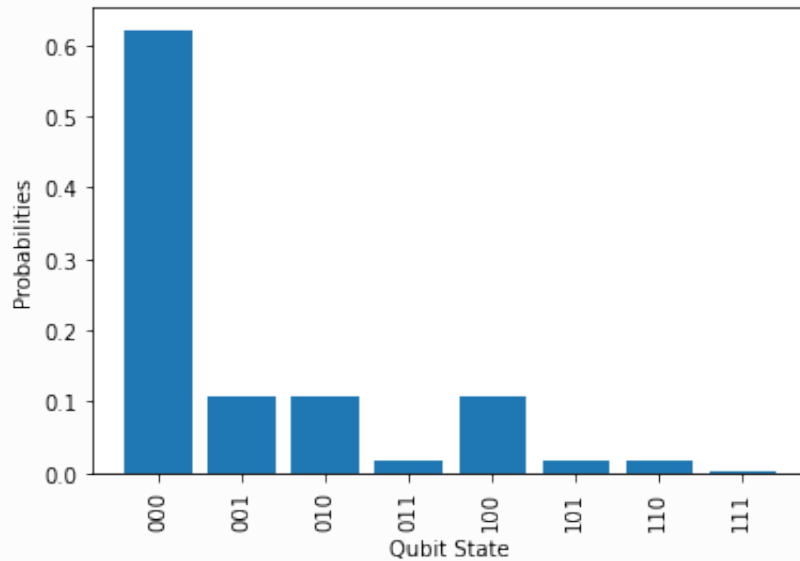


```
1 N = 3 # 设置量子比特数
2
3 # 启动 Paddle 动态图模式
4 with fluid.dygraph.guard():
5
6     # 初始化量子电路
7     cir = UAnsatz(N)
8
9     # 给每一个量子比特施加 Ry(pi/4) 旋转
10    cir.weak_superposition_layer()
11
12    # 制备输出态
```

```

13 # 如果用户不输入初始量子态, 默认初始为 |00..0>
14 final_state = cir.run_state_vector()
15
16 # 获取概率分布的理论值, 令 shots = 0
17 cir.measure(shots = 0, plot = True)

```



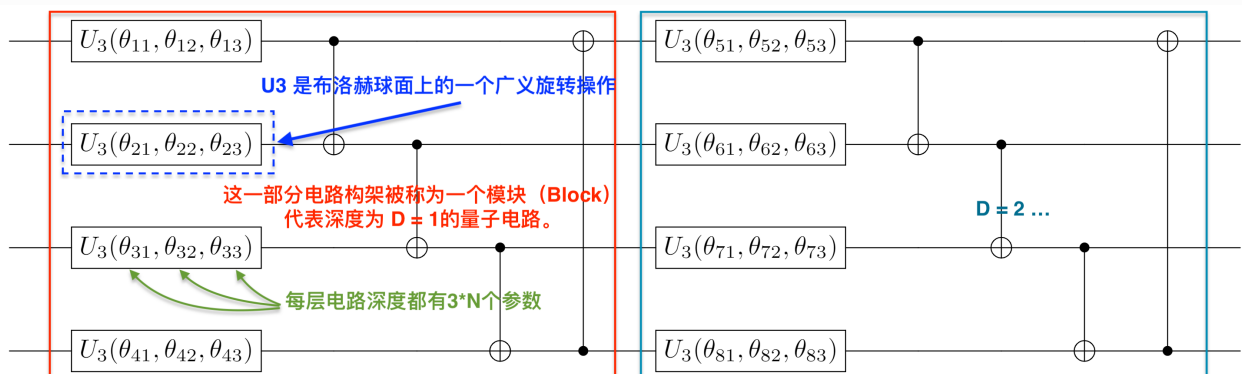
以下是一个使用频率较高的电路模板 `complex_entangled_layer(theta, DEPTH)`, 用户可按照电路深度参数 `DEPTH/D` 快速拓展电路。其中涉及的广义旋转门 U_3 的定义为:

$$U_3(\theta, \phi, \varphi) := \begin{bmatrix} \cos \frac{\theta}{2} & -e^{i\varphi} \sin \frac{\theta}{2} \\ e^{i\phi} \sin \frac{\theta}{2} & e^{i(\phi+\varphi)} \cos \frac{\theta}{2} \end{bmatrix}$$

U_3 旋转门在效果上是等价于以下组合旋转门的,

$$U_3(\theta, \phi, \varphi) = R_z(\phi) * R_y(\theta) * R_z(\varphi) := \begin{bmatrix} e^{-i\frac{\phi}{2}} & 0 \\ 0 & e^{i\frac{\phi}{2}} \end{bmatrix} \begin{bmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix} \begin{bmatrix} e^{-i\frac{\varphi}{2}} & 0 \\ 0 & e^{i\frac{\varphi}{2}} \end{bmatrix}$$

感兴趣的读者不妨自行验证一下。

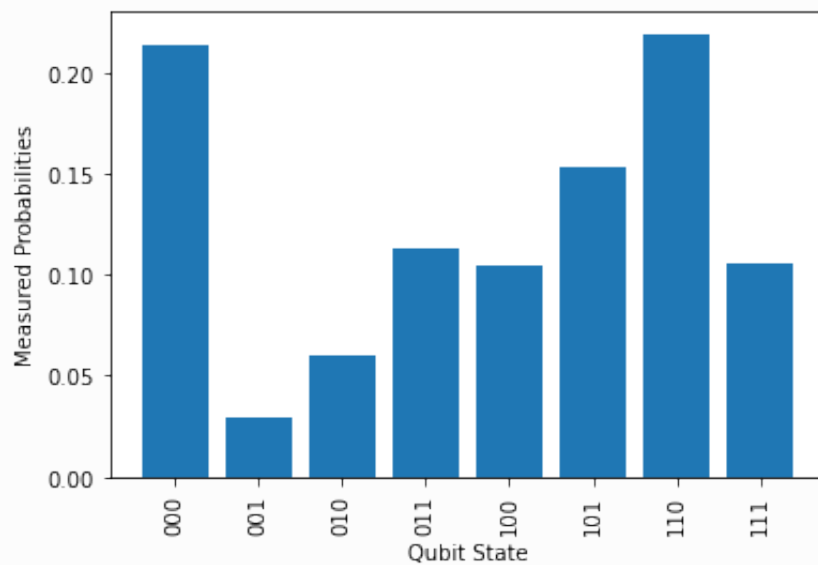


特别地, 当我们处理的任务不涉及虚数时, 使用电路模板 `real_entangled_layer(theta, DEPTH)` 会更加高效 (R_y 旋转门替代 U_3)。

```

1  N = 4          # 设置量子比特数
2  DEPTH = 6     # 设置量子电路深度
3  theta = np.random.randn(DEPTH, N, 3)
4
5  # 启动 Paddle 动态图模式
6  with fluid.dygraph.guard():
7
8      # 我们需要将 Numpy array 转换成 Paddle 动态图模式中支持的 variable
9      theta = fluid.dygraph.to_variable(theta)
10
11     # 初始化量子电路
12     cir = UAnsatz(N)
13
14     # 添加深度为 D = 6 的复数强纠缠结构QNN {Rz+Ry+Rz/U3 + CNOT's}
15     cir.complex_entangled_layer(theta, DEPTH)
16
17     # 制备输出态
18     # 如果用户不输入初始量子态, 默认初始为 |00..0>
19     final_state = cir.run_state_vector()
20
21     # 测量输出态的[0, 1, 2]号量子比特2048次, 统计测量结果的频率
22     cir.measure(shots = 2048, which_qubits = [0, 1, 2], plot =
True)

```



(回到 [目录](#))

量浆的运行模式说明

波函数向量模式

所谓的波函数模式也就是用复数向量表示和储存量子态。向量模式只能处理纯态，但这种模式在家用电脑硬件高效支持**20+**量子比特的运算。用户可以测试下自己电脑的极限在哪里。在这种表示下，量子门(酉矩阵)作用在量子比特(一个复向量)上本质上的运算是**矩阵乘以向量**：

$$|\psi\rangle = U|\psi_0\rangle$$

代码中，具体体现在 UAnsatz 的调用 `cir.run_state_vector(input_state = None)`。如果我们不输入任何初始量子态，就会默认所有的量子比特都处于 $|0\rangle$ 态。接着来看个具体的例子：

```
1 from paddle_quantum.state import vec, vec_random
2
3 N = 20          # 设置量子比特数
4 DEPTH = 6      # 设置量子电路深度
5 theta = np.random.randn(DEPTH, N, 1)
6
7 # 调用内置的 |00..0> 初始态
8 initial_state1 = vec(N)
9 # 调用内置的随机量子态 |psi>
10 initial_state2 = vec_random(N)
11
12 # 启动 Paddle 动态图模式
13 with fluid.dygraph.guard():
14
15     # 我们需要将 Numpy array 转换成 Paddle 动态图模式中支持的 variable
16     theta = fluid.dygraph.to_variable(theta)
17     initial_state = fluid.dygraph.to_variable(initial_state1)
18
19     # 初始化量子电路
20     cir = UAnsatz(N)
21
22     # 添加深度为 Depth 的实数强纠缠结构QNN {Ry+CNOT's}
23     cir.real_entangled_layer(theta, DEPTH)
24
25     # 制备输出态
26     # 如果用户不输入初始量子态，默认初始为 |00..0>
27     final_state = cir.run_state_vector(initial_state)
28     print(final_state.numpy())
```

```
1 [[-0.00044827+0.j  0.0008469 +0.j -0.00103331+0.j ...
2    -0.00063708+0.j    0.00011647+0.j -0.00075266+0.j]]
```

密度矩阵模式

同时 Paddle quantum 也支持了密度矩阵运算模式，也就是用一个密度矩阵 $\rho = \sum_i P_i |\psi_i\rangle\langle\psi_i|$ 表示和储存量子态。该模式下可以根据算法需要支持混合态模拟。但是在密度矩阵模式下，家用电脑硬件只能运行10个左右的量子比特。请用户注意这方面的限制，我们也在不断优化这个模式下的模拟器性能。在这种表示下，量子门(酉矩阵)作用在量子态(一个迹为1的厄尔米特矩阵)上本质上的运算是矩阵乘法：

$$\rho = U\rho_0U^\dagger$$

代码中，具体体现在 UAnsatz 的调用 `cir.run_density_matrix()`。接着来看个具体的例子：

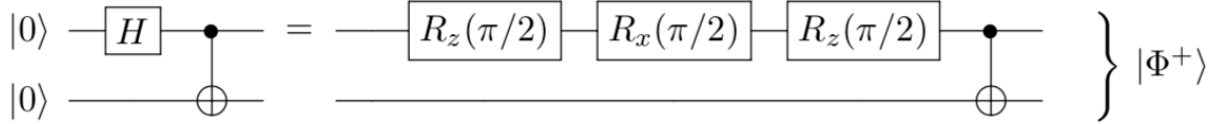
```
1 from paddle_quantum.state import density_op, density_op_random,
  completely_mixed_computational
2
3 N = 2          # 设置量子比特数
4 DEPTH = 6     # 设置量子电路深度
5 theta = np.random.randn(DEPTH, N, 1)
6
7 # 调用内置的 |00..0><00..0| 初始态
8 initial_state1 = density_op(N)
9 # 调用内置的随机量子态，可以指定是否允许复数元素和矩阵秩
10 initial_state2 = density_op_random(N, real_or_complex=2, rank=4)
11 # 调用内置的计算基下的完全混合态
12 initial_state3 = completely_mixed_computational(N)
13
14 # 启动 Paddle 动态图模式
15 with fluid.dygraph.guard():
16
17     # 我们需要将 Numpy array 转换成 Paddle 动态图模式中支持的 variable
18     theta = fluid.dygraph.to_variable(theta)
19     initial_state = fluid.dygraph.to_variable(initial_state1)
20
21     # 初始化量子电路
22     cir = UAnsatz(N)
23
24     # 添加深度为 Depth 的实数强纠缠结构QNN {Ry+CNOT's}
25     cir.real_entangled_layer(theta, DEPTH)
26
27     # 制备输出态
28     # 如果用户不输入初始量子态，默认初始为 |00..0><00..0|
29     final_state = cir.run_density_matrix(initial_state)
30     print(final_state.numpy())
```

练习：如何从计算基制备贝尔态

贝尔态是一种很常用的量子纠缠态，可以表示为

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

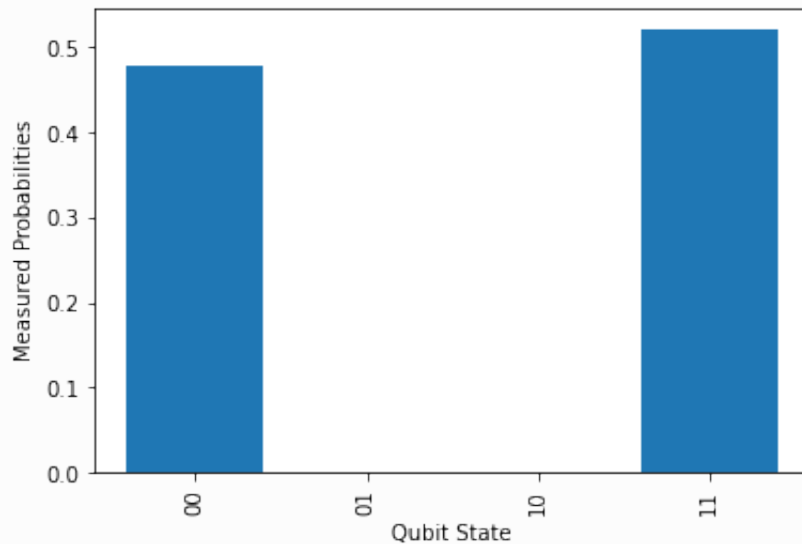
那么我们如何用量桨来制备一个贝尔态呢？只需要如下的量子电路：



```

1 # 启动 Paddle 动态图模式
2 with fluid.dygraph.guard():
3
4     # 初始化量子电路
5     cir = UAnsatz(2)
6
7     # 添加量子门
8     cir.h(0)
9     cir.cnot([0, 1])
10
11     # 如果用户不输入初始量子态，默认初始为 |00..0>
12     output_state = cir.run_state_vector()
13
14     # 我们测量输出态2048次，获得测量结果频率分布
15     cir.measure(shots = 2048, plot = True)
16
17     print('我们制备出的贝尔态是:\n', output_state.numpy())

```



```

1 我们制备出的贝尔态是：
2 [0.70710678+0.j 0.          +0.j 0.          +0.j 0.70710678+0.j]

```

(回到 [目录](#))

飞桨 PADDLEPADDLE 优化器使用教程

示例: 利用飞桨的梯度下降来优化多元函数

在这一节，我们学习如何用飞桨动态图机制找到一个多元函数的极小值

$$\min_{\theta} \mathcal{L}(\theta_1, \theta_2, \theta_3) = (\theta_1)^2 + (\theta_2)^2 + (\theta_3)^2 + 10$$

可以看出，只有当 $\theta_1 = \theta_2 = \theta_3 = 0$ 的时候， \mathcal{L} 取最小值10。

```
1 # 超参数设置
2 theta_size = 3
3 ITR = 200      # 设置迭代次数
4 LR = 0.5      # 设置学习速率
5 SEED = 1      # 固定随机数种子
6
7 class Optimization_ex1(fluid.dygraph.Layer):
8
9
10     def __init__(self, shape, param_attr=fluid.initializer.Uniform(
11         low=-5., high=5., seed=SEED), dtype='float64'):
12         super(Optimization_ex1, self).__init__()
13
14         # 初始化一个长度为 theta_size的可学习参数列表
15         # 并用 [-5, 5] 的均匀分布来填充初始值
16         self.theta = self.create_parameter(
17             shape=shape, attr=param_attr, dtype=dtype, is_bias=False)
18
19         # 定义损失函数和前向传播机制
20         def forward(self):
21             loss = self.theta[0] ** 2
22                 + self.theta[1] ** 2
23                 + self.theta[2] ** 2 + 10
24             return loss
25
26         # 记录中间优化结果
27         loss_list = []
28         parameter_list = []
29
30         # 初始化 paddle 动态图机制
31         with fluid.dygraph.guard():
32
33             # 定义网络维度
34             myLayer = Optimization_ex1([theta_size])
35
36             # 一般来说，我们利用Adam优化器来获得相对好的收敛
37             # 当然你可以改成SGD或者是RMSprop.
38             optimizer = fluid.optimizer.AdagradOptimizer(
39                 learning_rate = LR, parameter_list = myLayer.parameters())
```

```

40
41     # 优化循环
42     for itr in range(ITR):
43
44         # 向前传播计算损失函数
45         loss = myLayer()[0]
46
47         # 在动态图机制下，反向传播优化损失函数
48         loss.backward()
49         optimizer.minimize(loss)
50         myLayer.clear_gradients()
51
52         # 记录学习曲线
53         loss_list.append(loss.numpy()[0])
54         parameter_list.append(myLayer.parameters()[0].numpy())
55
56     print('损失函数的最小值是: ', loss_list[-1])

```

```

1  损失函数的最小值是:  10.000000153748474

```

练习: 特征值寻找

接下来，我们试一个更复杂的损失函数。首先我们介绍一个随机的埃尔米特矩阵 H 其特征值为矩阵 D 的对角元素。

$$D = \begin{bmatrix} 0.2 & 0 \\ 0 & 0.8 \end{bmatrix}$$

不用担心，我们会帮你生成这个埃尔米特矩阵 H 。

然后我们初始化参数向量 θ ，构造出一个简单的线性运算 $U(\theta) = R_z(\theta_1) * R_y(\theta_2) * R_z(\theta_3)$

$$U(\theta_1, \theta_2, \theta_3) = \begin{bmatrix} e^{-i\frac{\theta_1}{2}} & 0 \\ 0 & e^{i\frac{\theta_1}{2}} \end{bmatrix} \begin{bmatrix} \cos \frac{\theta_2}{2} & -\sin \frac{\theta_2}{2} \\ \sin \frac{\theta_2}{2} & \cos \frac{\theta_2}{2} \end{bmatrix} \begin{bmatrix} e^{-i\frac{\theta_3}{2}} & 0 \\ 0 & e^{i\frac{\theta_3}{2}} \end{bmatrix}$$

我们让这个矩阵（模板）乘以 $|0\rangle$ ，得到一个新的2维复向量 $|\phi\rangle$

$$|\phi\rangle = U(\theta_1, \theta_2, \theta_3)|0\rangle$$

然后，我们定义损失函数为：

$$\min_{\theta} \mathcal{L}(\theta_1, \theta_2, \theta_3) = \langle \phi | H | \phi \rangle = \langle 0 | U^\dagger H U | 0 \rangle$$

来看看优化后我们得到了什么！

```

1  from scipy.stats import unitary_group
2
3  # V 是一个 2x2 的随机酉矩阵
4  V = unitary_group.rvs(2)
5
6  # D 的对角元是H的特征值
7  # 你可以任意改变这里的对角元数值
8  D = np.diag([0.2, 0.8])
9
10 # V_dagger 是 V 的埃尔米特转置
11 V_dagger = V.conj().T
12
13 # @: 代表矩阵乘积运算
14 H = (V @ D @ V_dagger)
15 print('按照谱分解随机生成的矩阵 H 是:')
16 print(H, '\n')
17 print('不出所料, H 的特征值是:')
18 print(np.linalg.eigh(H)[0])

```

```

1  随机生成的矩阵 H 是:
2  [[0.45272355+0.00000000e+00j 0.08876211-2.82641516e-01j]
3   [0.08876211+2.82641516e-01j 0.54727645+1.38777878e-17j]]
4
5  不出所料, H 的特征值是:
6  [0.2 0.8]

```

```

1  # 超参数设置
2  theta_size = 3      # 设置 theta 维度
3  num_qubits = 1     # 设置量子比特数
4  ITR = 10           # 设置迭代次数
5  LR = 0.8           # 设置学习速率
6  SEED = 1           # 固定theta参数的随机数种子
7
8
9  # 单独设置电路模块
10 def U_theta(theta):
11
12     # 初始化电路然后添加量子门
13     cir = UAnsatz(num_qubits)
14     cir.rz(theta[0], 0)
15     cir.ry(theta[1], 0)
16     cir.rz(theta[2], 0)
17
18     # 返回参数化矩阵
19     return cir.U

```

```

1 class Optimization_ex2(fluid.dygraph.Layer):
2
3
4     def __init__(self, shape, param_attr=fluid.initializer.Uniform(
5         low=0., high=2*np.pi, seed=SEED), dtype='float64'):
6         super(Optimization_ex2, self).__init__()
7
8         # 初始化一个长度为 theta_size的可学习参数列表
9         # 并用 [0, 2*pi] 的均匀分布来填充初始值
10        self.theta = self.create_parameter(
11            shape=shape, attr=param_attr, dtype=dtype, is_bias=False)
12        self.H = fluid.dygraph.to_variable(H)
13
14        # 定义损失函数和前向传播机制
15        def forward(self):
16
17            # 获取量子神经网络的酉矩阵表示
18            U = U_theta(self.theta)
19
20            # 埃尔米特转置运算
21            U_dagger = dagger(U)
22
23            # 计算损失函数函数
24            loss = matmul(U_dagger, matmul(self.H, U)).real[0][0]
25
26            return loss

```

```

1 loss_list = []
2 parameter_list = []
3
4
5 # 初始化paddle动态图机制
6 with fluid.dygraph.guard():
7
8     myLayer = Optimization_ex2([theta_size])
9
10    # 一般来说, 我们利用Adam优化器来获得相对好的收敛
11    # 当然你可以改成SGD或者是RMS prop.
12    optimizer = fluid.optimizer.AdagradOptimizer(
13        learning_rate = LR, parameter_list = myLayer.parameters())
14
15    # 优化循环
16    for itr in range(ITR):
17
18        # 前向传播计算损失函数
19        loss = myLayer()[0]
20
21        # 在动态图机制下, 反向传播极小化损失函数
22        loss.backward()

```

```
23     optimizer.minimize(loss)
24     myLayer.clear_gradients()
25
26     # 记录学习曲线
27     loss_list.append(loss.numpy()[0])
28     parameter_list.append(myLayer.parameters()[0].numpy())
29     print('iter:', itr, ' loss: %.4f' % loss.numpy())
30
31     print('损失函数的最小值是: ', loss_list[-1])
```

```
1  iter: 0    loss: 0.2773
2  iter: 1    loss: 0.2750
3  iter: 2    loss: 0.2010
4  iter: 3    loss: 0.2000
5  iter: 4    loss: 0.2000
6  iter: 5    loss: 0.2000
7  iter: 6    loss: 0.2000
8  iter: 7    loss: 0.2000
9  iter: 8    loss: 0.2000
10 iter: 9    loss: 0.2000
11 损失函数的最小值是:  0.2000000000000055
```

我们可以改变一下 H 的特征值。如果将它对角化后的的对角矩阵改变为

$$D = \begin{bmatrix} 0.8 & 0 \\ 0 & 1.2 \end{bmatrix}$$

你会发现我们仍然得到了 H 的最小特征值0.8, 你能找到背后的原因吗? 还是说这背后隐藏着什么理论?

(回到 [目录](#))

量子机器学习案例

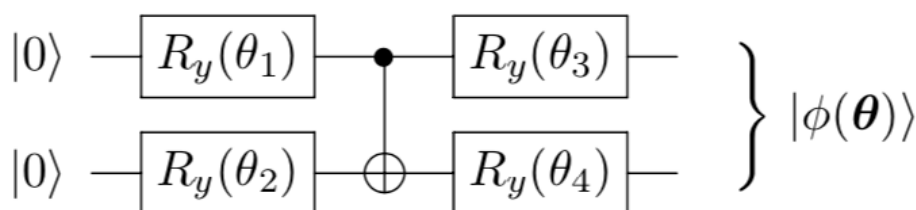
变分量子特征求解器 (VQE) -- 无监督学习

目前阶段，大规模的可容错的量子计算机还未实现。我们目前只能造出有噪音的，中等规模量子计算系统 (NISQ)。现在一个利用 NISQ 的量子设备很有前景的算法种类就是量子-经典混合算法。人们期望这套方法也许可以在某些应用中超越经典计算机的表现。变分量子特征求解器 (VQE) 就是里面的一个重要应用。它利用参数化的电路搜寻广阔的希尔伯特空间，并利用经典机器学习中的梯度下降来找到最优参数，并接近一个哈密顿量的基态（也就是找到一个埃尔米特矩阵的最小特征值）。为了确保你能理解，我们来一起过一遍以下两量子比特 (2-qubit) 的例子。

假设我们想找到如下哈密顿量的基态：

$$H = 0.4 Z \otimes I + 0.4 I \otimes Z + 0.2 X \otimes X$$

给定一种常见的量子神经网络架构



我们已经学会如何建造这个电路了。如果你忘了，请转到 [这里](#)。

```
1 from paddle_quantum.utils import pauli_str_to_matrix
2
3 # 首先生成泡利字符串表示下的哈密顿量
4 # 相当于0.4*kron(I, Z) + 0.4*kron(Z, I) + 0.2*kron(X, X)
5 # 其中, x, y, z是泡利矩阵, I是单位矩阵
6 H_info = [[0.4, 'z0'], [0.4, 'z1'], [0.2, 'x0,x1']]
7
8 # 超参数设置
9 num_qubits = 2
10 theta_size = 4
11 ITR = 10
12 LR = 0.5
13 SEED = 1
14
15 # 把记录的关于哈密顿量的信息转化为矩阵表示
16 H_matrix = pauli_str_to_matrix(H_info, num_qubits)
```

```

1 class vqe_demo(fluid.dygraph.Layer):
2
3
4     def __init__(self, shape, param_attr=fluid.initializer.Uniform(
5         low=0., high=2*np.pi, seed=SEED), dtype='float64'):
6         super(vqe_demo, self).__init__()
7
8         # 初始化一个长度为theta_size的可学习参数列表
9         # 并用[0, 2*pi]的均匀分布来填充初始值
10        self.theta = self.create_parameter(
11            shape=shape, attr=param_attr, dtype=dtype, is_bias=False)
12        self.H = fluid.dygraph.to_variable(H)
13
14        # 定义损失函数和前向传播机制
15        def forward(self):
16
17            # 初始量子电路
18            cir = UAnsatz(num_qubits)
19
20            # 添加量子门
21            cir.ry(self.theta[0], 0)
22            cir.ry(self.theta[1], 1)
23            cir.cnot([0, 1])
24            cir.ry(self.theta[2], 0)
25            cir.ry(self.theta[3], 1)
26
27            # 选择用量子态的向量表示
28            cir.run_state_vector()
29
30            # 计算当前量子态下关于观测量H_info的期望值
31            # 也就是 <psi|H|psi>
32            loss = cir.expecval(H_info)
33
34            return loss

```

```

1 loss_list = []
2 parameter_list = []
3
4
5 # 初始化paddle动态图机制
6 with fluid.dygraph.guard():
7
8     # 定义网络维度
9     vqe = vqe_demo([theta_size])
10
11    # 一般来说, 我们利用Adam优化器来获得相对好的收敛
12    # 当然你可以改成SGD或者是RMS prop.
13    optimizer = fluid.optimizer.AdagradOptimizer(
14        learning_rate = LR, parameter_list = vqe.parameters())

```

```

15
16     # 优化循环
17     for itr in range(ITR):
18
19         # 前向传播计算损失函数
20         loss = vqe()
21
22         # 在动态图机制下, 反向传播极小化损失函数
23         loss.backward()
24         optimizer.minimize(loss)
25         vqe.clear_gradients()
26
27         # 记录学习曲线
28         loss_list.append(loss.numpy()[0])
29         parameter_list.append(vqe.parameters()[0].numpy())
30         print('iter:', itr, '  loss: %.4f' % loss.numpy())
31
32
33     print('计算得到的基态能量是: ', loss_list[-1])
34     print('真实的基态能量为: ', np.linalg.eigh(H_matrix)[0][0])

```

```

1  iter: 0    loss: 0.3195
2  iter: 1    loss: -0.3615
3  iter: 2    loss: -0.6528
4  iter: 3    loss: -0.7420
5  iter: 4    loss: -0.7813
6  iter: 5    loss: -0.8022
7  iter: 6    loss: -0.8130
8  iter: 7    loss: -0.8187
9  iter: 8    loss: -0.8216
10 iter: 9    loss: -0.8231
11 计算得到的基态能量是:  -0.8230721264692568
12 真实的基态能量为:  -0.8246211251235321

```

(回到 [目录](#))

参考文献

- (1) Peruzzo, A. et al. A variational eigenvalue solver on a photonic quantum processor. *Nat. Commun.* 5, 4213 (2014).
- (2) McClean, J. R., Romero, J., Babbush, R. & Aspuru-Guzik, A. The theory of variational hybrid quantum-classical algorithms. *New J. Phys.* 18, 023023 (2016).
- (3) Kandala, A. et al. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature* 549, 242–246 (2017).
- (4) Mitarai, K., Negoro, M., Kitagawa, M. & Fujii, K. Quantum circuit learning. *Phys. Rev. A* 98, 032309 (2018).

([回到 目录](#))