TorchQuantum and QuantumNAS

Hanrui Wang
MIT HAN Lab

2022/01/18 @ Meta AI

Outline

- Quantum Basics
- Quantum Neural Networks for Machine Learning
- TorchQuantum
- QuantumNAS

Outline

- Quantum Basics
- Quantum Neural Networks for Machine Learning
- TorchQuantum
- QuantumNAS

Quantum Bit

- Quantum Bit (Qubit)
  - Statevector: contains 2^n complex numbers for n qubit system
  - The square sum of magnitude of 2^n numbers are 1
  - 1 qubit:

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \quad a_0, a_1 \in \mathbb{C}$$
$$|a_0|^2 + |a_1|^2 = 1$$

  - 2 qubits:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad a_0, a_1, a_2, a_3 \in \mathbb{C}$$
$$|a_0|^2 + |a_1|^2 + |a_2|^2 + |a_3|^2 = 1$$

Quantum Bit

- Classical bits represented in statevector
  - Classical 0: $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$
  - Classical 1: $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$
  - An arbitrary quantum states: $\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = a_0 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} + a_1 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

Quantum Gates

- Qubit gates: operations on one qubit or multiple qubits
- The qubit gates can be represented with matrix format with dimension $2^n \times 2^n$
- All gate matrices are unitary matrices: the conjugate transpose is the same as its inverse
- Single qubit gates:
  - Not (X) gate:

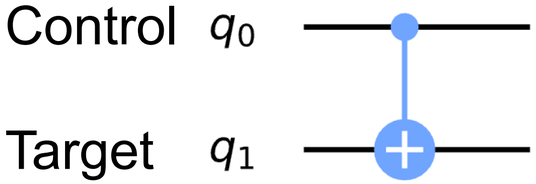
$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

  - Parameterized gate: Rotation X (RX) with parameter theta

$$RX(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$$

Quantum Gates

- 2-qubit gates:
  - Controlled Not (CNOT) gate:



$$CNOT = \begin{array}{c} \\ 00 \\ 01 \\ 10 \\ 11 \end{array} \begin{array}{c} \begin{array}{cccc} \text{Output} & & & \\ 00 & 01 & 10 & 11 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

  - Controlled Rotation X (CRX) gate

$$CRX(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ 0 & 0 & -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$$

Quantum Gates

- Applying a gate to qubits is performing matrix-vector multiplication between the gate matrix and statevector
  - Apply an X gate to classical state 0, we get 1

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

  - Apply an CNOT gate to state 10, we get 11

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

PyTorch Implementations

- Statevector

```
_state = torch.zeros(2 ** self.n_wires, dtype=C_DTYPE)
_state[0] = 1 + 0j
```

- Quantum Gates

```
'cnot': torch.tensor([[1, 0, 0, 0],
                      [0, 1, 0, 0],
                      [0, 0, 0, 1],
                      [0, 0, 1, 0]], dtype=C_DTYPE),
```

PyTorch Implementations

- Quantum Gates

```
def crx_matrix(params):
    theta = params.type(C_DTYPE)
    co = torch.cos(theta / 2)
    jsi = 1j * torch.sin(-theta / 2)

    matrix = torch.tensor([[1, 0, 0, 0],
                           [0, 1, 0, 0],
                           [0, 0, 0, 0],
                           [0, 0, 0, 0]], dtype=C_DTYPE, device=params.device
                          ).unsqueeze(0).repeat(co.shape[0], 1, 1)
    matrix[:, 2, 2] = co[:, 0]
    matrix[:, 2, 3] = jsi[:, 0]
    matrix[:, 3, 2] = jsi[:, 0]
    matrix[:, 3, 3] = co[:, 0]

    return matrix.squeeze(0)
```

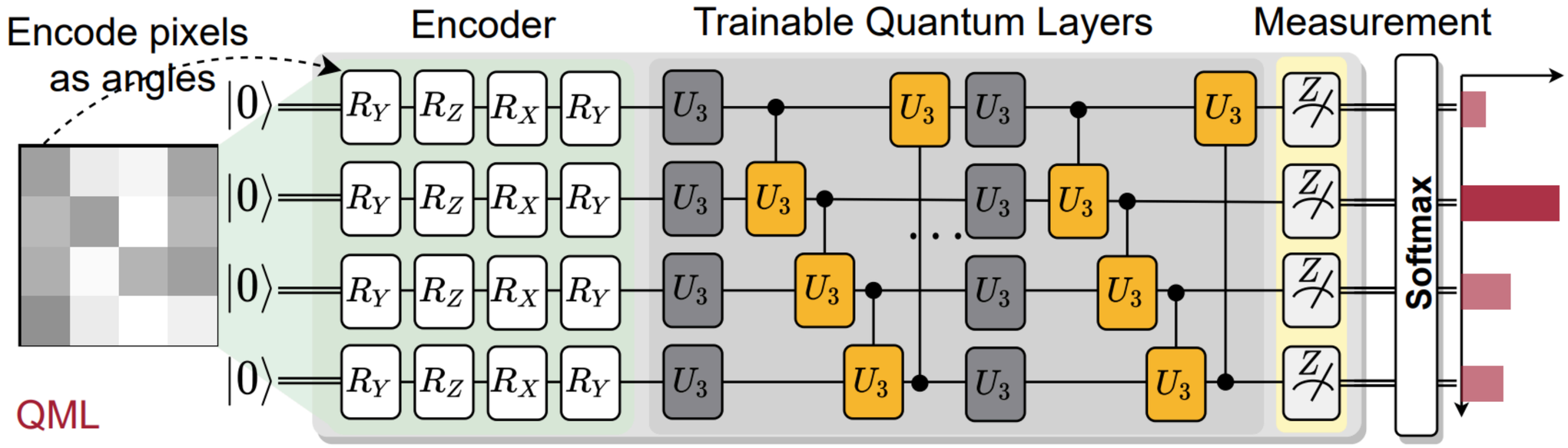
- Matrix-vector multiplication: torch.einsum and torch.bmm

Outline

- Quantum Basics
- Quantum Neural Networks for Machine Learning
- TorchQuantum
- QuantumNAS

Quantum Neural Networks (QNN)

- One way to perform machine learning task on quantum machine (others include quantum kernel methods etc.)
  - Encoder: encode the input data into quantum states
  - Parameterized quantum layers: containing learnable parameters
  - Measurement: obtain classical results



Wang, Hanrui, et al. "Quantumnas: Noise-adaptive search for robust quantum circuits." ***HPCA* (2022)**.

Essences of QNN and Why use it?

- Complex number
- Unitary matrix
- Encoding and measurement provide non-linearity
- Two possible sources of quantum advantage of QNN:
  - Complex number potentially provide more capacity over real number NN
  - The number of parameters and equivalent operations can be exponentially large
    - Parameter: For n qubit, a gate can contains $2^n \times 2^n$ parameters
    - Operation: Applying a gate is equivalent to perform $2^n \times 2^n$ multiplications and additions

Outline

- Quantum Basics
- Quantum Neural Networks for Machine Learning
- TorchQuantum
- QuantumNAS

TorchQuantum

- All the statevectors and quantum gates are implemented with PyTorch native operators
- Leverage PyTorch Autograd to train the parameters
- Batch mode training and inference on CPU/GPU
- Provide convertors from PyTorch to other frameworks such as IBM Qiskit for deployment

MNIST Example

Initialize a quantum device

Specify encoder gates

Specify trainable gates

```
import torch.nn as nn
import torch.nn.functional as F
import torchquantum as tq
import torchquantum.functional as tqf

class QFCModel(nn.Module):
  def __init__(self):
    super().__init__()
    self.n_wires = 4
    self.q_device = tq.QuantumDevice(n_wires=self.n_wires)
    self.measure = tq.MeasureAll(tq.PauliZ)

    self.encoder_gates = [tqf.rx] * 4 + [tqf.ry] * 4 + \
                         [tqf.rz] * 4 + [tqf.rx] * 4
    self.rx0 = tq.RX(has_params=True, trainable=True)
    self.ry0 = tq.RY(has_params=True, trainable=True)
    self.rz0 = tq.RZ(has_params=True, trainable=True)
    self.crx0 = tq.CRX(has_params=True, trainable=True)
```

MNIST Example

Reset statevector

Encode classical pixels

Apply the trainable gates

Apply some non-trainable gates

Measure to get classical values

```
def forward(self, x):
  bsz = x.shape[0]
  # down-sample the image
  x = F.avg_pool2d(x, 6).view(bsz, 16)

  # reset qubit states
  self.q_device.reset_states(bsz)

  # encode the classical image to quantum domain
  for k, gate in enumerate(self.encoder_gates):
    gate(self.q_device, wires=k % self.n_wires, params=x[:, k])

  # add some trainable gates (need to instantiate ahead of time)
  self.rx0(self.q_device, wires=0)
  self.ry0(self.q_device, wires=1)
  self.rz0(self.q_device, wires=3)
  self.crx0(self.q_device, wires=[0, 2])

  # add some more non-parameterized gates (add on-the-fly)
  tqf.hadamard(self.q_device, wires=3)
  tqf.sx(self.q_device, wires=2)
  tqf.cnot(self.q_device, wires=[3, 0])
  tqf.qubitunitary(self.q_device0, wires=[1, 2], params=[[1, 0, 0, 0],
                                                         [0, 1, 0, 0],
                                                         [0, 0, 0, 1j],
                                                         [0, 0, -1j, 0]])

  # perform measurement to get expectations (back to classical domain)
  x = self.measure(self.q_device).reshape(bsz, 2, 2)

  # classification
  x = x.sum(-1).squeeze()
  x = F.log_softmax(x, dim=1)

  return x
```

MNIST Example

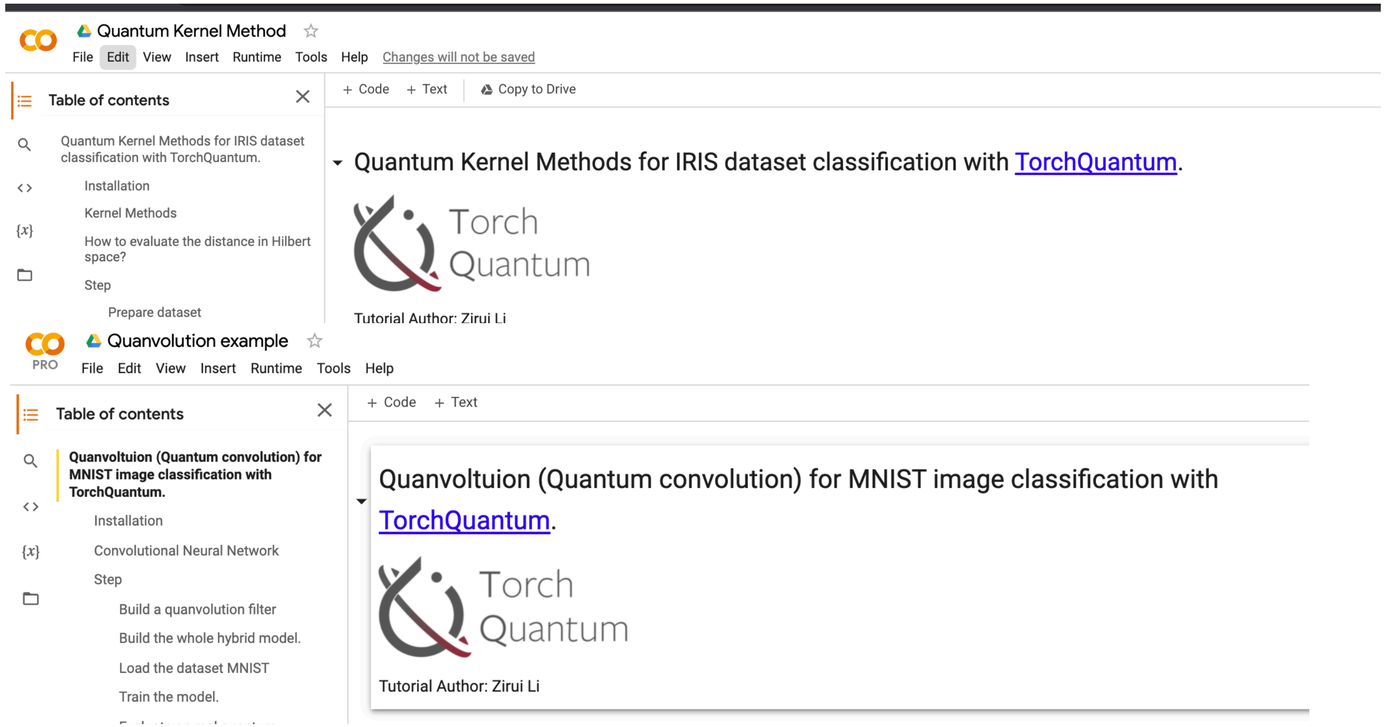
Initialize a QiskitProcessor

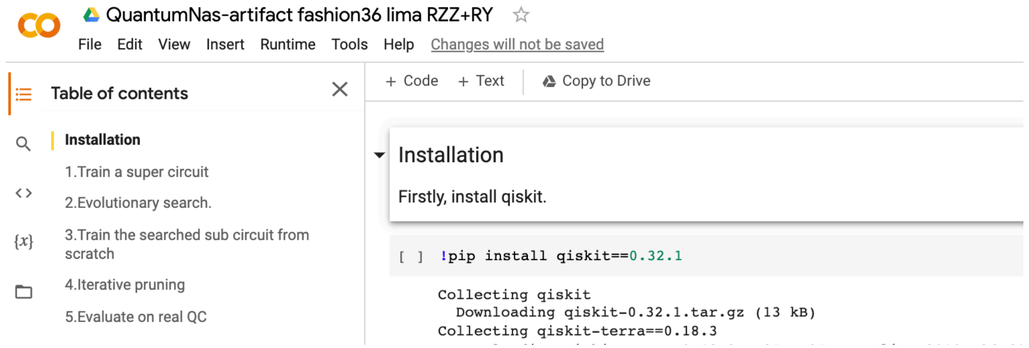
The ‘forward’ will be run on real QC

```
# then try to run on REAL QC
backend_name = 'ibmqx2'
print(f"\nTest on Real Quantum Computer {backend_name}")
processor_real_qc = QiskitProcessor(use_real_qc=True,
                                    backend_name=backend_name)
model.set_qiskit_processor(processor_real_qc)
valid_test(dataflow, 'test', model, device, qiskit=True)
```

TorchQuantum

- We provide many Colab tutorials
- Support other backend quantum machines
- Integration to PyTorch?



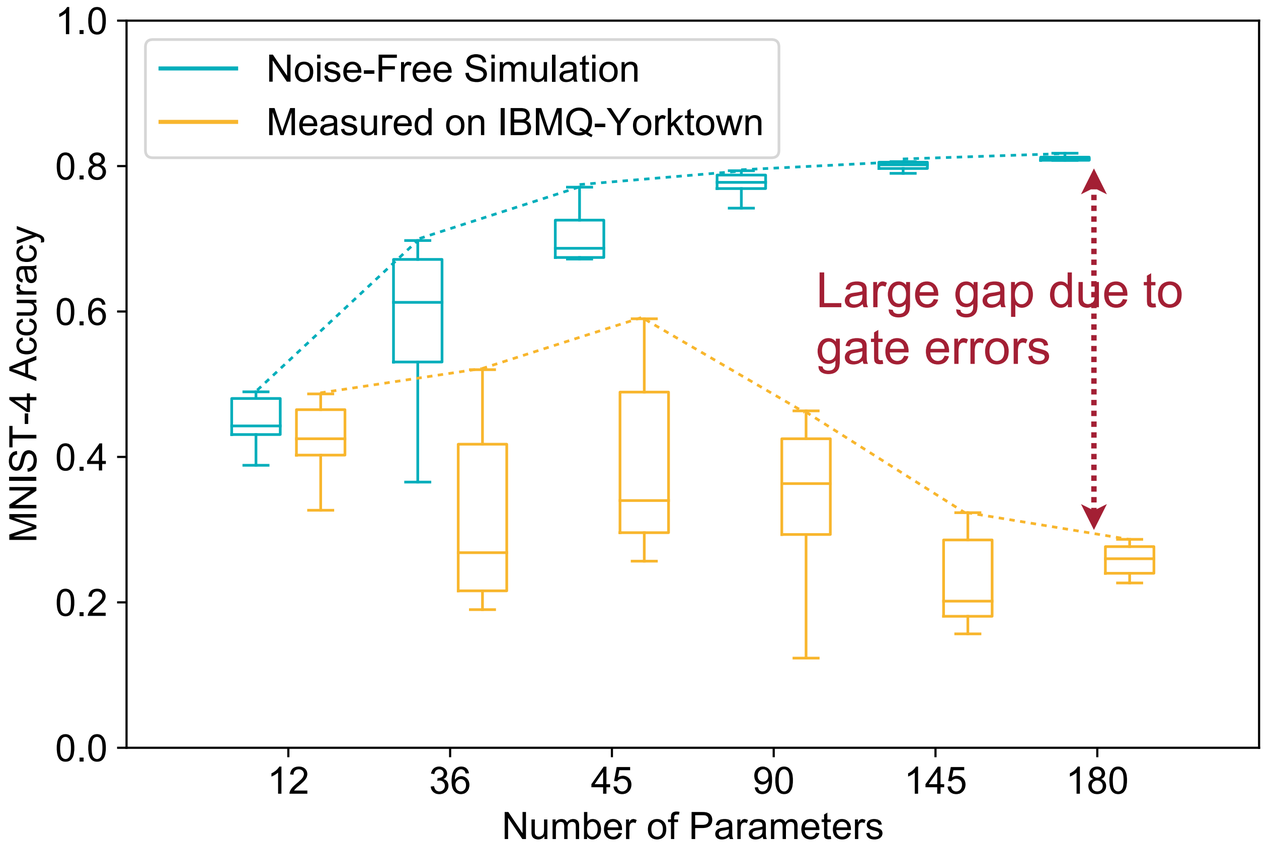


Outline

- Quantum Basics
- Quantum Neural Networks for Machine Learning
- TorchQuantum
- QuantumNAS

QuantumNAS with TorchQuantum (HPCA'22)

- Motivation
  - Large gap between the noise-free accuracy and real deployment accuracy.



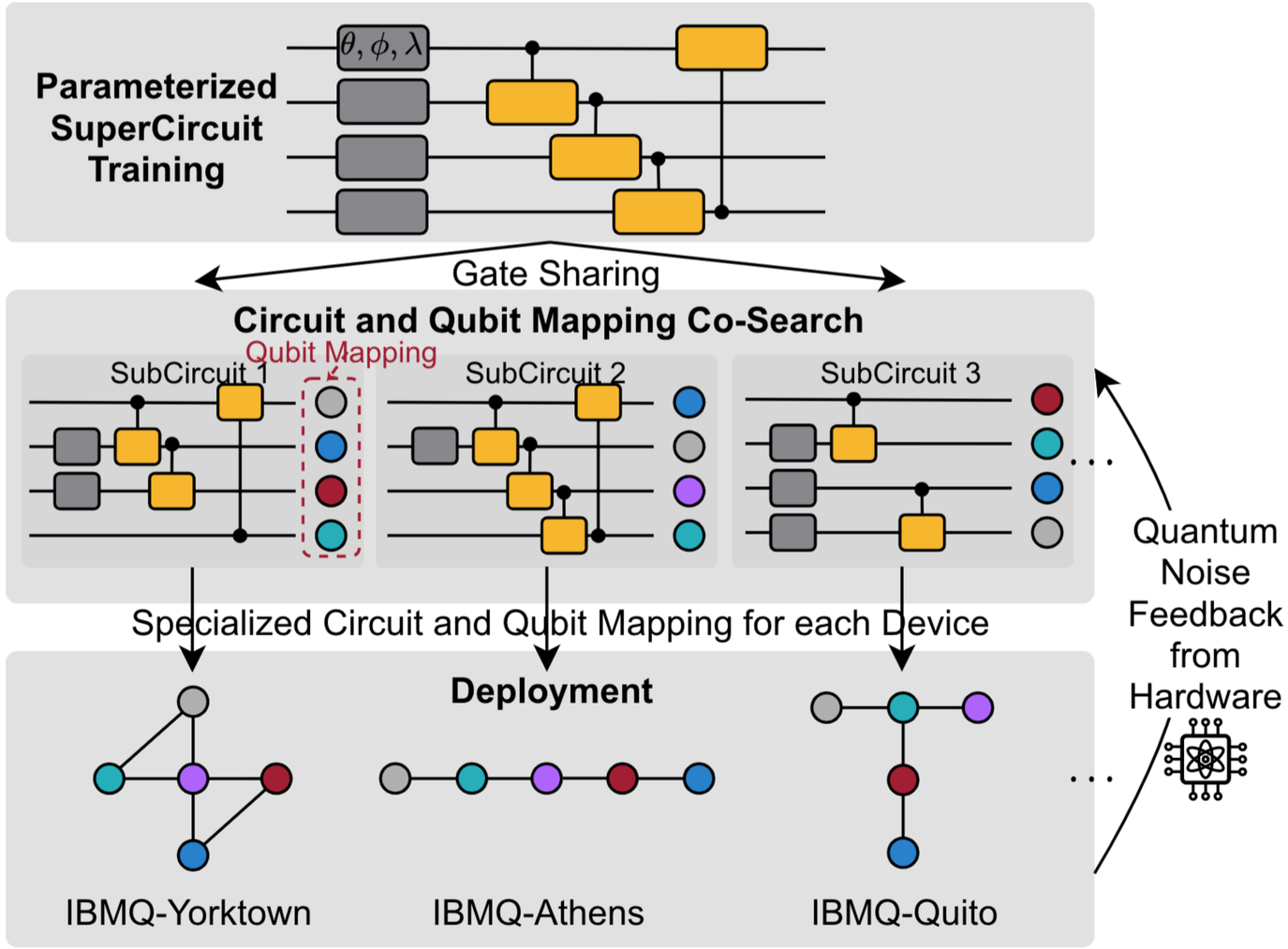
Torch Quantum

Wang, Hanrui, et al. "Quantumnas: Noise-adaptive search for robust quantum circuits." ***HPCA* (2022)**.

HANLAB

Neural Architecture Search (NAS) for QNN

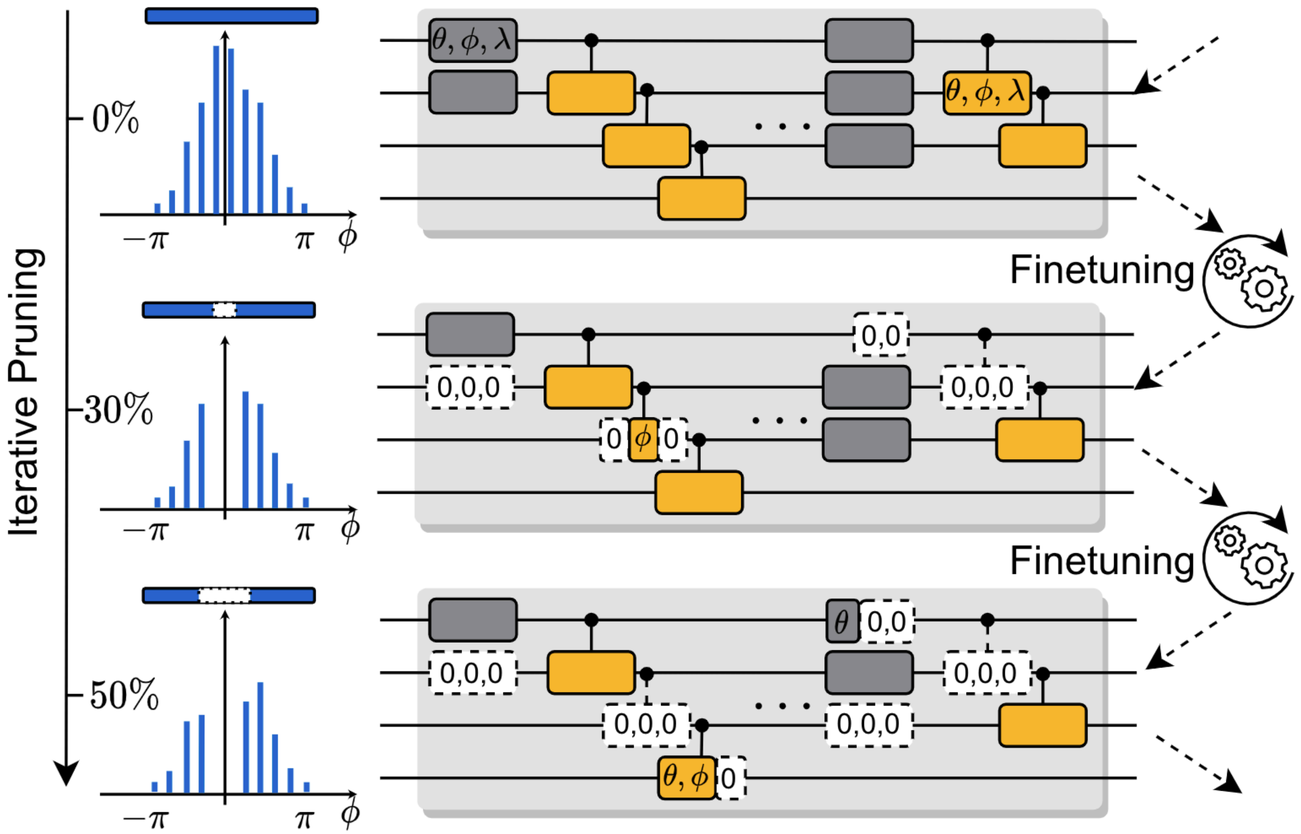
- We borrow the idea of NAS from the classical machine learning community
- Make QNN architecture **hardware-aware**
  - Quantum noise aware
  - Find the most robust QNN architecture
- SuperCircuit based method
  - Train a **SuperCircuit**
  - Search for robust **SubCircuits** inside SuperCircuit
    - Together with **qubit mapping**
  - Train the SubCircuit and deploy on **real** IBM machines



Wang, Hanrui, et al. "Quantumnas: Noise-adaptive search for robust quantum circuits." ***HPCA* (2022)**.

QNN Pruning

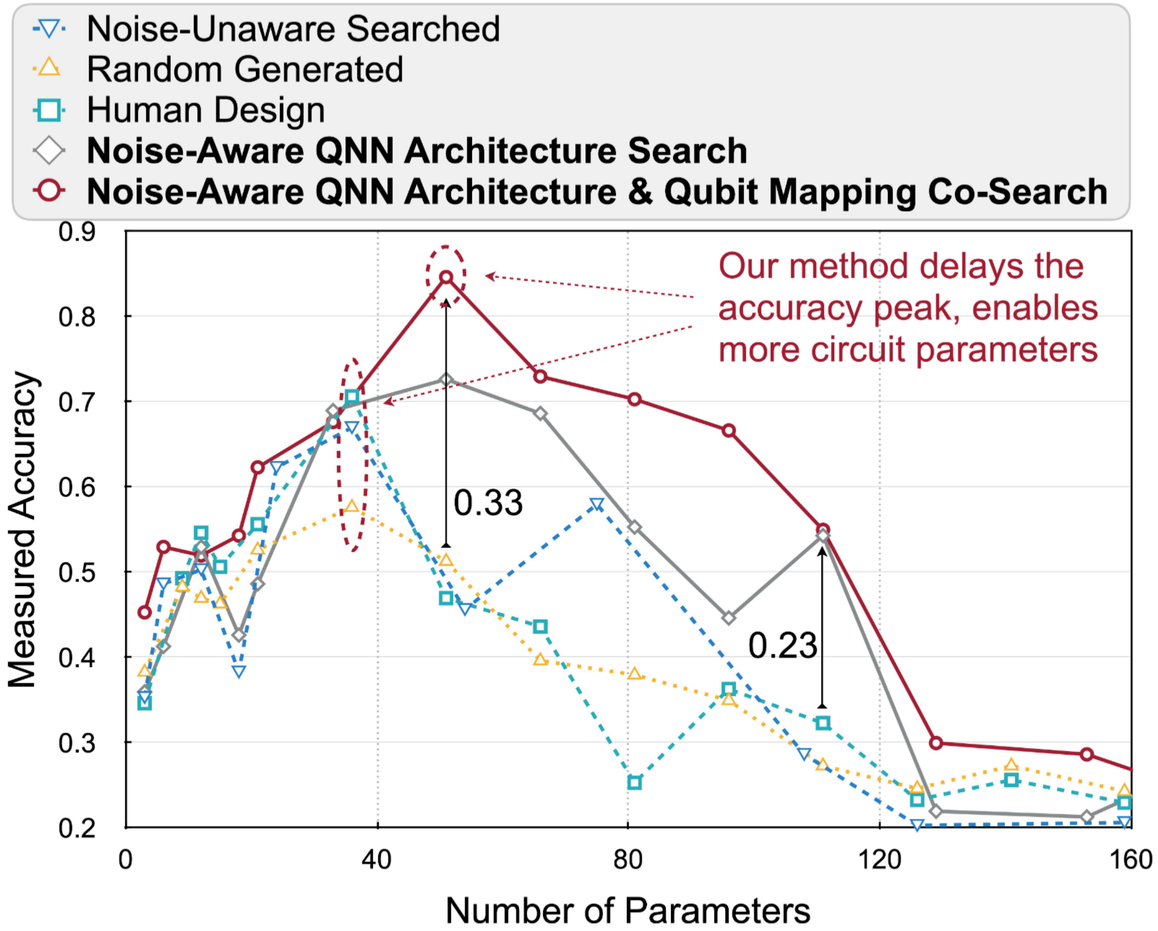
- Some parameters are close to zero
  - Remove small parameters
  - fine-tune the QNN model
- With fine-tuning, we can recover the noise-free accuracy.
- Since we **remove some gates**, there are less noise sources, the real deployment accuracy can be improved.



Wang, Hanrui, et al. "Quantumnas: Noise-adaptive search for robust quantum circuits." ***HPCA* (2022)**.

Experiment Results

- QNN architecture search and pruning
- We can **delay the accuracy peak** and enable more parameters



Wang, Hanrui, et al. "Quantumnas: Noise-adaptive search for robust quantum circuits." ***HPCA* (2022)**.

Thank you for listening!

Torch Quantum

https://github.com/mit-han-lab/torchquantum