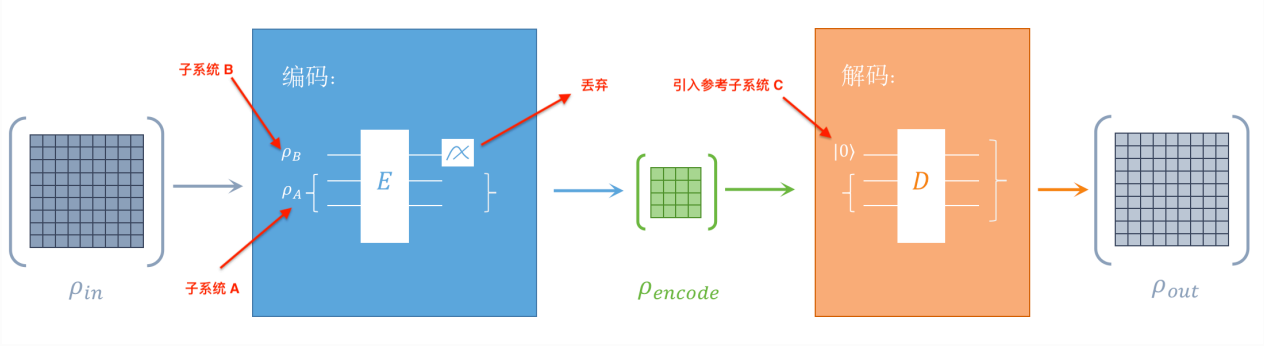
量子变分自编码器

(QUANTUM AUTOENCODER)

概览

在这个案例中，我们将展示如何训练量子自编码器来压缩和重建给定的量子态（混合态）[1]。量子自编码器的形式与经典自编码器非常相似，同样由编码器 E（encoder）和解码器 D（decoder）组成。对于输入的 N 量子比特系统的量子态 ρ_{in}（这里我们采用量子力学的密度算符表示来描述混合态），先用编码器 $E=U(\theta)$ 将信息编码到其中的部分量子比特上。这部分量子比特记为**系统** A。对剩余的量子比特 （这部分记为**系统** B）进行测量并丢弃后，我们就得到了压缩后的量子态 ρ_{encode}！压缩后的量子态维度和量子系统 A 的维度大小保持一致。假设我们需要 N_A 个量子比特来描述系统 A，那么编码后量子态 ρ_{encode} 的维度就是 $2^{N_A}\times 2^{N_A}$。 这里比较特殊的是，对应我们这一步测量并丢弃的操作的数学操作是 partial trace。读者在直观上可以理解为张量积 $\otimes$ 的逆运算。

我们不妨来看一个具体的例子来理解：
给定一个由 N_A 个量子比特描述的量子态 ρ_A 和另外一个由 N_B 个量子比特描述的量子态 ρ_B，那么由 A、B 两个子系统构成的的整体量子系统 $N=N_A+N_B$ 的量子态就可以描述为: $\rho_{AB}=\rho_A\otimes\rho_B$。现在我们让整个量子系统在酉矩阵 U 的作用下演化一段时间后，得到了一个新的量子态 $\rho_{\tilde{A}B}=U\rho_{AB}U^\dagger$。 那么如果这时候我们只想得到量子子系统 A 所处于的新的量子态 $\tilde{\rho_A}$， 我们该怎么做呢？很简单，只需要测量量子子系统 B 然后丢弃测量结果。运算上这一步由 partial trace 来完成 $\tilde{\rho_A}=\mathrm{Tr}_B(\rho_{\tilde{A}B})$。在量桨上，我们可以用内置的 `partial_trace(rho_AB, 2**N_A, 2**N_B, 2)` 指令来完成这一运算。**注意：** 其中最后一个输入为2，表示我们想丢弃量子系统 B 的量子态。



在讨论完编码的过程后，我们来看看如何解码。为了解码量子态 ρ_{encode}，我们需要引入与系统 B 维度相同的系统 C 并且初始态取为全0态。再用解码器 $D=U^{\dagger}(\theta)$ 作用在整个量子系统 $A+C$ 上对系统 A 中压缩的信息进行解码。我们希望最后得到的量子态 ρ_{out} 与 ρ_{in} 尽可能相似并用 Uhlmann-Josza 保真度 F （Fidelity）来衡量他们之间的相似度。

$$F(\rho_{in},\rho_{out})=\left(\mathrm{tr}\sqrt{\sqrt{\rho_{in}}\rho_{out}\sqrt{\rho_{in}}}\right)^2$$

最后，通过优化编码器里的参数，我们就可以尽可能地提高 ρ_{in} 与 ρ_{out} 的保真度。这里我们先引入必要的 package。

```
import numpy as np
from numpy import diag
import scipy

import paddle
from paddle import fluid
from paddle_quantum.circuit import UAnsatz
from paddle.complex import matmul, trace, kron
from paddle_quantum.utils import hermitian, state_fidelity,
partial_trace
```

生成初始态

下面我们通过一个简单的例子来展示量子自编码器的工作流程和原理。

我们考虑 $N=3$ 个量子比特上的量子态 ρ_{in}。我们先通过编码器将其信息编码到下方的两个量子比特（系统 A），之后对第一个量子比特（系统 B）测量并丢弃，之后引入一个处于0态的量子比特（新的参考系统 C）来替代丢弃的量子比特 B 的维度。最后通过解码器，将 A 中压缩的信息复原成 ρ_{out}。在这里，我们假设初态是一个混合态并且 ρ_{in} 的特征谱为 $\lambda_i\in\{0.4,\ 0.2,\ 0.2,\ 0.1,\ 0.1,\ 0,\ 0,\ 0\}$，然后通过作用一个随机的酉变换来生成初态 ρ_{in}。

```
N_A = 2         # 系统 A 的量子比特数
N_B = 1         # 系统 B 的量子比特数
N = N_A + N_B   # 总的量子比特数

scipy.random.seed(1)                                  # 固定随机种子
V = scipy.stats.unitary_group.rvs(2**N)               # 随机生成一个酉矩阵
D = diag([0.4, 0.2, 0.2, 0.1, 0.1, 0, 0, 0])          # 输入目标态rho的谱
V_H = V.conj().T                                      # 进行厄尔米特转置
rho_in = (V @ D @ V_H).astype('complex128')           # 生成 rho_in

# 将 C 量子系统初始化为
rho_C = np.diag([1,0]).astype('complex128')
```

搭建量子神经网络

在这里，我们用量子神经网络来作为编码器和解码器。假设系统 A 有 N_A 个量子比特，系统 B 和 C 分别有N_B 个量子比特，量子神经网络的深度为 D。编码器 E 作用在系统 A 和 B 共同构成的总系统上，解码器 D 作用在 A 和 C 共同构成的总系统上。在我们的问题里，$N_A = 2$，$N_B = 1$。

```
# 设置电路参数
cir_depth = 6                               # 电路深度
block_len = 2                               # 每个模组的长度
theta_size = N*block_len*cir_depth    # 网络参数 theta 的大小


# 搭建编码器 Encoder E
def Encoder(theta):

    # 用 UAnsatz 初始化网络
    cir = UAnsatz(N)

    # 搭建层级结构：
    for layer_num in range(cir_depth):

        for which_qubit in range(N):
            cir.ry(theta[block_len*layer_num*N + which_qubit],
                                                    which_qubit)
            cir.rz(theta[(block_len*layer_num + 1)*N
                                + which_qubit], which_qubit)

        for which_qubit in range(N-1):
            cir.cnot([which_qubit, which_qubit + 1])
        cir.cnot([N-1, 0])

    return cir.U
```

配置训练模型-损失函数

在这里，我们定义的损失函数为

$$Loss = 1 - \langle 0...0|\rho_{trash}|0...0\rangle$$

其中 ρ_{trash} 是经过编码后丢弃的 B 系统的量子态。接着我们通过飞桨的自动微分框架来训练量子神经网络，最小化损失函数。如果损失函数最后达到 0，则输入态和输出态完全相同。这就意味着我们完美地实现了压缩和解压缩，这时初态和末态的保真度为 $F(\rho_{in}, \rho_{out}) = 1$。

```python
# 超参数设置
N_A = 2          # 系统 A 的量子比特数
N_B = 1          # 系统 B 的量子比特数
N = N_A + N_B    # 总的量子比特数
LR = 0.2         # 设置学习速率
ITR = 100        # 设置迭代次数
SEED = 14        # 固定初始化参数用的随机数种子

class NET(fluid.dygraph.Layer):
    """
    Construct the model net
    """
    def __init__(self, shape, param_attr=fluid.initializer.Uniform(
        low=0.0, high=2 * np.pi, seed = SEED), dtype='float64'):
        super(NET, self).__init__()

        # 将 Numpy array 转换成 Paddle 动态图模式中支持的 variable
        self.rho_in = fluid.dygraph.to_variable(rho_in)
        self.rho_C = fluid.dygraph.to_variable(rho_C)
        self.theta = self.create_parameter(shape=shape,
                     attr=param_attr, dtype=dtype, is_bias=False)

    # 定义损失函数和前向传播机制
    def forward(self):

        # 生成初始的编码器 E 和解码器 D
        E = Encoder(self.theta)
        E_dagger = hermitian(E)
        D = E_dagger
        D_dagger = E

        # 编码量子态 rho_in
        rho_BA = matmul(matmul(E, self.rho_in), E_dagger)

        # 取 partial_trace() 获得 rho_encode 与 rho_trash
        rho_encode = partial_trace(rho_BA, 2 ** N_B, 2 ** N_A, 1)
        rho_trash = partial_trace(rho_BA, 2 ** N_B, 2 ** N_A, 2)

        # 解码得到量子态 rho_out
        rho_CA = kron(self.rho_C, rho_encode)
        rho_out = matmul(matmul(D, rho_CA), D_dagger)

        # 通过 rho_trash 计算损失函数
        zero_Hamiltonian = fluid.dygraph.to_variable(
                           np.diag([1,0]).astype('complex128'))
        loss = 1 - (trace(matmul(zero_Hamiltonian,
                                 rho_trash))).real

        return loss, self.rho_in, rho_out


# 初始化paddle动态图机制
```

```
with fluid.dygraph.guard():

    # 生成网络
    net = NET([theta_size])

    # 一般来说，我们利用Adam优化器来获得相对好的收敛
    # 当然你可以改成SGD或者是RMS prop.
    opt = fluid.optimizer.AdagradOptimizer(
          learning_rate=LR, parameter_list=net.parameters())

    # 优化循环
    for itr in range(1, ITR + 1):

        #  前向传播计算损失函数
        loss, rho_in, rho_out = net()

        # 在动态图机制下，反向传播极小化损失函数
        loss.backward()
        opt.minimize(loss)
        net.clear_gradients()

        # 计算并打印保真度
        fid = state_fidelity(rho_in.numpy(), rho_out.numpy())

        if itr % 10 == 0:
            print('iter:', itr, 'loss:', '%.4f' % loss, 'fid:',
'%.4f' % np.square(fid))
```

```
iter: 10 loss: 0.1203 fid: 0.8751
iter: 20 loss: 0.1060 fid: 0.8889
iter: 30 loss: 0.1043 fid: 0.8908
iter: 40 loss: 0.1037 fid: 0.8916
iter: 50 loss: 0.1034 fid: 0.8921
iter: 60 loss: 0.1032 fid: 0.8925
iter: 70 loss: 0.1030 fid: 0.8929
iter: 80 loss: 0.1027 fid: 0.8932
iter: 90 loss: 0.1025 fid: 0.8935
iter: 100 loss: 0.1022 fid: 0.8939
```

总结

如果系统 A 的维度为 d_A，容易证明量子自编码器能实现的压缩重建最大保真度为 ρ_{in} 最大的 d_A 个本征值之和。在我们的示例里 $d_A = 4$，最大保真度为

$$F_{limit} = 0.4 + 0.2 + 0.2 + 0.1 = 0.9$$

通过 100 次迭代，我们训练出的量子自编码器保真度达到 0.89 以上，和最优情况非常接近。

参考文献

(1) Romero, J., Olson, J. P. & Aspuru-Guzik, A. Quantum autoencoders for efficient compression of quantum data. Quantum Sci. Technol. 2, 045001 (2017).