PQC Function Evaluation

Weeks 1-3

David Amorim

01/07/2024 - 19/07/2024

Table of Contents

Background

- Hayes 2023[1] presents a scheme to encode a complex vector $\boldsymbol{h} = \{\tilde{A}_j e^{i\Psi(j)} | 0 \leq j < N\}$ as the state

$$|h\rangle = \frac{1}{|\tilde{A}|} \sum_{j=0}^{2^n-1} \tilde{A}(j) e^{i\Psi(j)} |j\rangle, \tag{1}$$

  using $n = \lceil \log_2 N \rceil$ qubits
- This requires operators $\hat{U}_A$ and $\hat{U}_\Psi$ such that

$$\hat{U}_A |0\rangle^{\otimes n} = \frac{1}{|\tilde{A}|} \sum_{j=0}^{2^n-1} \tilde{A}(j) |j\rangle, \tag{2}$$

$$\hat{U}_\Psi |j\rangle = e^{i\Psi(j)} |j\rangle \tag{3}$$

[1] https://arxiv.org/pdf/2306.11073

Background

- $\hat{U}_\Psi$ is constructed via an operator $\hat{Q}_\Psi$ that performs function evaluation in an ancilla register:

$$\hat{Q}_\Psi \left|j\right\rangle \left|0\right\rangle_a^{\otimes m} = \left|j\right\rangle \left|\Psi'(j)\right\rangle_a , \tag{4}$$

  with $\Psi'(j) \equiv \Psi(j)/2\pi$
- Currently, $\hat{Q}_\Psi$ is implemented using gate-intensive *linear piecewise functions (LPFs)*

Background

Aim

Implement $\hat{Q}_\Psi$ in a gate-efficient way using a parametrised quantum circuit (PQC)

Remark

The n-qubit register containing the $|j\rangle$ and the m-qubit register containing the $|\Psi'(j)\rangle$ will be referred to as the input register and target register, respectively.

Approach: a QCNN

- A *quantum convolutional neural network* (QCNN) is used to tackle the problem
- A QCNN is a parametrised quantum circuit involving multiple layers
- Two types of network layers are implemented:
  - Convolutional layers (CL) involve multi-qubit entanglement gates
  - Input layers (IL)[2] involve controlled single-qubit operations on target qubits
- Input qubits only appear as controls throughout the QCNN

[2] Replacing the conventional QCNN *pooling layers*

Convolutional Layers (CLs)

- Each CL involves the cascaded application of a two-qubit operator on the target register
- A general two-qubit operator involves 15 parameters
- To reduce the parameter space, the canonical three-parameter operator

$$\mathcal{N}(\alpha, \beta, \gamma) = \exp\left(i\left[\alpha X \otimes X + \beta Y \otimes Y + \gamma Z \otimes Z\right]\right) \tag{5}$$

  is applied, at the cost of restricting the search space
- This can be decomposed[3] into 3 CX, 3 R_z, and 2 R_y gates
- A two-parameter real version, $\mathcal{N}_{\mathbb{R}}(\lambda, \mu)$, can be obtained by removing the R_z

[3] https://arxiv.org/pdf/quant-ph/0308006

Convolutional Layers (CLs)

- Two types of convolutional layers are implemented:
  - Neighbour-to-neighbour / linear CLs: the $\mathcal{N}$ (or $\mathcal{N}_{\mathbb{R}}$) gate is applied to neighbouring target qubits
  - All-to-all /quadratic CLs: the $\mathcal{N}$ (or $\mathcal{N}_{\mathbb{R}}$) gate is applied to all combinations of target qubits
- The $\mathcal{N}$-gate cost of neighbour-to-neighbour (NN) layers is $\mathcal{O}(m)$ while that of all-to-all (AA) layers is $\mathcal{O}(m^2)$
- Currently, the QCNN uses alternating linear and quadratic CLs

Input Layers (ILs)

- ILs, replacing pooling layers, feed information about the input register into the target register
- An IL involves a sequence of controlled generic single-qubit rotations ($CU3$ gates) on the target qubits, with input qubits as controls
- For an IL producing states with real amplitudes, the $CU3$ gates are replaced with CR_y gates
- Each input qubit controls precisely one $CU3$ (or CR_y operation), resulting in an $\mathcal{O}(n)$ gate cost (no CX gates!)
- ILs are inserted after every second convolutional layer, alternating between control states 0 and 1

Summary: QCNN Structure

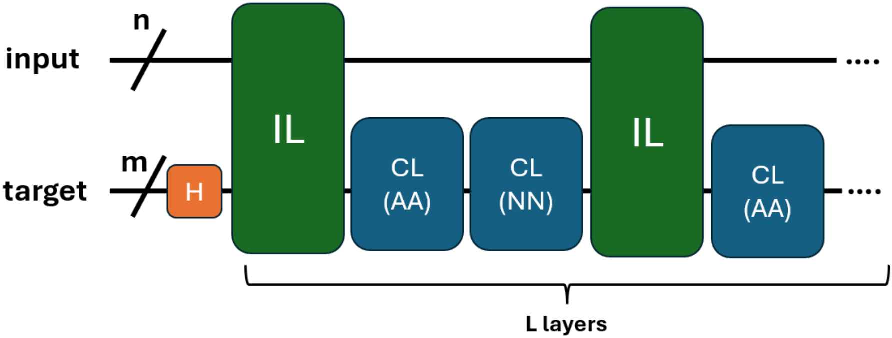

Figure 1: Schematic of QCNN structure

Training the QCNN

- For training, the QCNN is wrapped as a *SamplerQNN* object and connected to PyTorch's Adam optimiser via *TorchConnector*
- The optimiser determines improved parameter values for each training run (epoch) based on the loss between output and target state
- Beyond loss, mismatch is an important metric:

$$\mathrm{M} = 1 - |\langle \psi_{\text{target}} | \psi_{\text{out}} \rangle| \tag{6}$$

- There are two ways to train the QCNN on input data:[4]
  1. Training on individual states
  2. Training in superposition

[4] One can also train the QCNN to produce a target distribution independent of the input register, which is equivalent to constructing $\hat{U}_A$

Training the QCNN

1. Training on Individual States

- One of the 2^n input states, $|j\rangle$, is randomly chosen each epoch
- The network is taught to transform $|j\rangle|0\rangle \mapsto |j\rangle|\Psi'(j)\rangle$ for each of the states individually

2. Training in Superposition

- The same input state is chosen each epoch
- The network is taught to transform

$$\left(\frac{1}{\sqrt{2^n}}\sum_{j=0}^{2^n-1}|j\rangle\right)|0\rangle \mapsto \frac{1}{\sqrt{2^n}}\sum_{j=0}^{2^n-1}|j\rangle|\Psi'(j)\rangle \tag{7}$$

- By linearity, this teaches the network to transform $|j\rangle|0\rangle \mapsto |j\rangle|\Psi'(j)\rangle$ for each $|j\rangle$

Initial Tests

- Initial tests need to be carried out to inform QCNN design choices regarding:
  a. Number of layers
  b. Number of epochs
  c. Training mode (individually versus in superposition)
  d. Use of $\mathcal{N}$ and $CU3$ versus $\mathcal{N}_{\mathbb{R}}$ and R_y
  e. Choice of loss function
  f. Network structure
- The case $n = m = 2$, $\Psi(x) = x$ (the simplest non-trivial configuration) is an ideal benchmark problem
- Unclear, however, how well these findings extrapolate to more general cases