



Torch  
Quantum

# TorchQuantum Tutorial

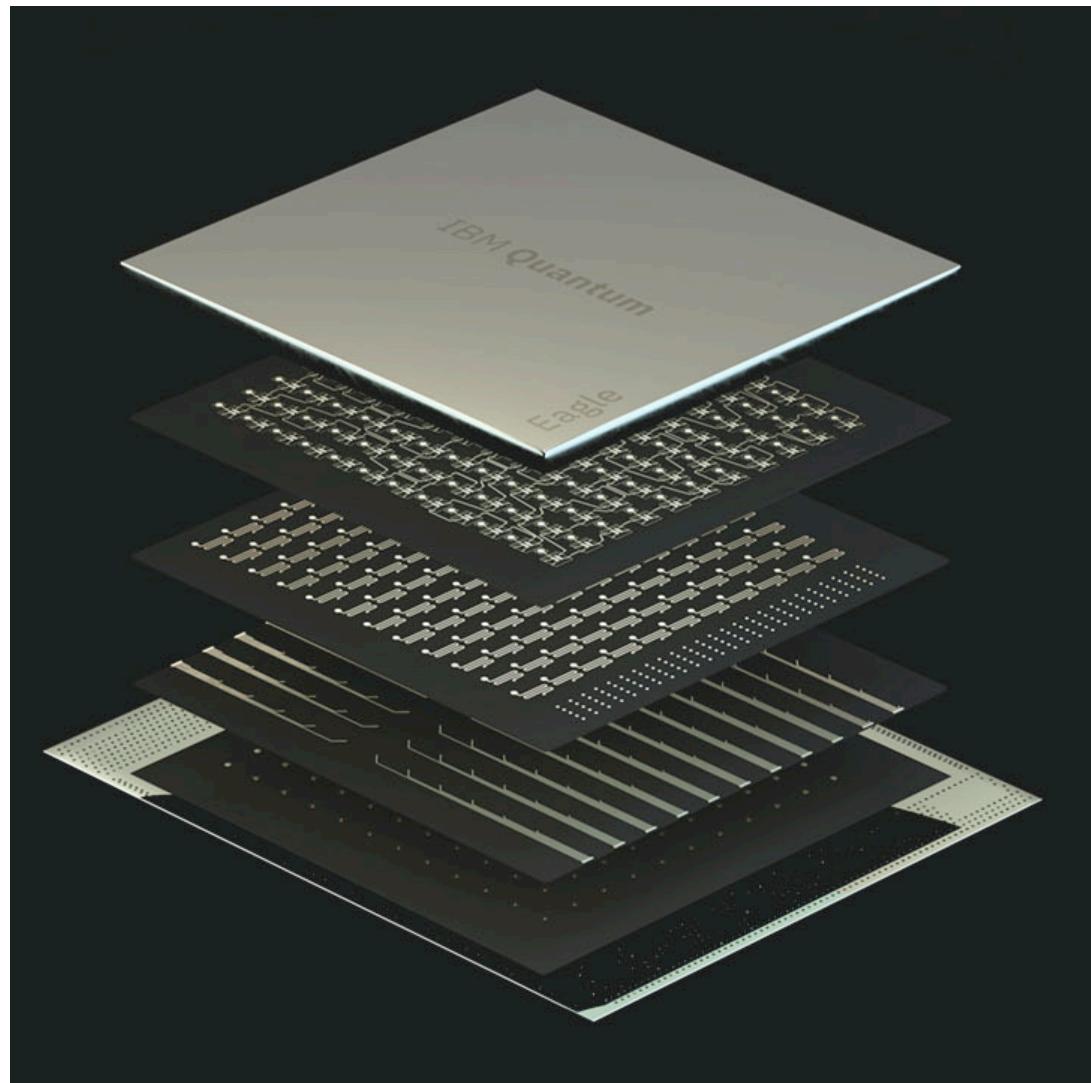
Speakers:

Hanrui Wang, Song Han (MIT)  
Zhidong Liang, Yiyu Shi (Notre Dame)  
Jinglei Cheng, Xuehai Qian (Purdue)  
Zhepeng Wang, Weiwen Jiang (GMU)

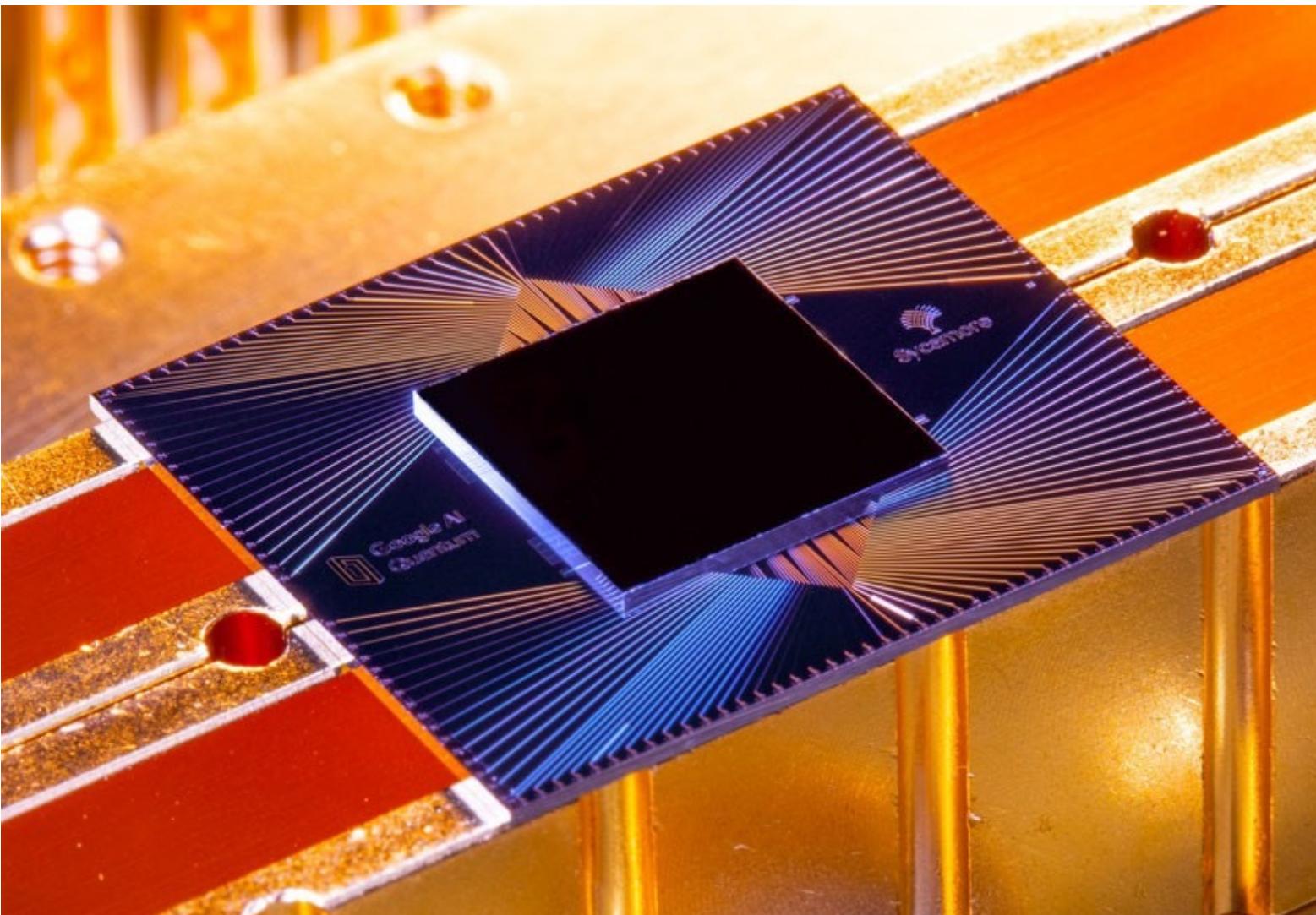
<https://torchquantum.org>

# Quantum Computing

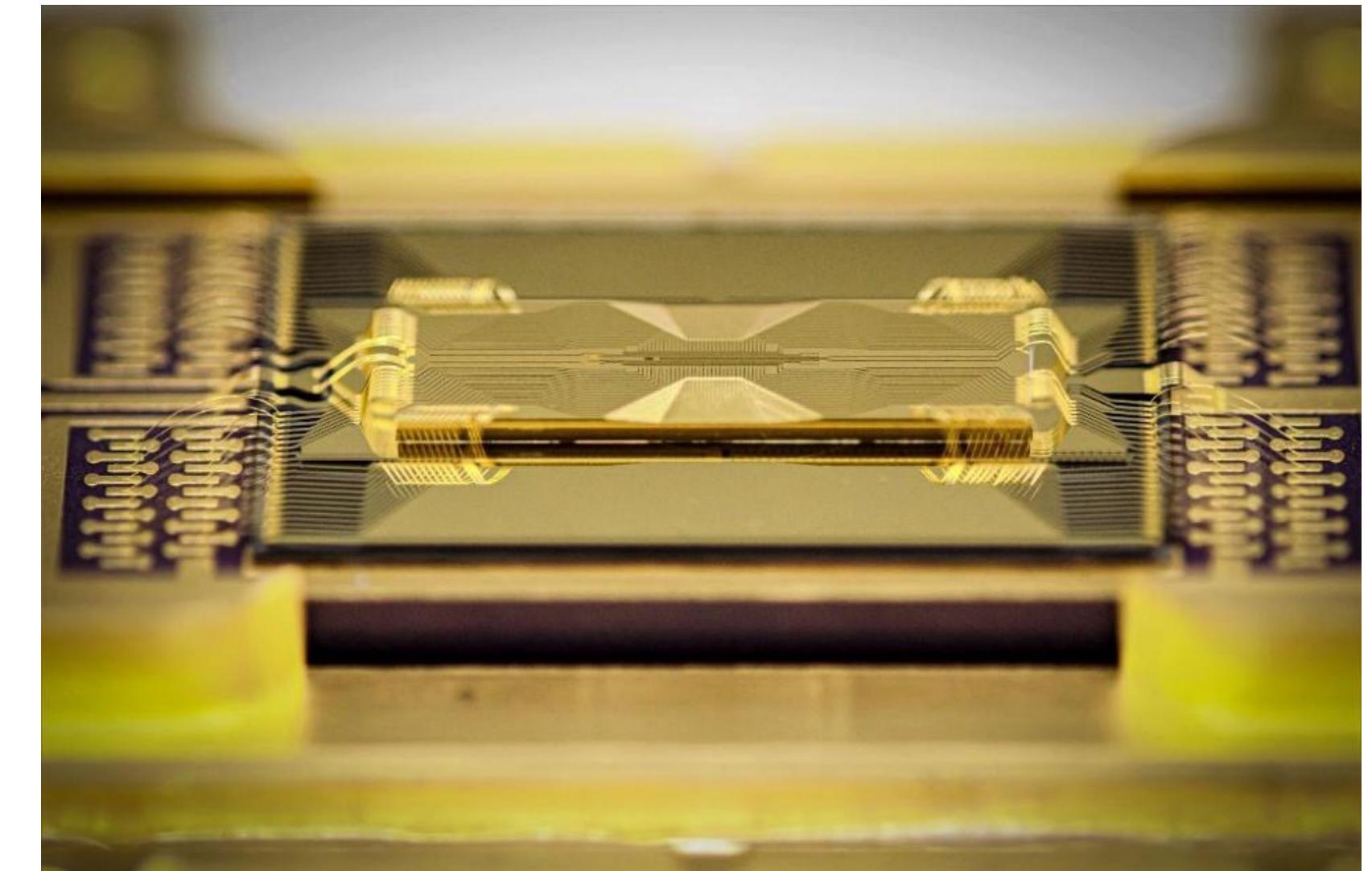
- Fast progress of quantum devices
- Different technologies
  - Superconducting, trapped ion, neutral atom, photonics, etc.



**IBM**  
**127 Qubit**



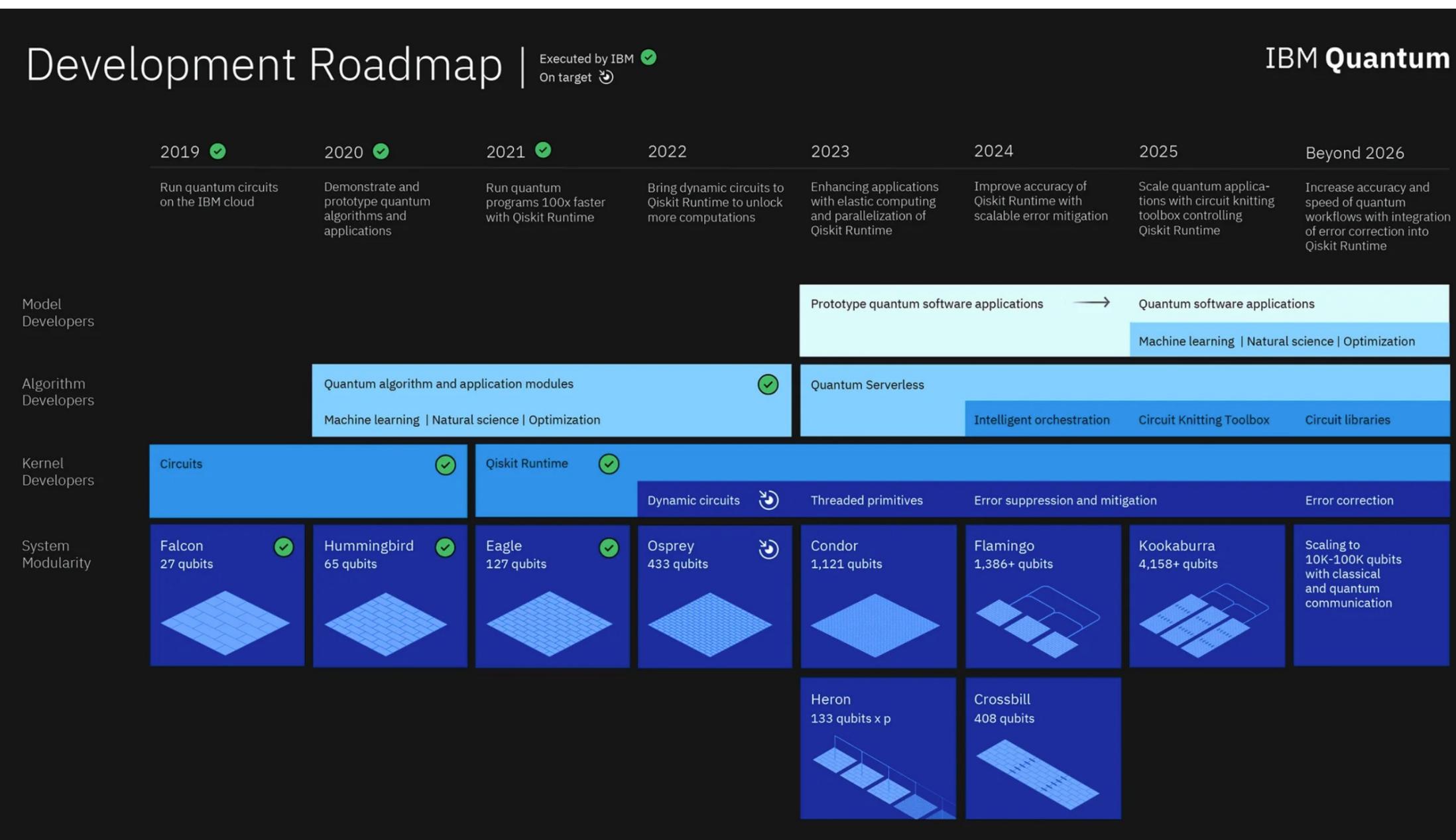
**Google**  
**53 Qubits**



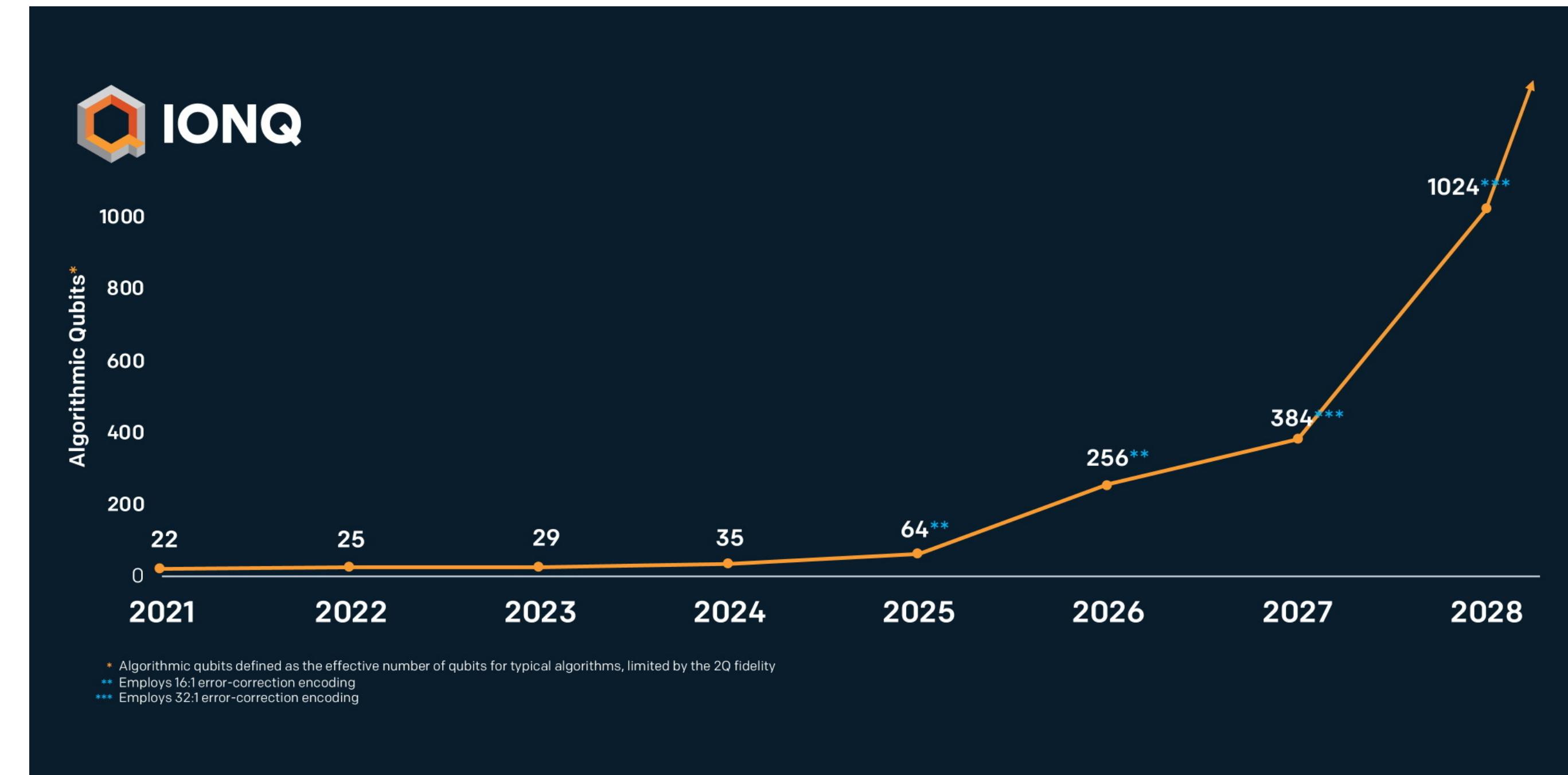
**IonQ**  
**32+ Qubits**

# Quantum Computing

- Double exponentially increasing computing power



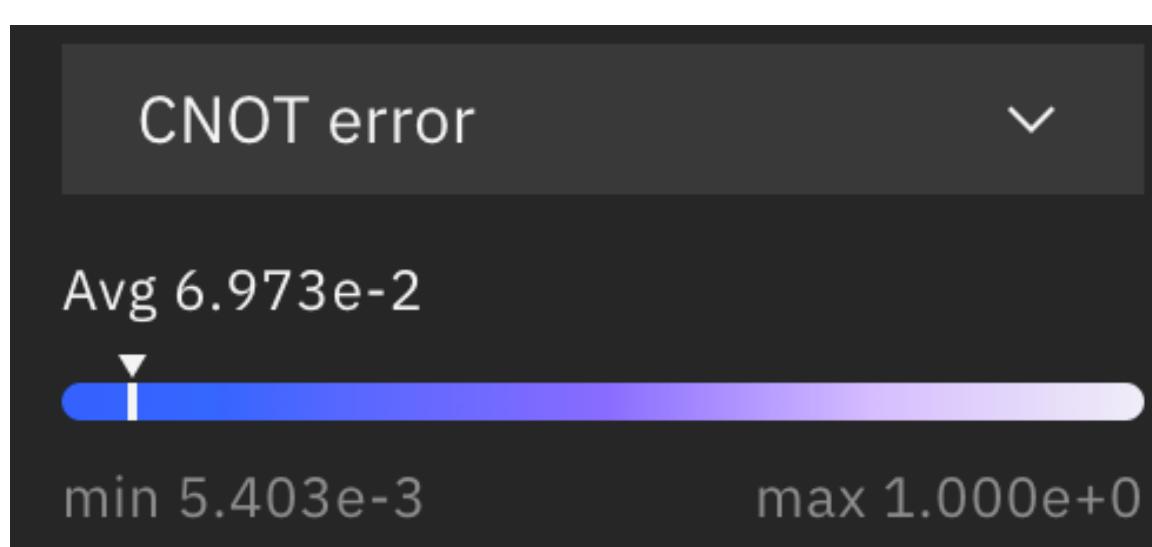
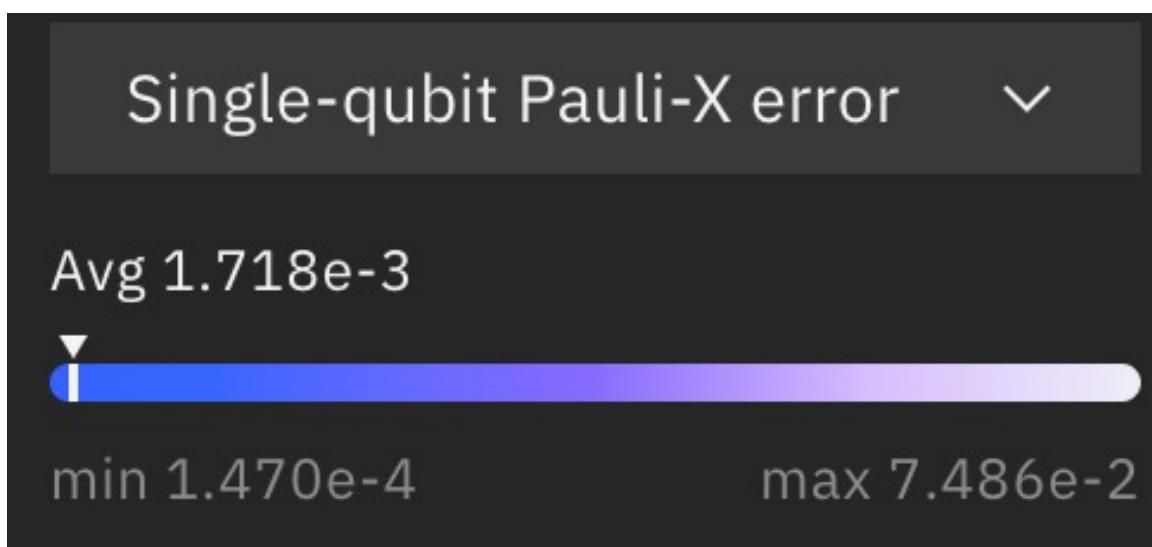
**IBM Roadmap**



**IonQ Roadmap**

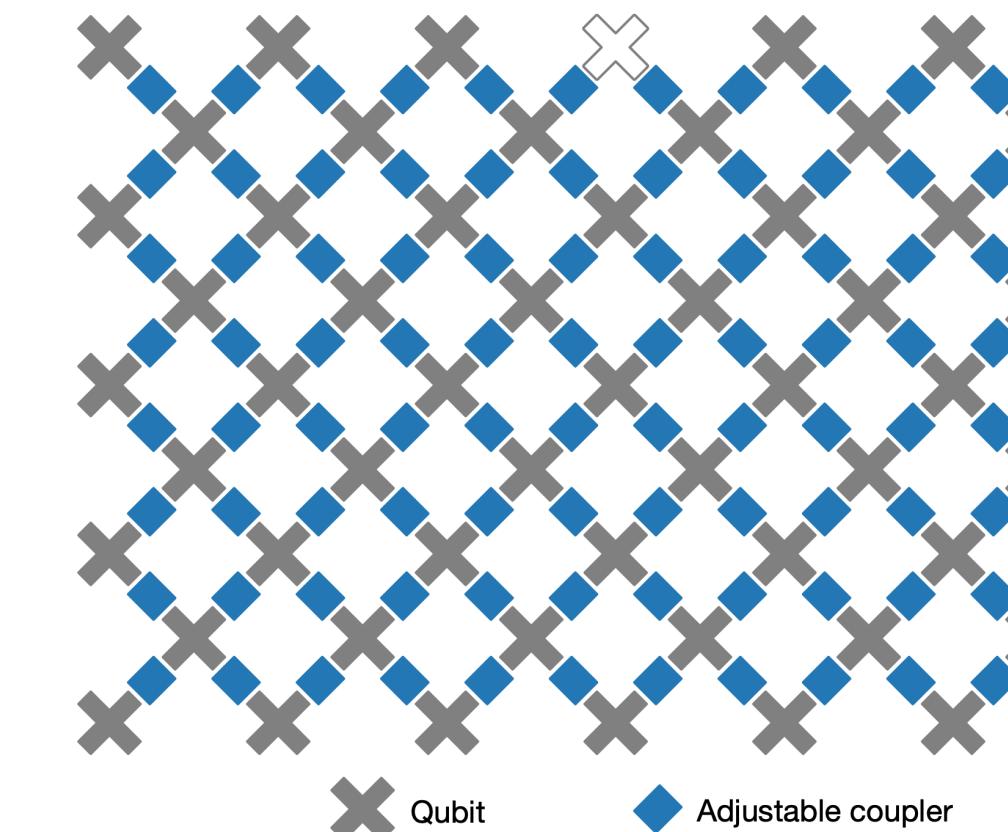
# Quantum Computing in NISQ Era

- Noisy Intermediate-Scale Quantum (NISQ)
  - **Noisy**: qubits are sensitive to environment; quantum gates are unreliable
  - **Limited number** of qubits: tens to hundreds of qubits
  - **Limited connectivity**



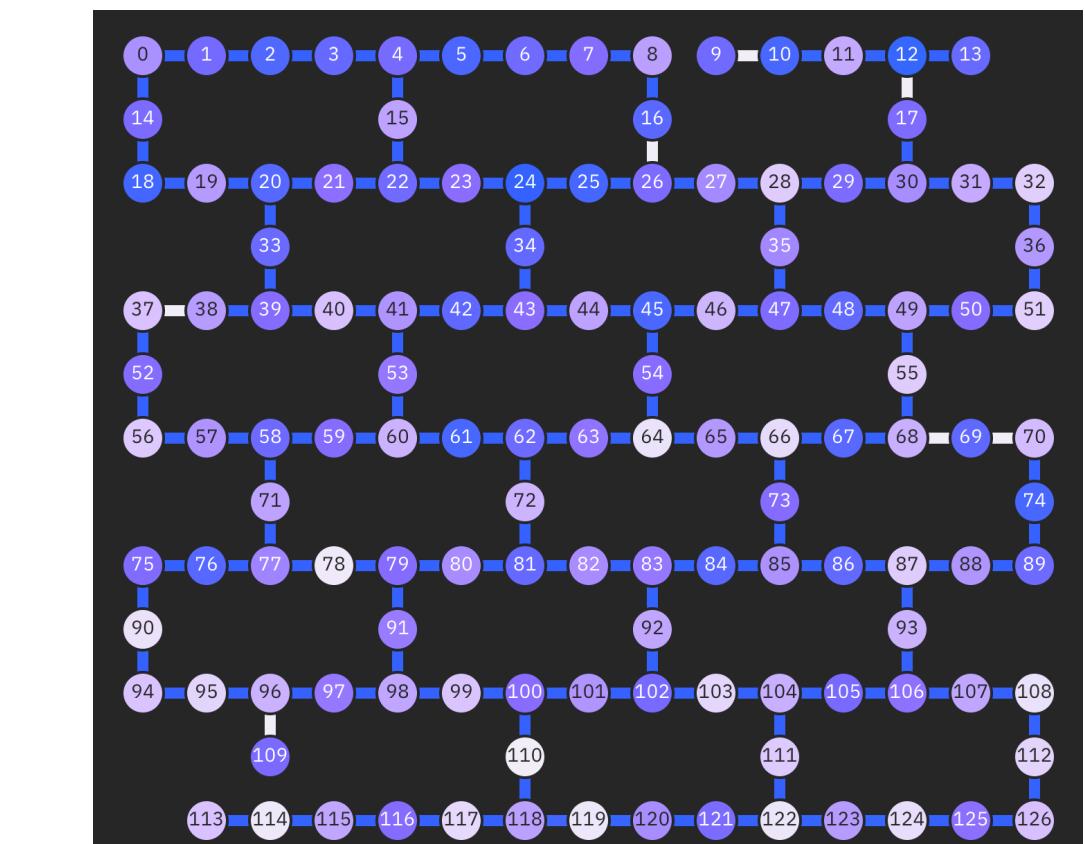
IBMQ Gate Error Rate

<https://quantum-computing.ibm.com/>



Google Sycamore 53Q

<https://www.nature.com/articles/s41586-019-1666-5>

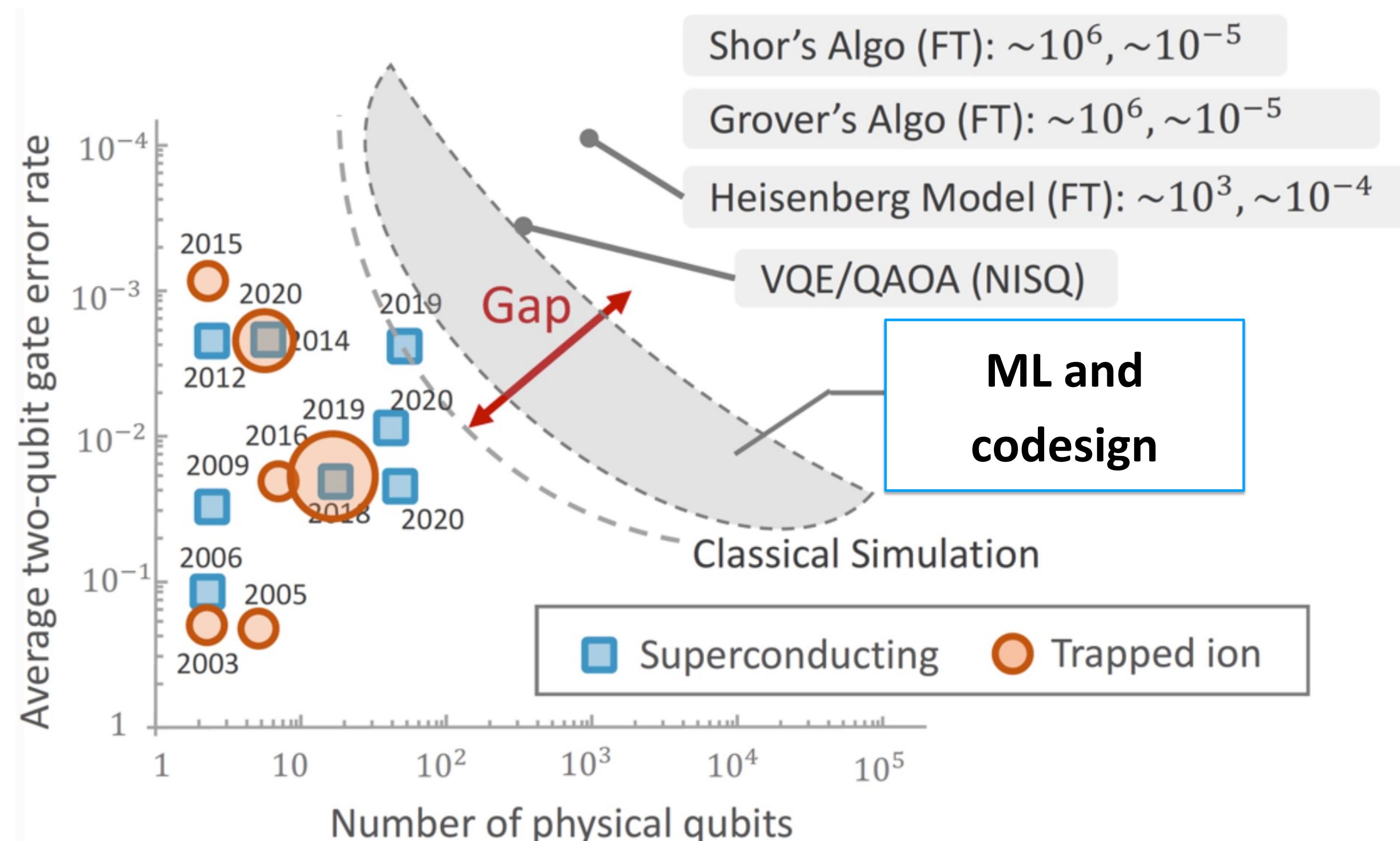


IBM Washington 127Q

<https://quantum-computing.ibm.com/>

# Close the Gap with ML and Codesign

- Close the gap with **machine learning and algorithm-hardware codesign**



\*Size of data point indicates connectivity; larger means denser connectivity.

# Good Infrastructure is Critical

- To enable ML and algorithm-hardware codesign for quantum
- Need a fast and easy-to-use classical simulation framework to
  - Analysis circuit behavior
  - Study noise impact
  - Develop codesign strategy
  - Develop ML model for quantum optimization

# TorchQuantum Library

- A fast library for classical simulation of quantum circuit in **PyTorch**
  - Automatic **gradient** computation for training parameterized quantum circuit
  - **Batch mode** tensorized processing, accelerated on CPUs and GPUs
  - **Dynamic computation graph** for easy debugging
  - Easy construction of **hybrid classical and quantum** neural networks
  - **Gate** level and **pulse** level simulation support
  - **Converters** to other frameworks such as IBM Qiskit
  - Classical ML models to predict

# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations 

1.3 TQ for State Prep 

1.4 TQ for VQE 

1.5 TQ for QNN 

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning 

2.2 QuantumNAT: Noise Injection and Quantization 

2.3 QOC: On-Chip Training 

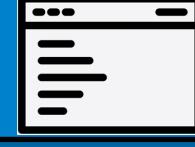
2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression 

## Section 3

### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control 

3.2 Variational Pulse Learning 

# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations



1.3 TQ for State Prep



1.4 TQ for VQE



1.5 TQ for QNN



## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning



2.2 QuantumNAT: Noise Injection and Quantization



2.3 QOC: On-Chip Training



2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression

3.1 Quantum Optimal Control



3.2 Variational Pulse Learning



## Section 3

### Use TorchQuantum on Pulse level

# 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:

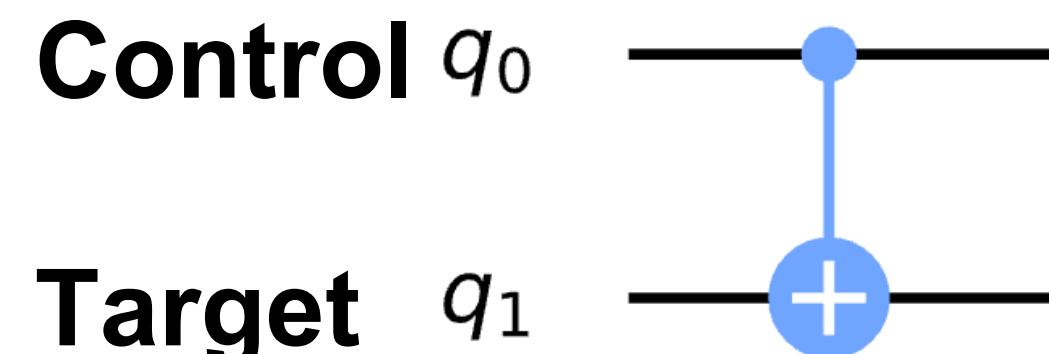
$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

- 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:



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

- 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

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

# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations 

1.3 TQ for State Prep 

1.4 TQ for VQE 

1.5 TQ for QNN 

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning 

2.2 QuantumNAT: Noise Injection and Quantization 

2.3 QOC: On-Chip Training 

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression 

## Section 3

### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control 

3.2 Variational Pulse Learning 

# TQ for Statevector simulation

- Performing matrix-vector multiplication between the gate matrix and statevector
  - The tq.QuantumDevice stores the statevectors
    - `q_dev = tq.QuantumDevice(n_wires=5)`
  - Two ways of applying quantum gates: method 1:
    - `import torchquantum.functional as tqf`
    - `tqf.h(q_dev, wires=1)`

# TQ for Statevector simulation

- Performing matrix-vector multiplication between the gate matrix and statevector
  - The tq.QuantumDevice stores the statevectors
    - `q_dev = tq.QuantumDevice(n_wires=5)`
  - Two ways of applying quantum gates: method 2:
    - `h_gate = tq.H()`
    - `h_gate(q_dev)`

# TQ for Statevector simulation

- Performing matrix-vector multiplication between the gate matrix and statevector
  - The tq.QuantumState class can also store the statevectors
    - `q_state = tq.QuantumState(n_wires=5)`
  - Three ways of applying quantum gates
    - `import torchquantum.functional as tqf`
    - `tqf.h(q_state, wires=1)`
    - `h_gate = tq.H()`
    - `h_gate(q_state)`
    - `q_state.h()`
    - `q_state.rx(wires=1, params=0.2 * np.pi)`

# TQ for Statevector simulation

- Performing matrix-vector multiplication between the gate matrix and statevector
- The implementation of statevector and quantum gates are using the native data structure in PyTorch

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

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

# Dynamic computation graph

- Performing matrix-vector multiplication between the gate matrix and statevector
  - The tq.QuantumState class can also store the statevectors
    - `q_state = tq.QuantumState(n_wires=5)`
    - `q_state.h(wires=1)`
    - `print(q_state)`
    - `q_state.sx(wires=3)`
    - `print(q_state)`

# Batch Mode Tensorized Processing

- Performing matrix-vector multiplication between the gate matrix and statevector
  - The tq.QuantumState class can also store the statevectors
    - `q_state = tq.QuantumState(n_wires=5, bsz=64)`

# GPU Support

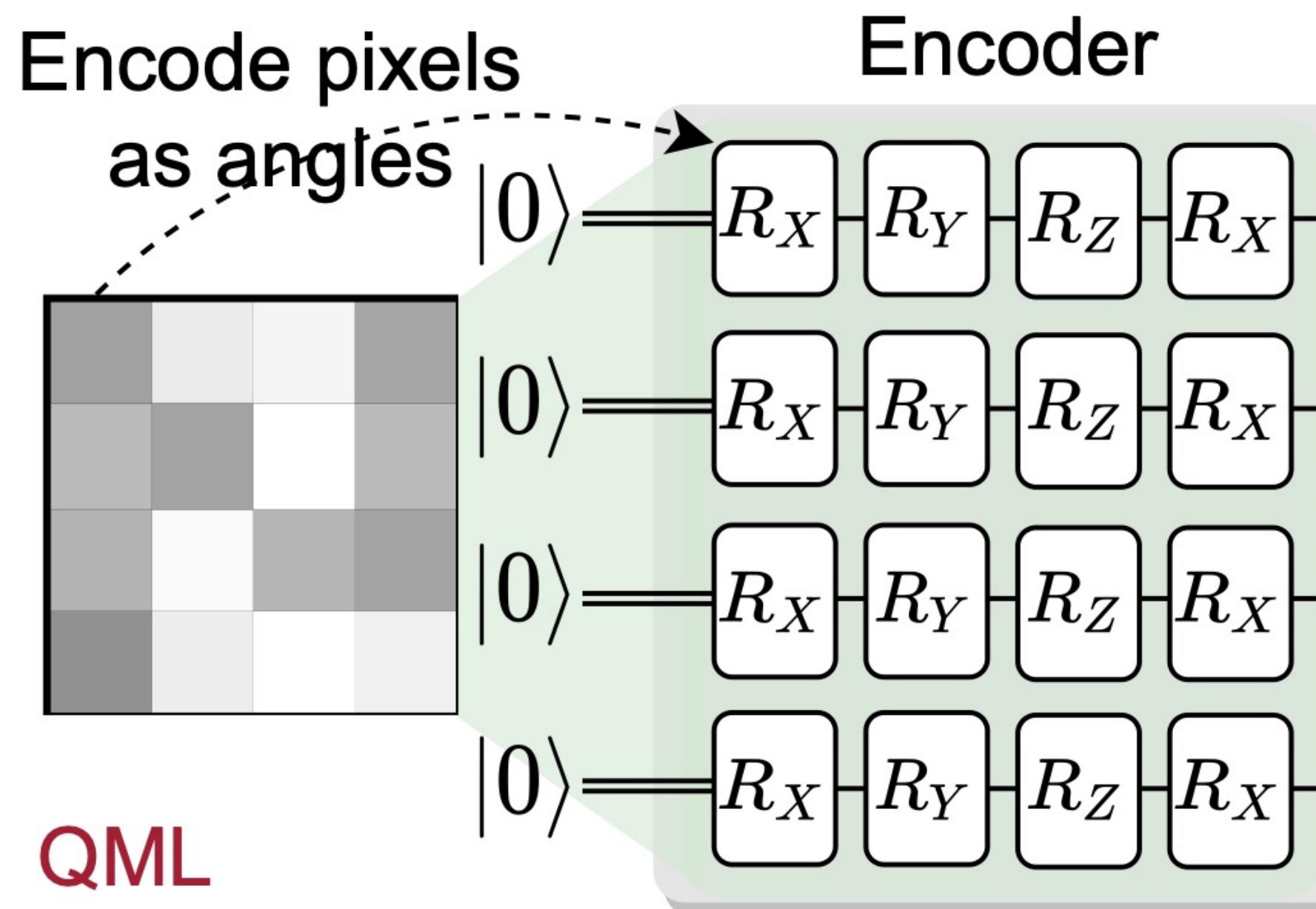
- Performing matrix-vector multiplication between the gate matrix and statevector
  - The tq.QuantumState class can also store the statevectors
    - `q_state = tq.QuantumState(n_wires=5, bsz=64)`
    - `q_state.to(torch.device('cuda'))`
  - Leverage the fast GPU support

# Automatic Gradient Computation

- Performing matrix-vector multiplication between the gate matrix and statevector
  - The tq.QuantumState class can also store the statevectors
    - `q_state = tq.QuantumState(n_wires=2)`
    - `target_quantum_state = torch.tensor([0, 0, 0, 1])`
    - `loss = 1 - (q_state.get_states_1d()[0] @ target_quantum_state).abs()`
    - `loss.backward()`

# Encoding Classical Data to Quantum various encoder support

- `tq.AmplitudeEncoder()` encodes the classical values to the amplitude of quantum statevector
- `tq.PhaseEncoder()` encodes the classical values using the rotation gates



# Construct a Circuit Class

- Construct a class for circuit model
- `tq.QuantumModule` class
- In the `__init__` function, create all the gates that will be used
- In the `forward` function, specify how the gates will be used in the circuit
- Class `q_model(tq.QuantumModule)`

```
def __init__():
    self.n_wires = 2
    self.rx_0 = tq.RX(has_params=True, trainable=True)
    self.ry_0 = tq.RY(has_params=True, trainable=True)

def forward(q_dev):
    self.rx_0(q_dev)
    self.ry_0(q_dev)
```

# Conversion tq model to other frameworks

- Convert the tq.QuantumModule to other frameworks such as Qiskit
- `from torchquantum.plugins.qiskit_plugin import tq2qiskit`
- `circ = tq2qiskit(q_dev, q_model)`
- `circ.draw('mpl')`

# Easy Deployment on Real Quantum Machines

- Convert the tq.QuantumModule to other frameworks such as Qiskit
- ```
from torchquantum.plugins.qiskit_plugin import tq2qiskit
```
- ```
from torchquantum.plugins.qiskit_processor import QiskitProcessor
```
- ```
processor = QiskitProcessor(use_real_qc=False, max_jobs=1)
```
- ```
circ = tq2qiskit(q_dev, model)
```
- ```
circ.measure_all()
```
- ```
res = processor.process_ready_circs(q_dev, [circ])
```

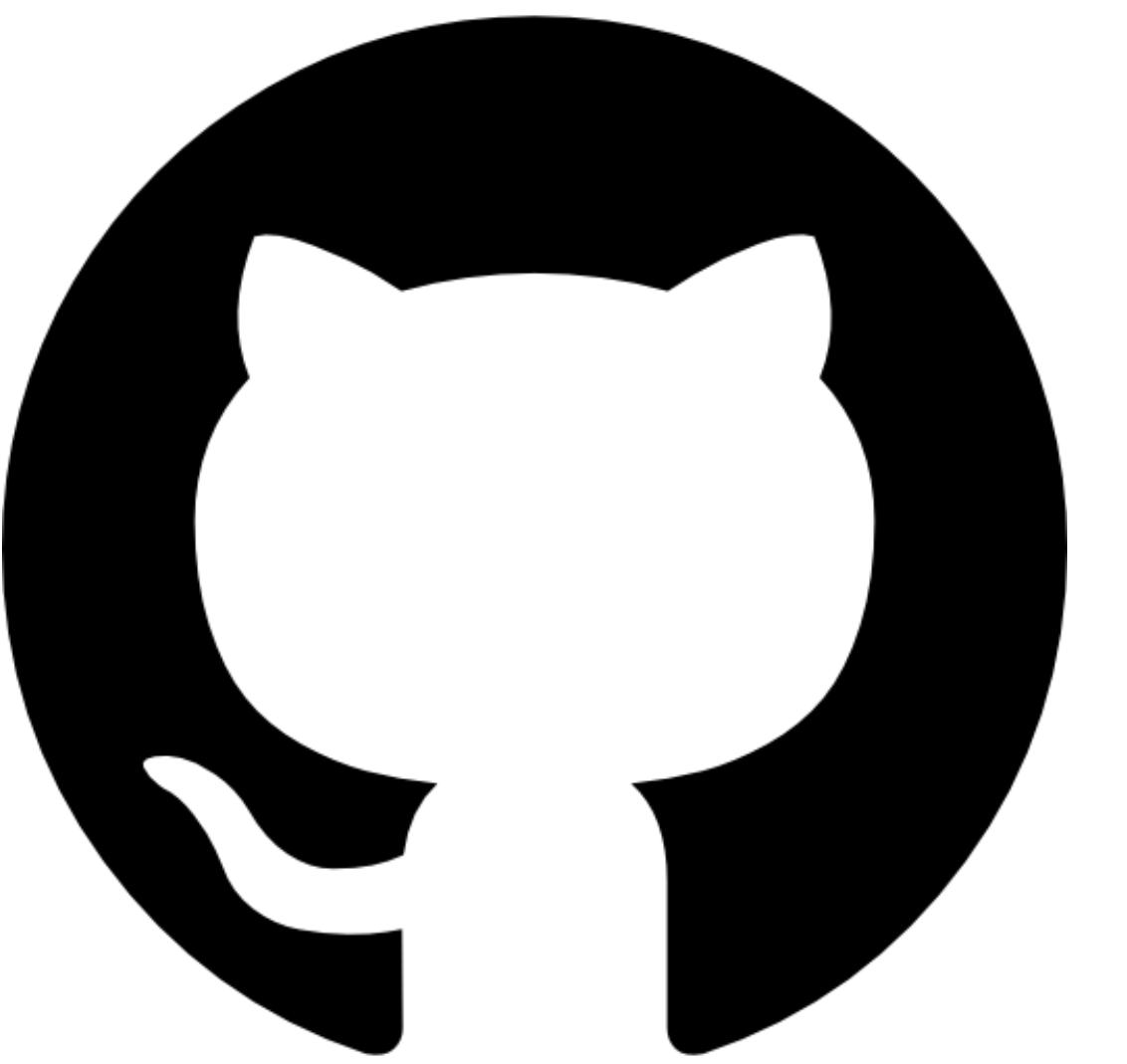
# Compare the Speed of TQ and PennyLane

# Compare the Speed of TQ and Qiskit

-

# Hands-On Section

## 1.2 TorchQuantum Operations



# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations

1.3 TQ for State Prep

1.4 TQ for VQE

1.5 TQ for QNN

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning

2.2 QuantumNAT: Noise Injection and Quantization

2.3 QOC: On-Chip Training

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression

## Section 3

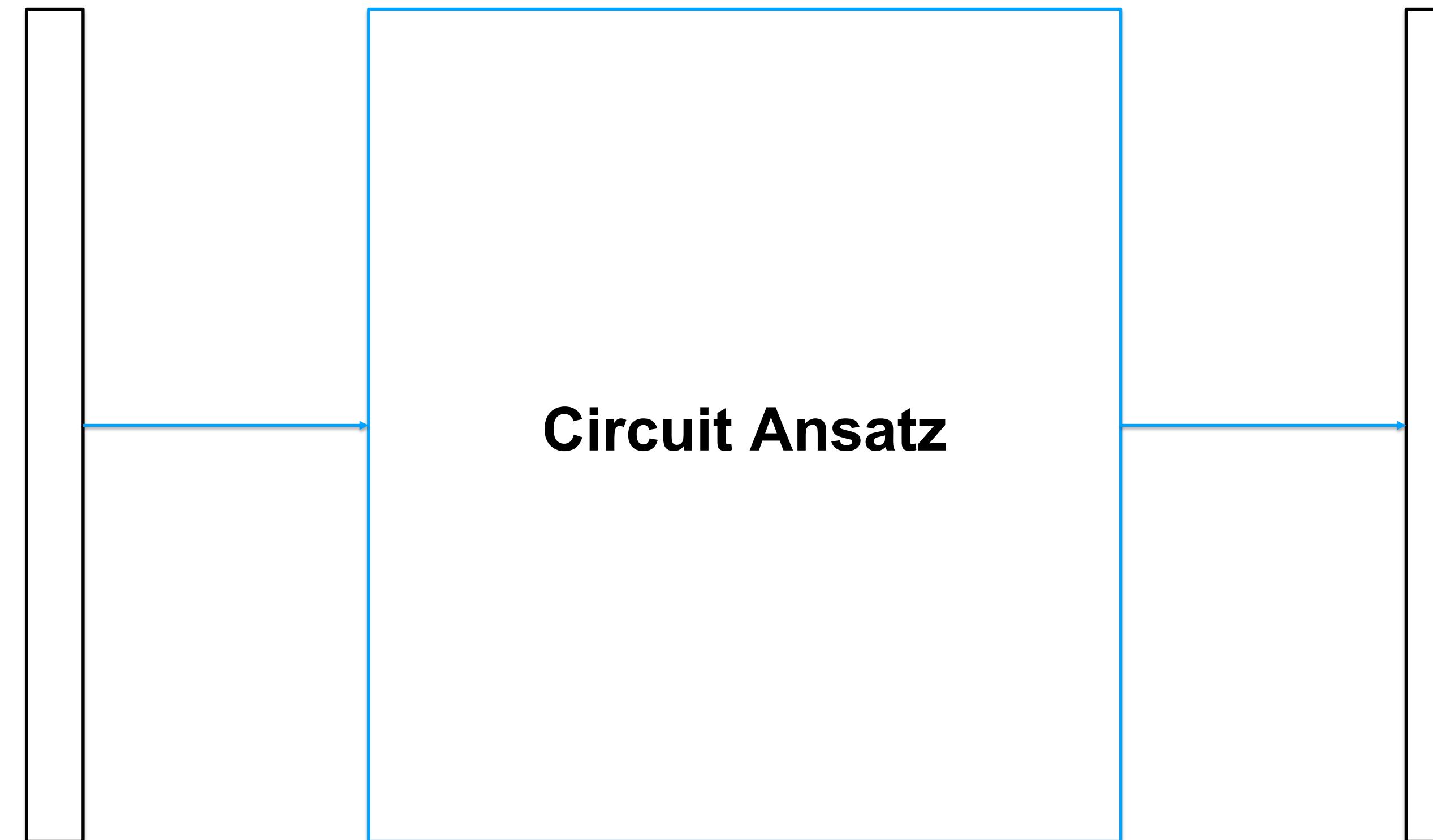
### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control

3.2 Variational Pulse Learning

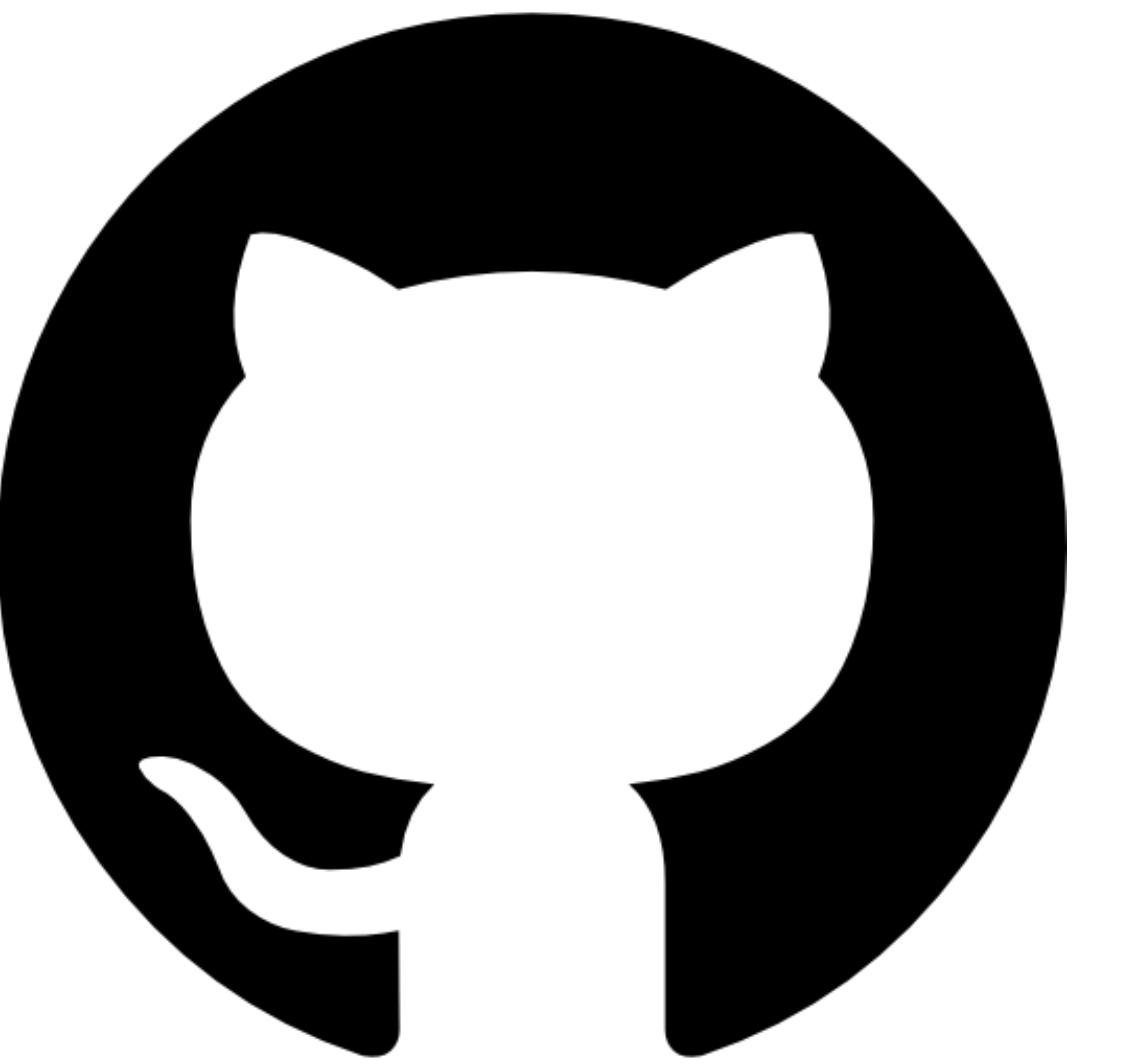
# State Preparation with Parameterized Circuit

- Use parameterized circuit to prepare initial quantum states



# Hands-On Section

## 1.2 TorchQuantum Operations



# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations

1.3 TQ for State Prep

1.4 TQ for VQE

1.5 TQ for QNN

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning

2.2 QuantumNAT: Noise Injection and Quantization

2.3 QOC: On-Chip Training

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression

## Section 3

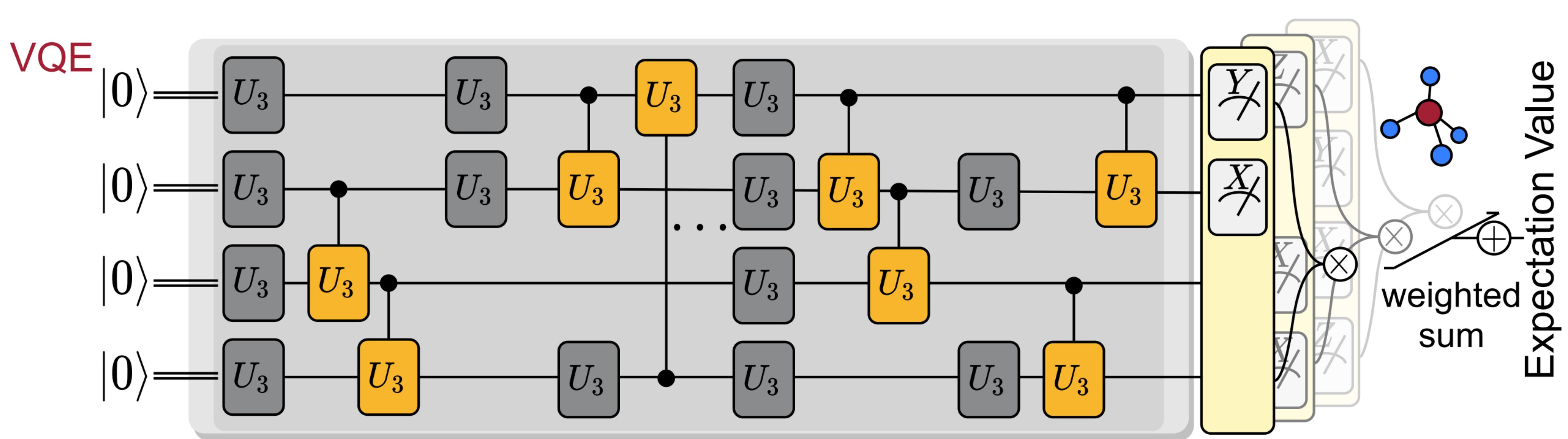
### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control

3.2 Variational Pulse Learning

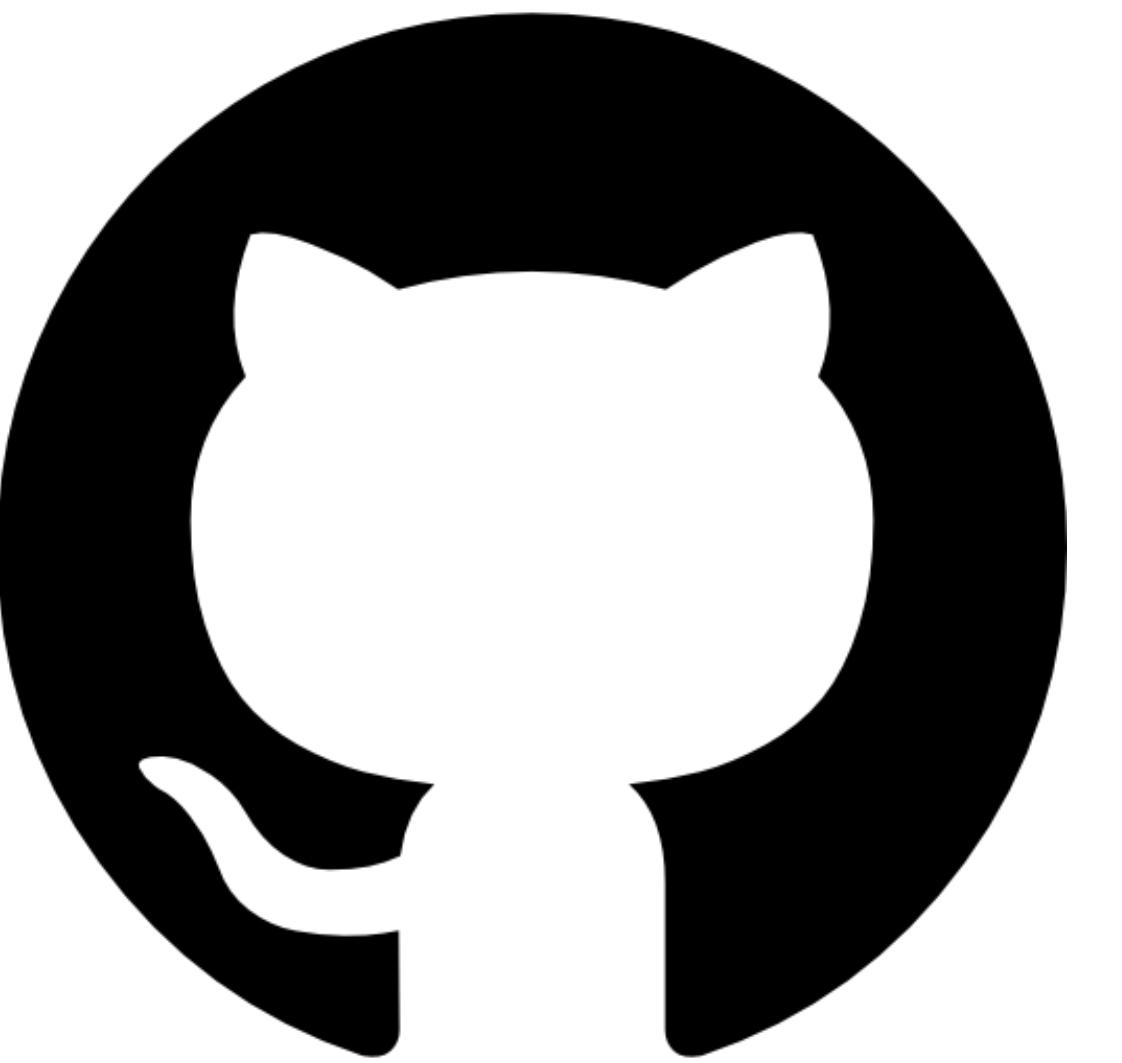
# Variational Quantum Eigensolver

- VQE: Finds the ground state energy of molecule Hamiltonian



# Hands-On Section

## 1.3 TorchQuantum Operations



# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations

1.3 TQ for State Prep

1.4 TQ for VQE

1.5 TQ for QNN

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning

2.2 QuantumNAT: Noise Injection and Quantization

2.3 QOC: On-Chip Training

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression

## Section 3

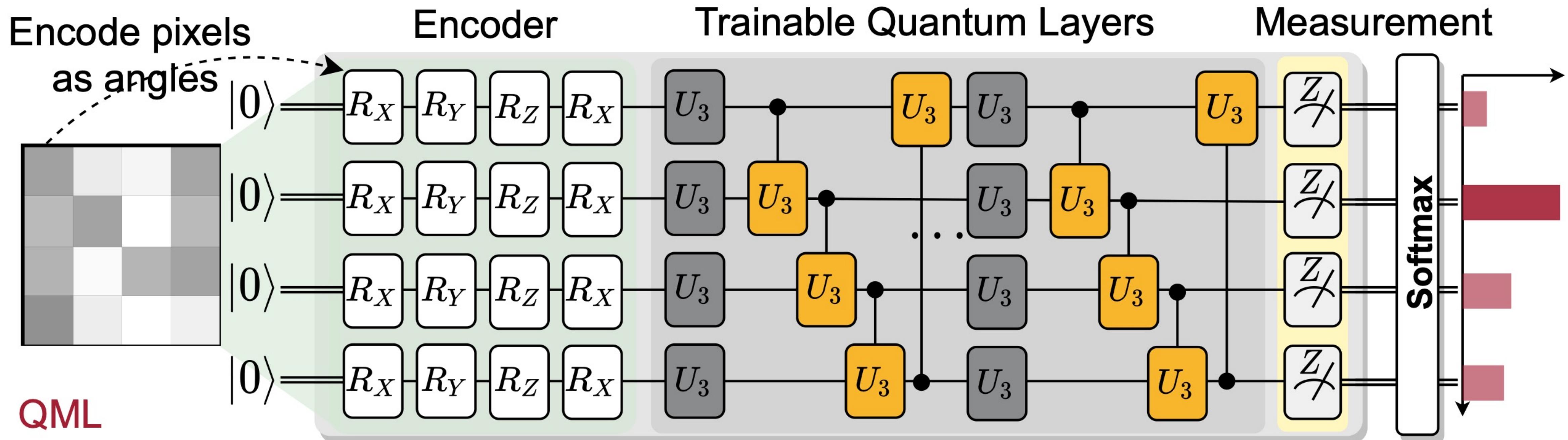
### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control

3.2 Variational Pulse Learning

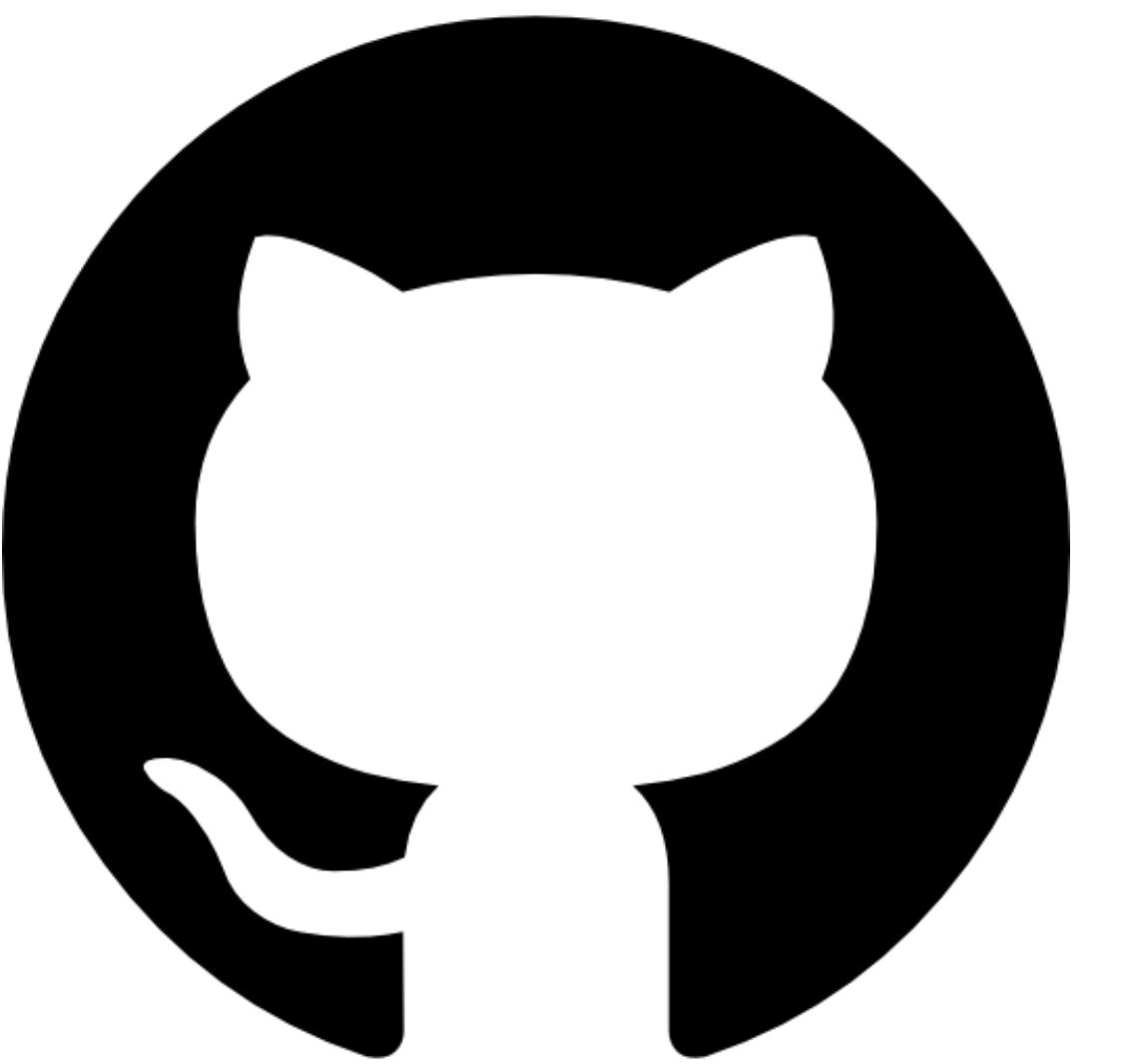
# Quantum Neural Networks for MNIST Classification

- Encode the classical values using phase encoding
- Then trainable quantum layers
- Finally measurement layers



# Hands-On Section

## *TorchQuantum Operations*



# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations

1.3 TQ for State Prep

1.4 TQ for VQE

1.4 TQ for QNN

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning

2.2 QuantumNAT: Noise Injection and Quantization

2.3 QOC: On-Chip Training

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression

## Section 3

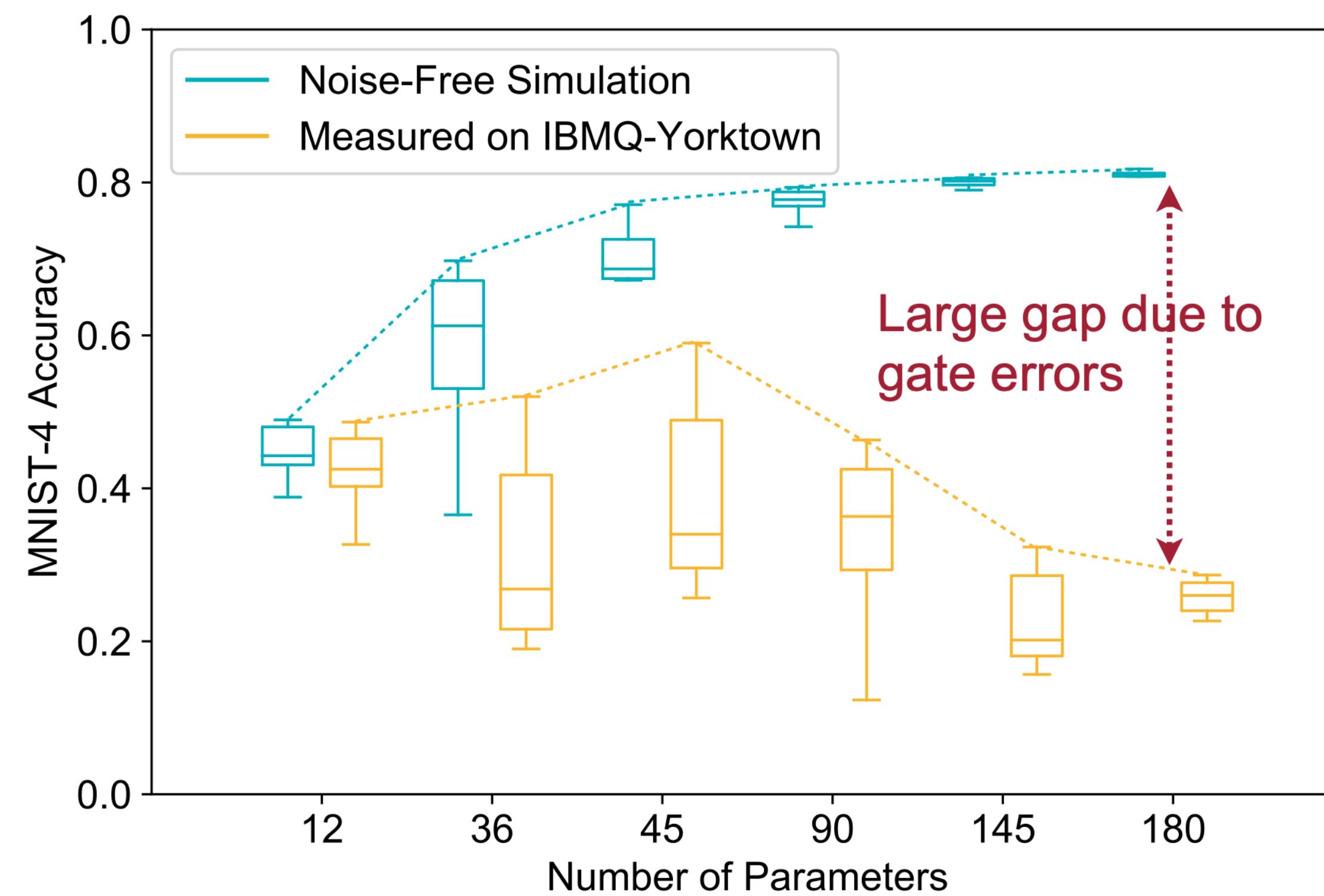
### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control

3.2 Variational Pulse Learning

# Challenges of PQC — Noise

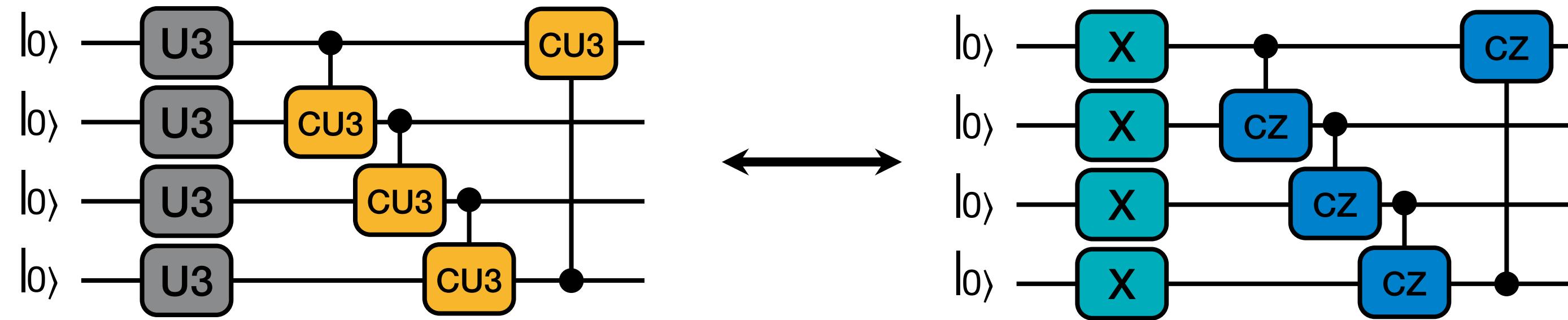
- Noise **degrades** PQC reliability
- More parameters increase the noise-free accuracy but degrade the measured accuracy
- Therefore, circuit architecture is critical



# Challenges of PQC — Large Design Space

- Large design space for circuit architecture

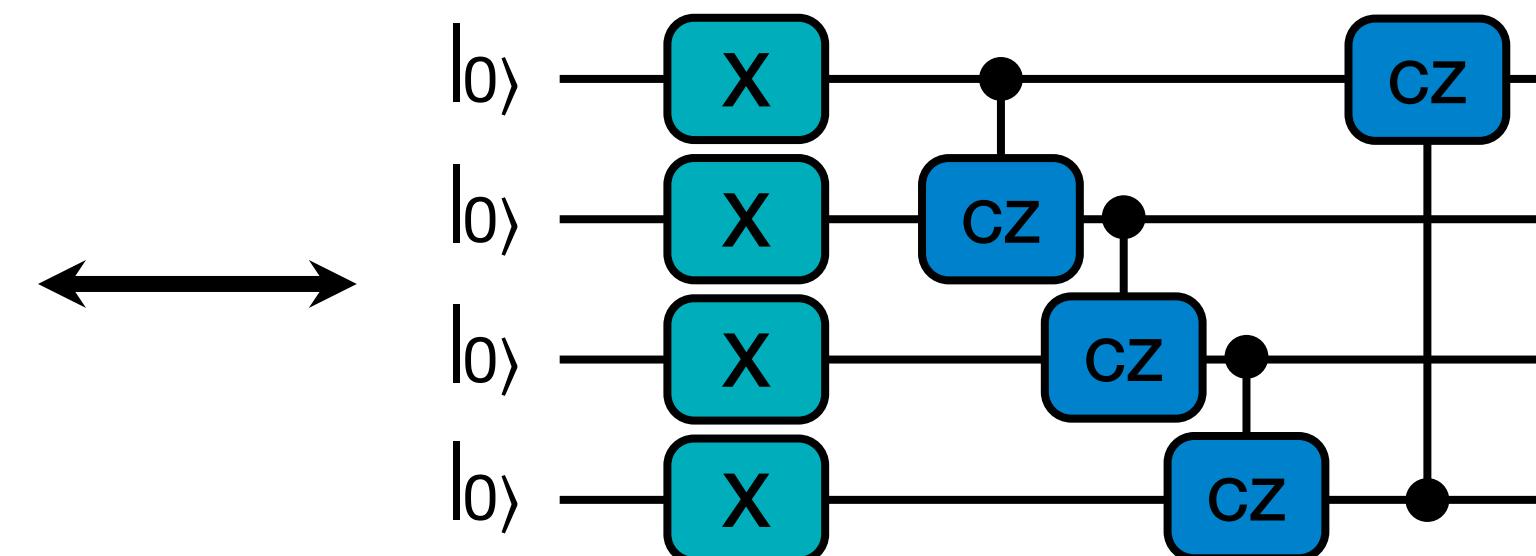
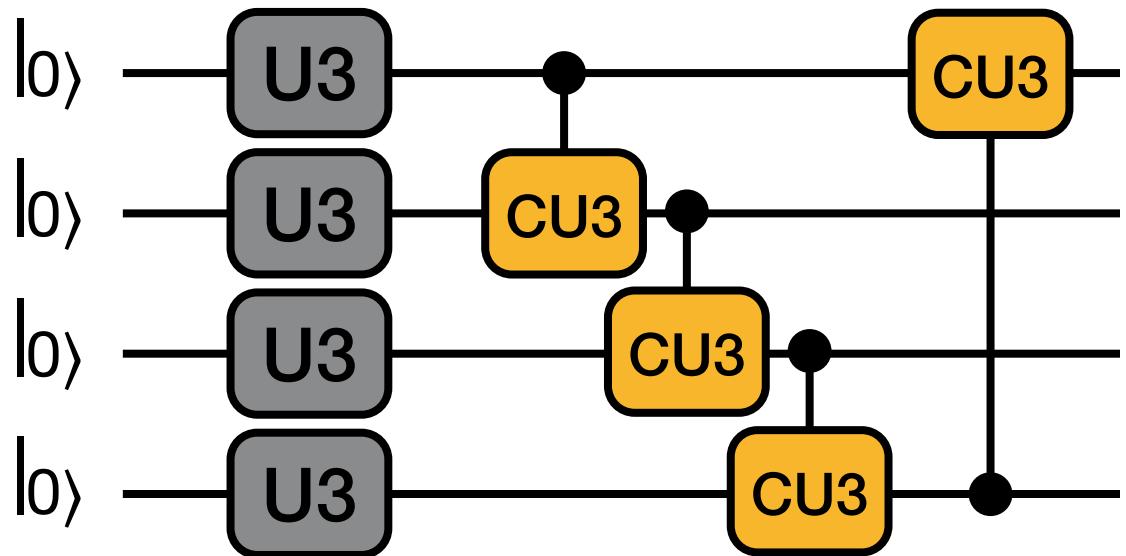
- Type of gates



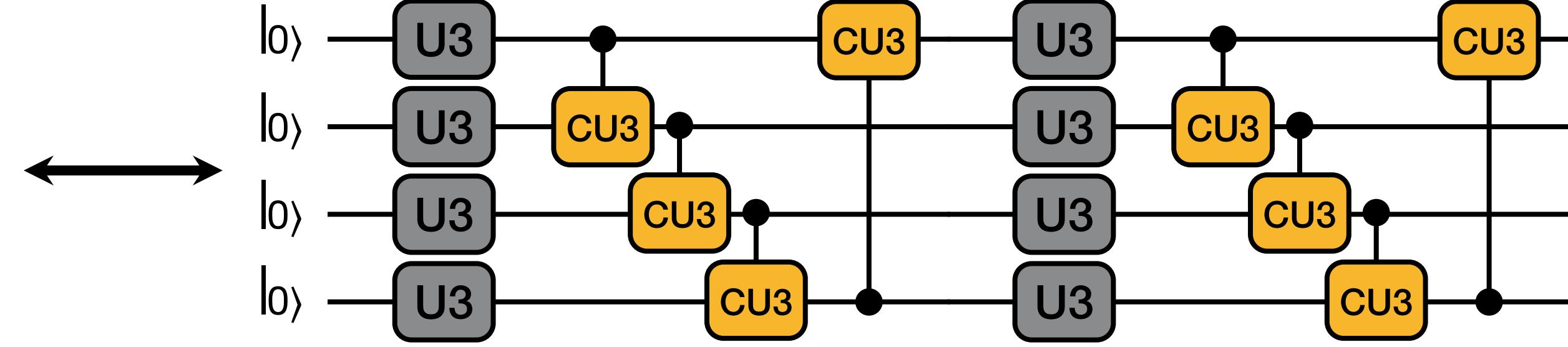
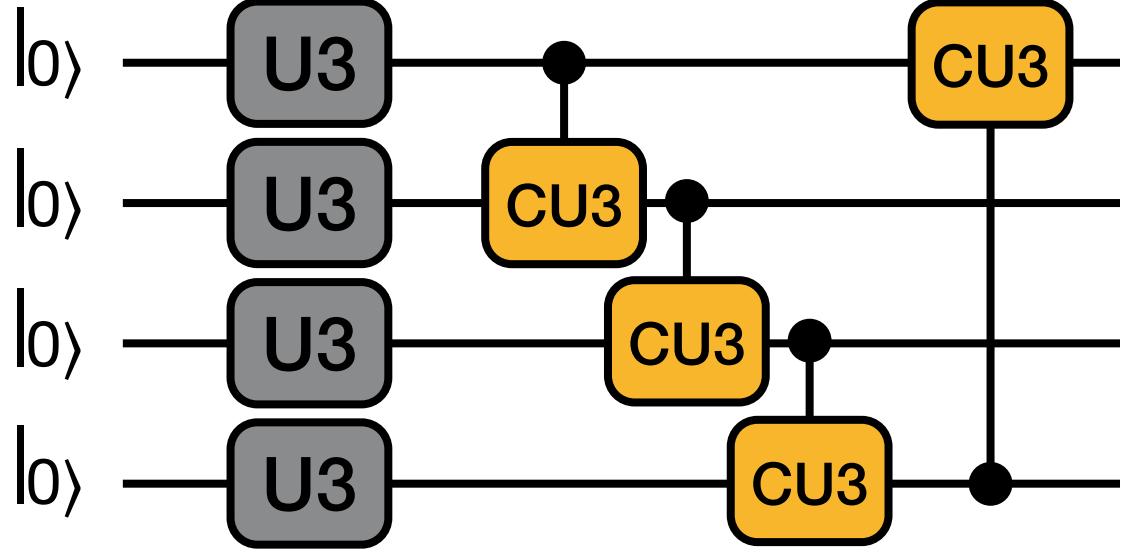
# Challenges of PQC — Large Design Space

- Large design space for circuit architecture

- Type of gates



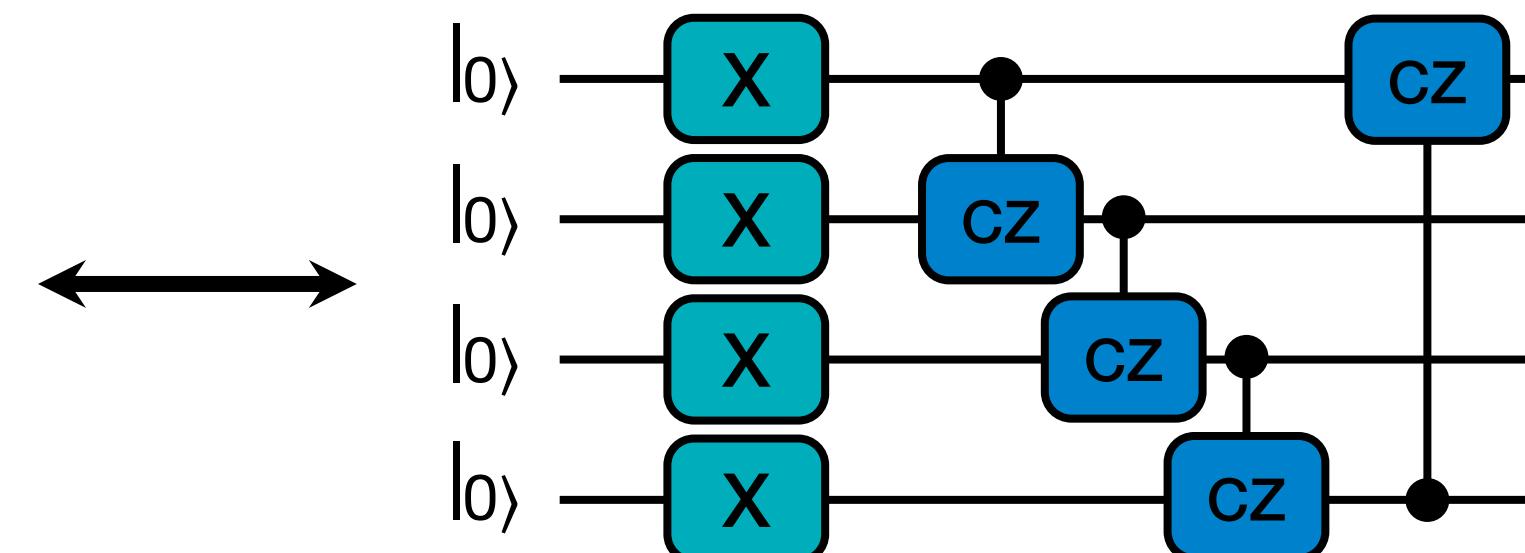
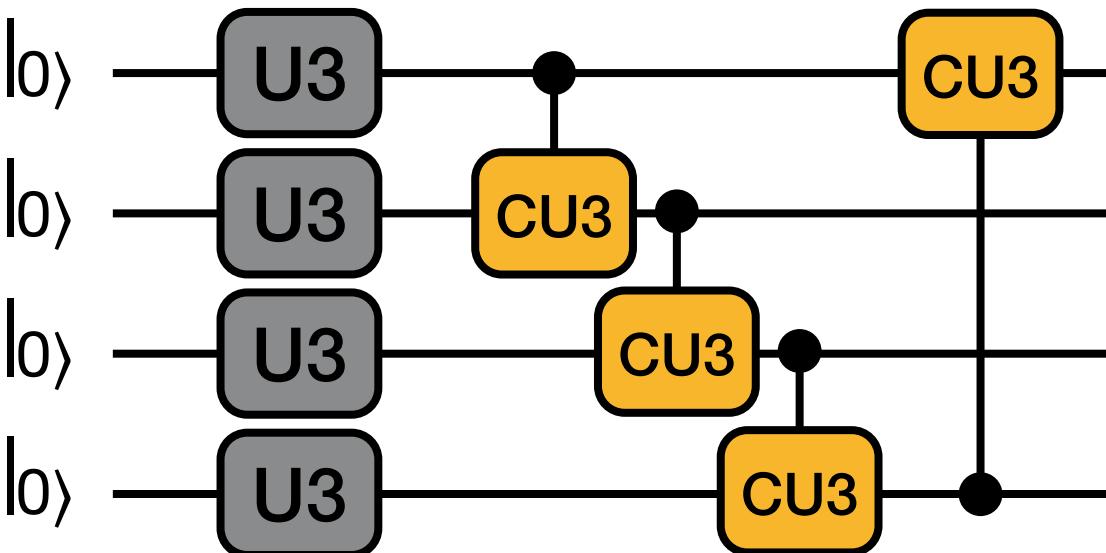
- Number of gates



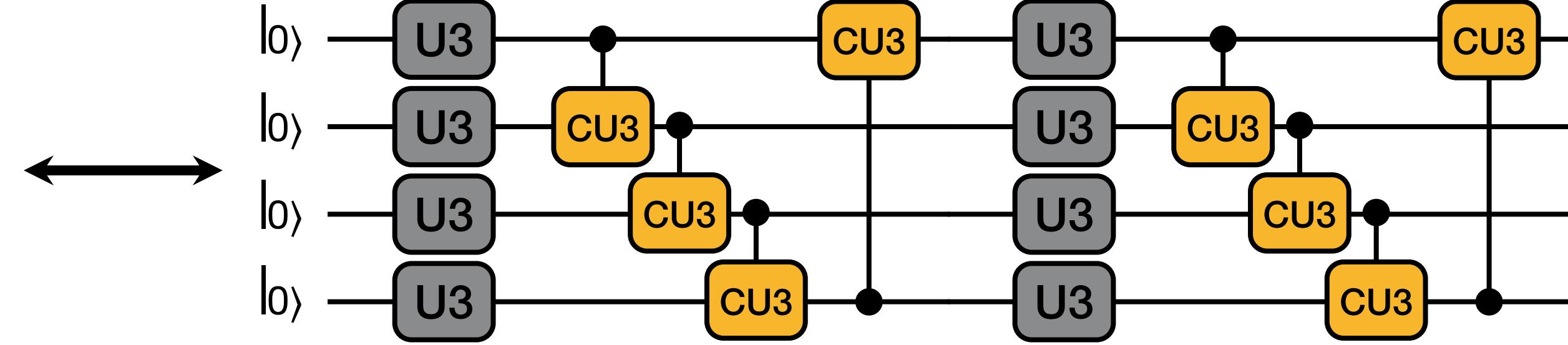
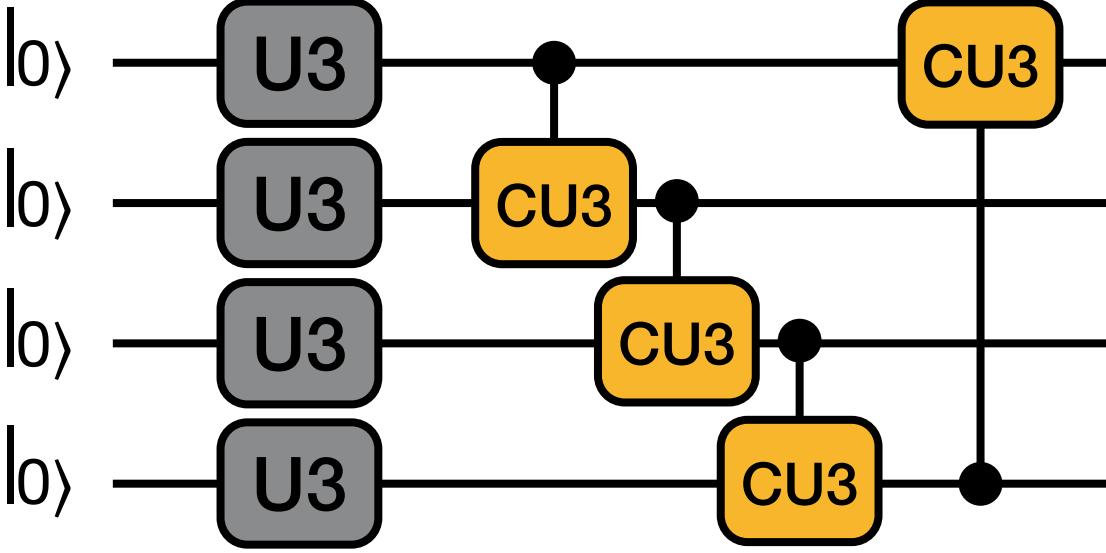
# Challenges of PQC — Large Design Space

- Large design space for circuit architecture

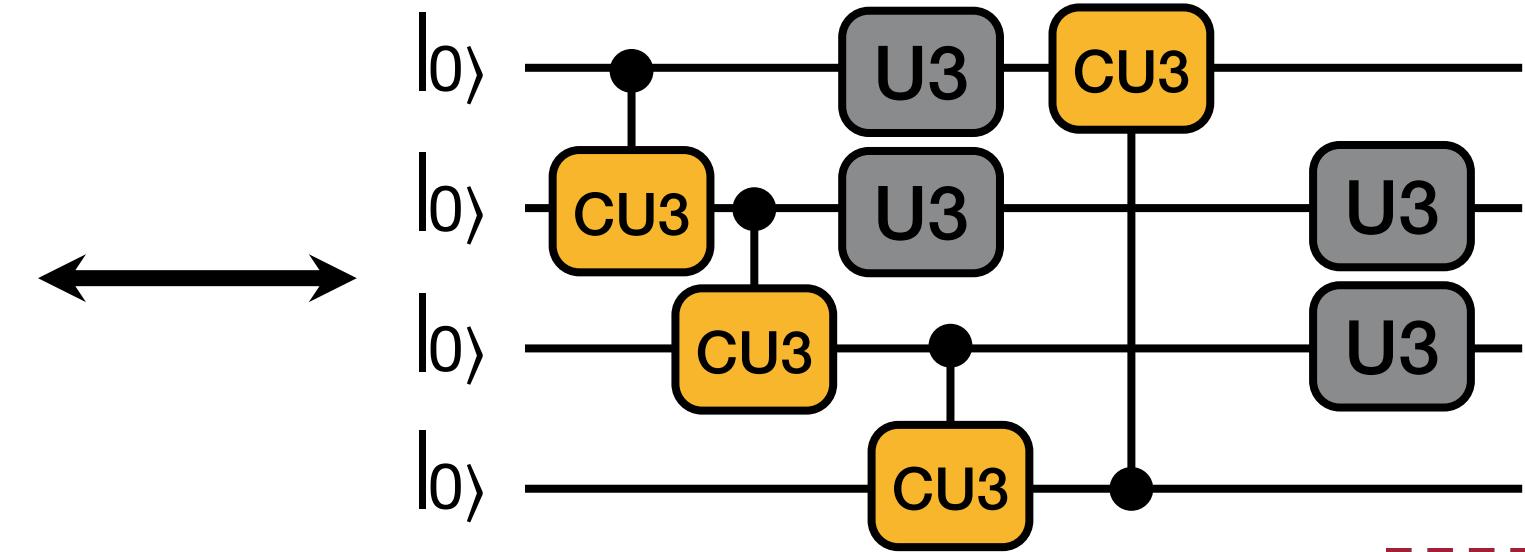
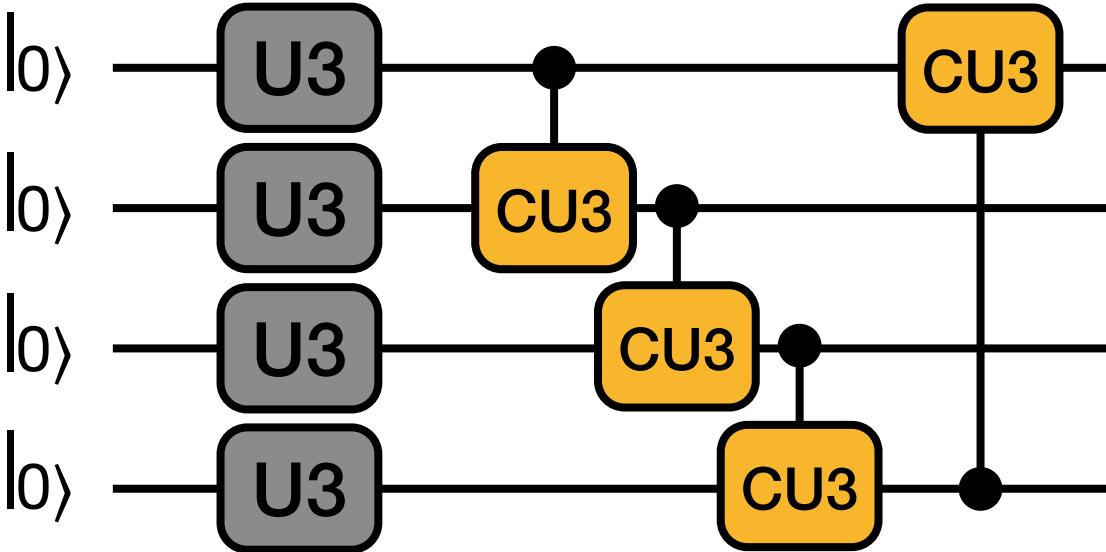
- Type of gates



- Number of gates



- Position of gates



# Goal of QuantumNAS

Automatically & efficiently search for noise-robust quantum circuit

Train one “SuperCircuit”,  
providing parameters to  
many “SubCircuits”

Solve the challenge of large  
design space

- 
- (1) Quantum noise feedback in the search loop
  - (2) Co-search the circuit architecture and qubit mapping

Solve the challenge of large  
quantum noise

# QuantumNAS

- SuperCircuit Construction and Training
- Noise-Adaptive Evolutionary Co-Search of SubCircuit and Qubit Mapping
- Train the Searched SubCircuit
- Iterative Quantum Gate Pruning

# QuantumNAS

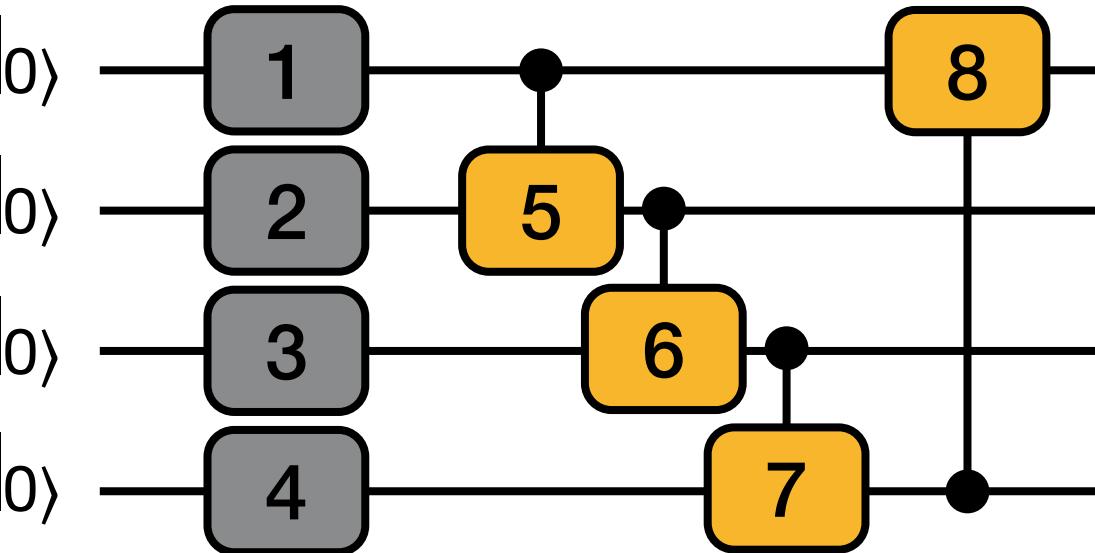
- SuperCircuit Construction and Training
- Noise-Adaptive Evolutionary Co-Search of SubCircuit and Qubit Mapping
- Train the Searched SubCircuit
- Iterative Quantum Gate Pruning

# SuperCircuit & SubCircuit

- Firstly construct a design space. For example, a design space of maximum 4 U3 in the first layer and 4 CU3 gates in the second layer

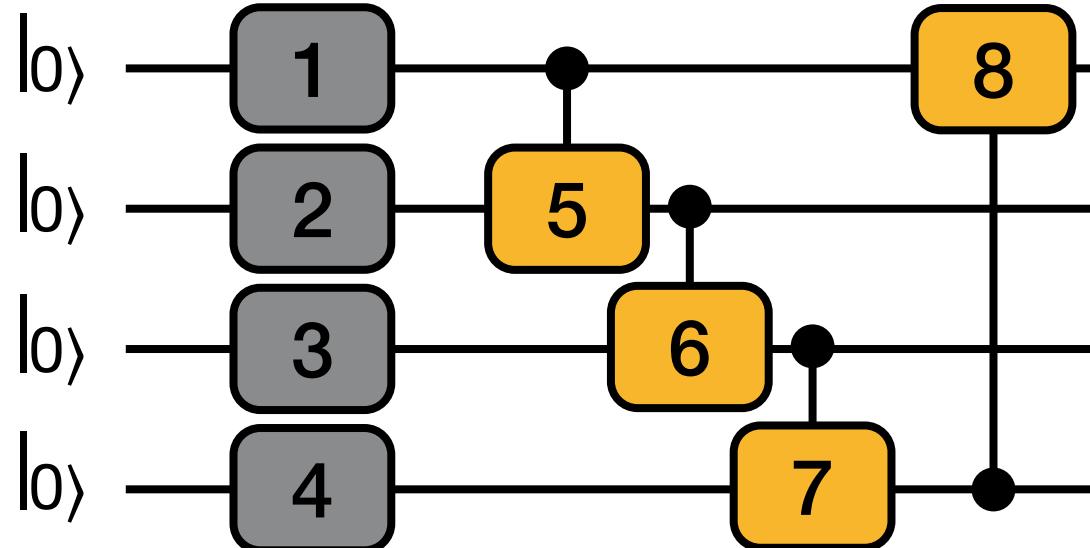
# SuperCircuit & SubCircuit

- Firstly construct a design space. For example, a design space of maximum 4 U3 in the first layer and 4 CU3 gates in the second layer
- SuperCircuit: the circuit with the **largest** number of gates in the design space
  - Example: SuperCircuit in U3+CU3 space

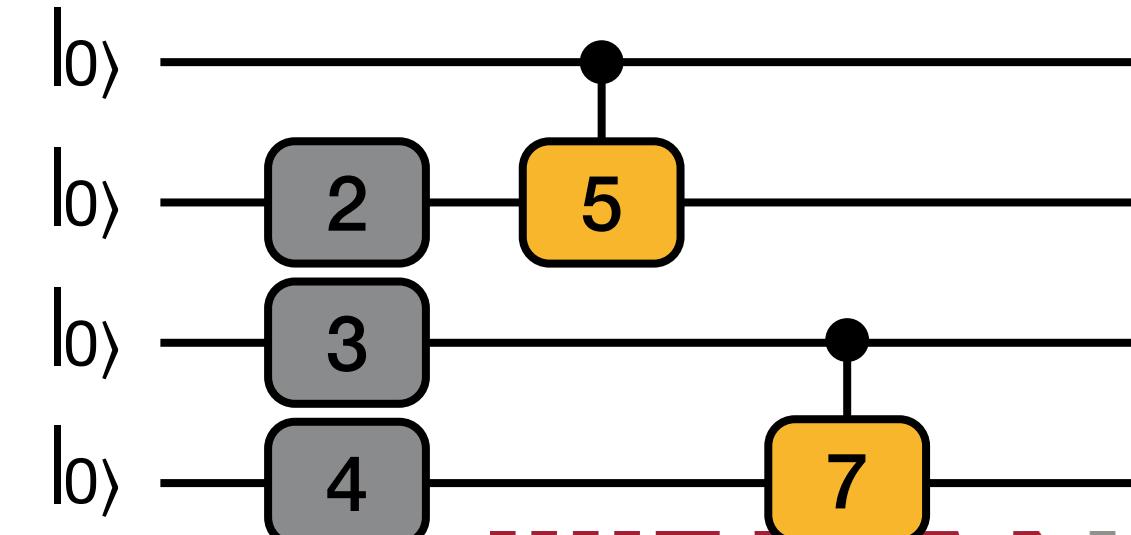
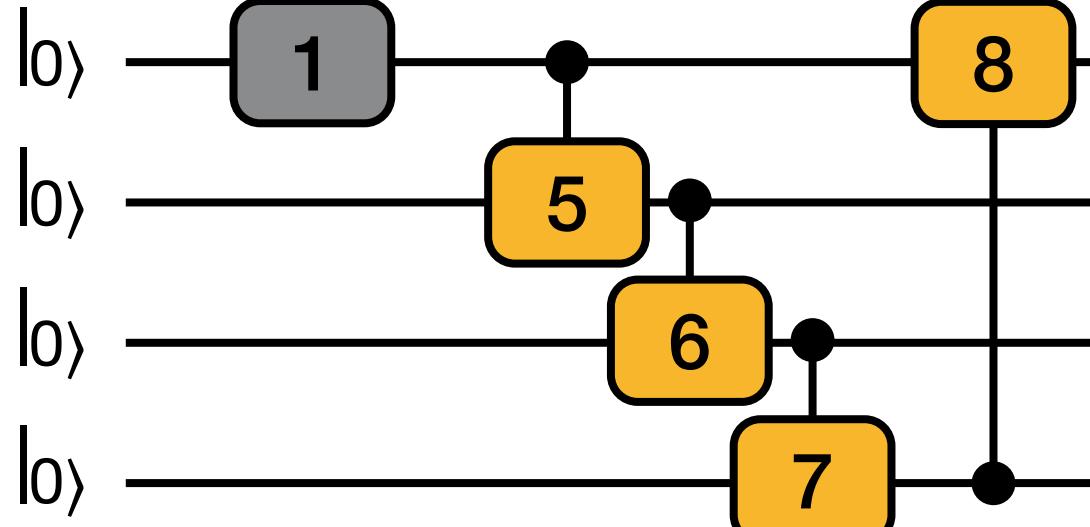
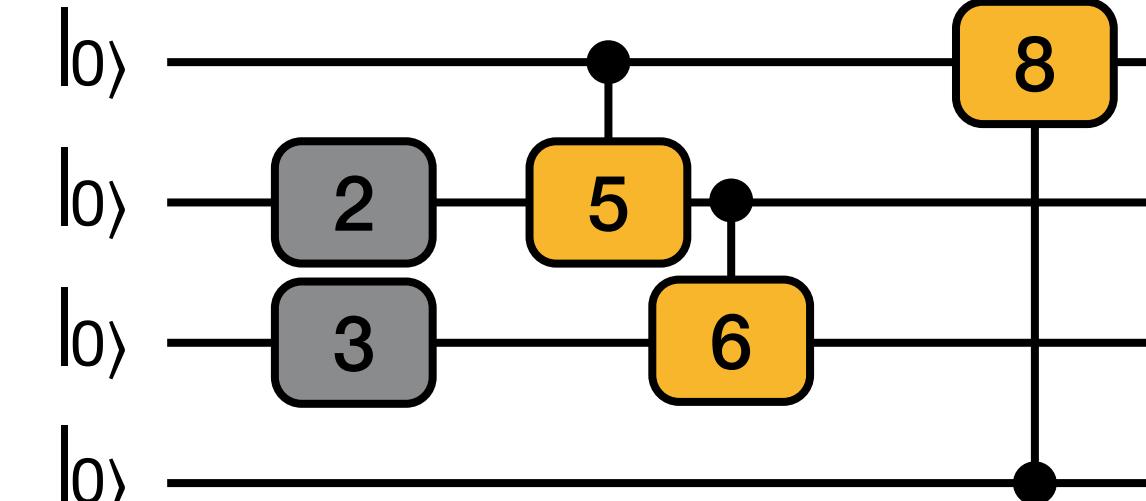


# SuperCircuit & SubCircuit

- Firstly construct a design space. For example, a design space of maximum 4 U3 in the first layer and 4 CU3 gates in the second layer
- SuperCircuit: the circuit with the **largest** number of gates in the design space
  - Example: SuperCircuit in U3+CU3 space



- Each candidate circuit in the design space (called SubCircuit) is a **subset** of the SuperCircuit



# SuperCircuit Construction

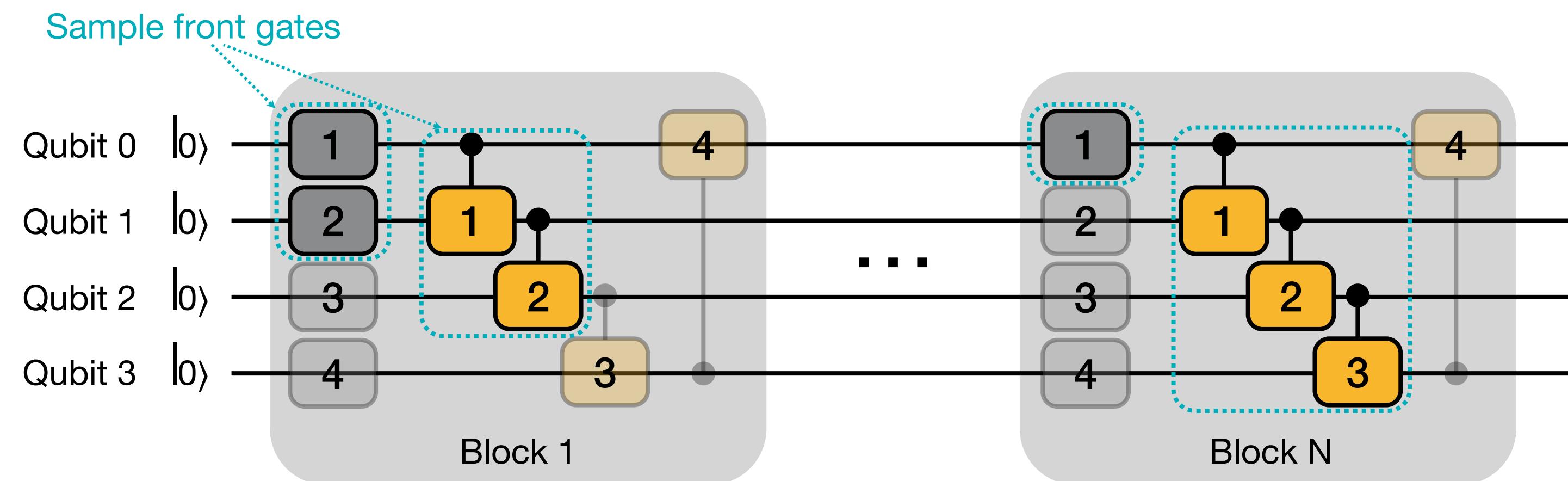
- Why use a SuperCircuit?
  - Enables **efficient** search of architecture candidates without training each
  - SubCircuit inherits parameters from SuperCircuit
  - With **inherited** parameters, we find some good SubCircuits, we find that they are **also good SubCircuits** with parameters **trained from-scratch** individually

# SuperCircuit Training

- In one SuperCircuit Training step:
  - Sample a gate subset of SuperCircuit (a SubCircuit)
    - Front Sampling and Restricted Sampling
  - Only use the subset to perform the task and updates the parameters in the subset
  - Parameter updates are cumulative across steps

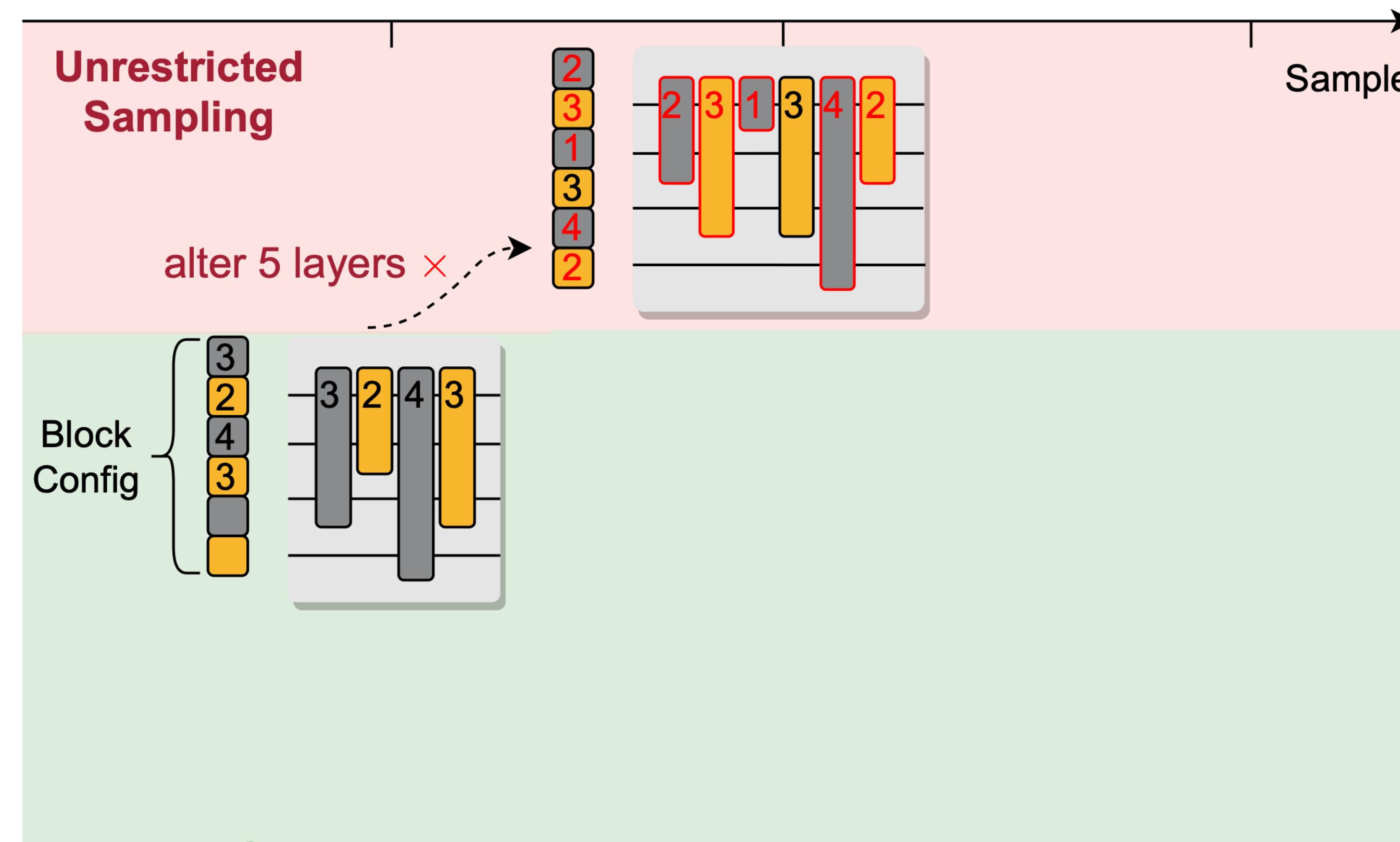
# Front Sampling

- During sampling, we first sample total number of blocks, then sample gates within each block
  - Front sampling: Only the **front** several blocks and **front** several gates can be sampled to make SuperCircuit training more stable



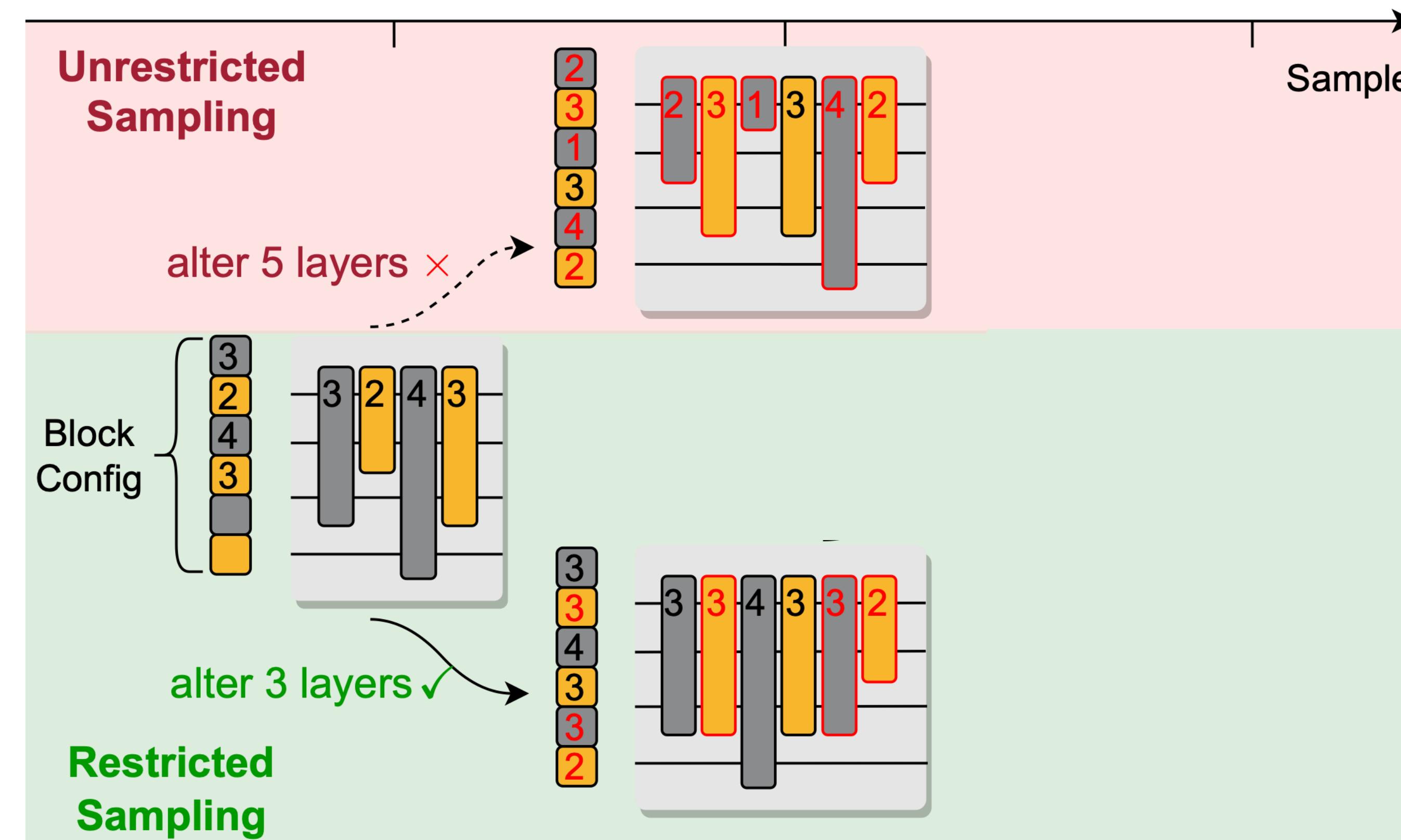
# Restricted Sampling

- Restricted Sampling:
  - Restrict the difference between SubCircuits of two consecutive steps
  - For example: restrict to at most 4 different layers



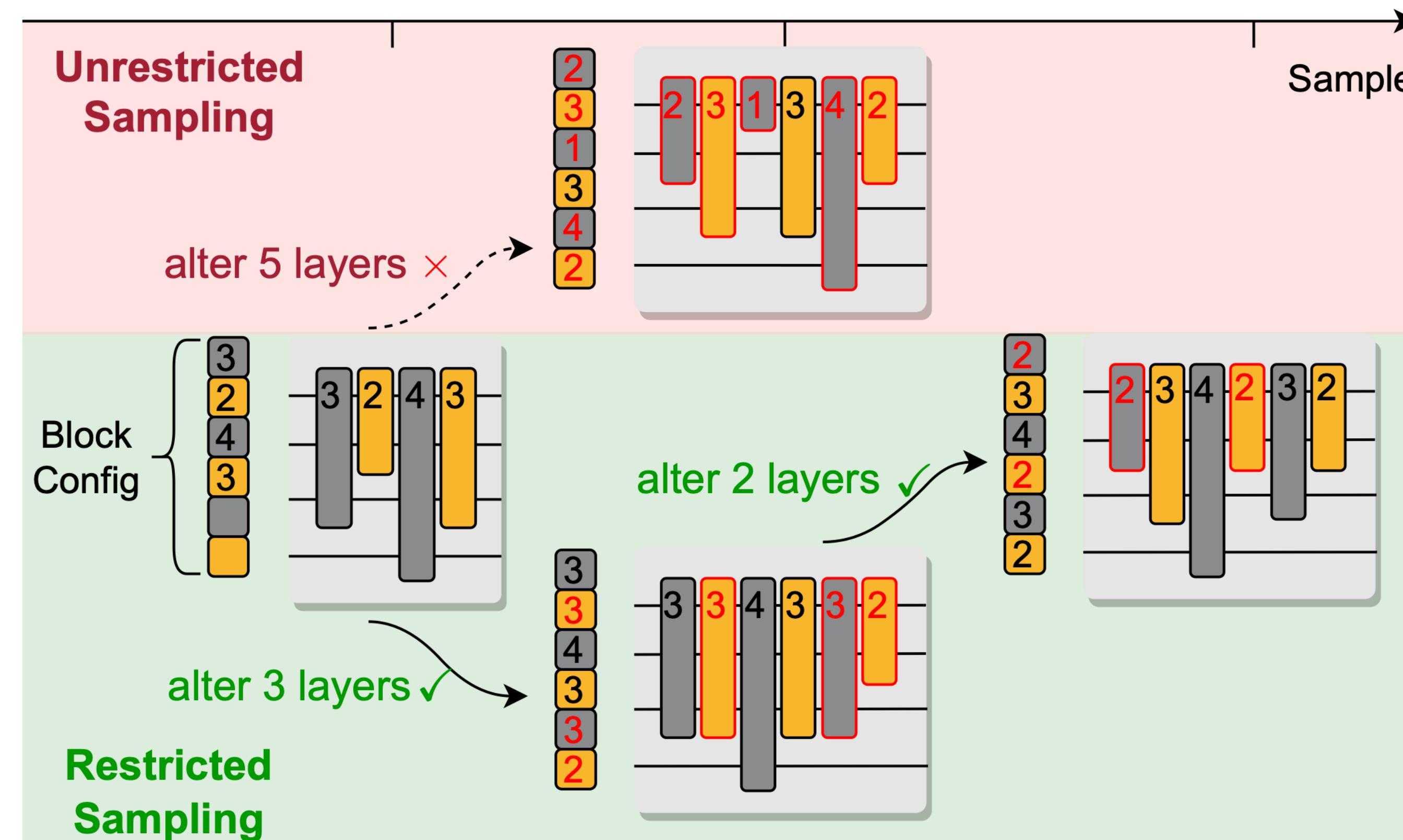
# Restricted Sampling

- Restricted Sampling:
  - Restrict the difference between SubCircuits of two consecutive steps
  - For example: restrict to at most 4 different layers



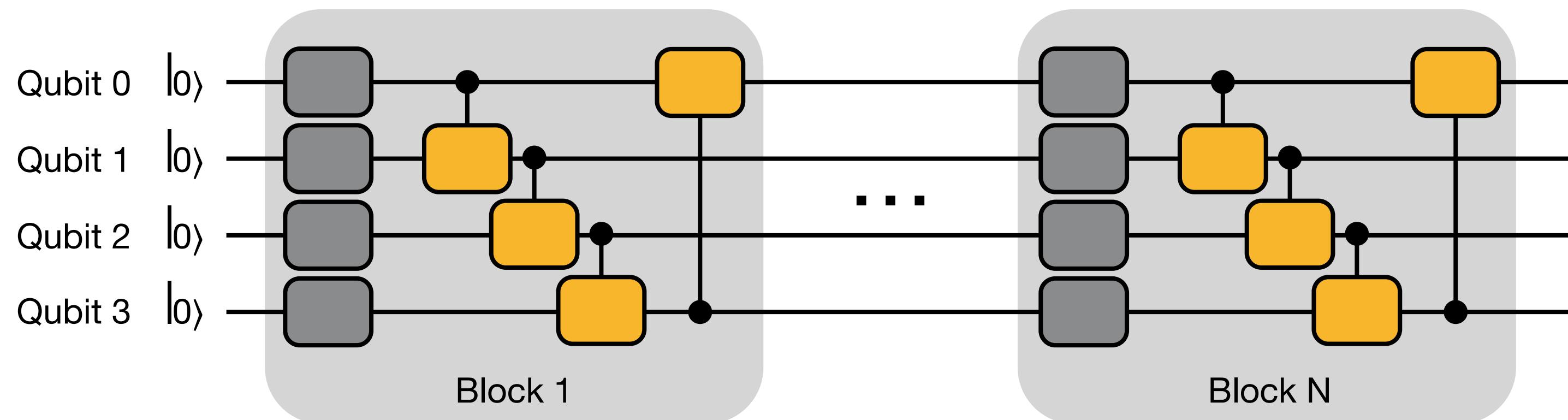
# Restricted Sampling

- Restricted Sampling:
  - Restrict the difference between SubCircuits of two consecutive steps
  - For example: restrict to at most 4 different layers



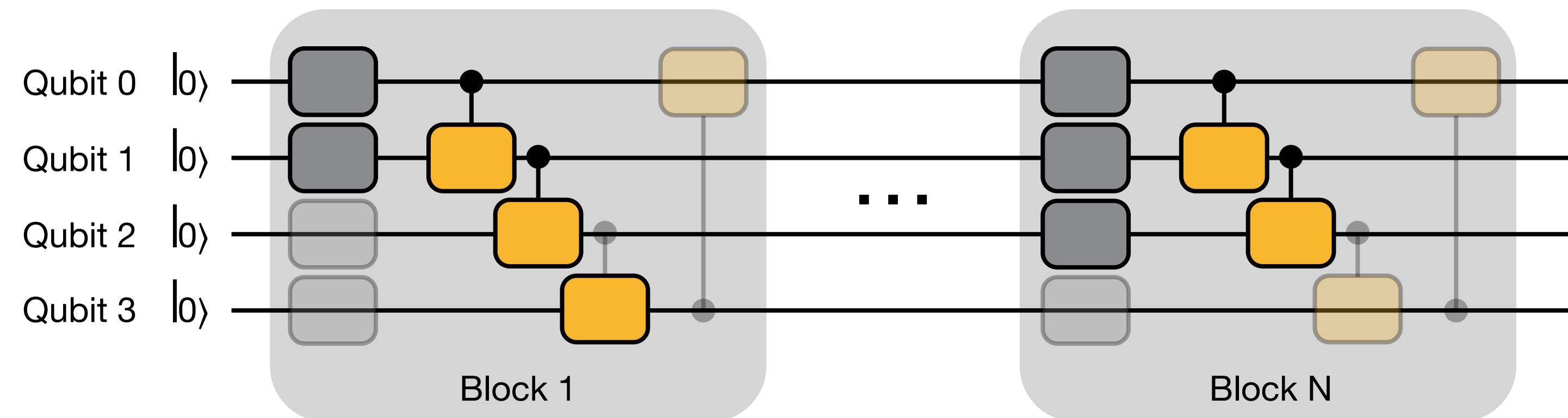
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



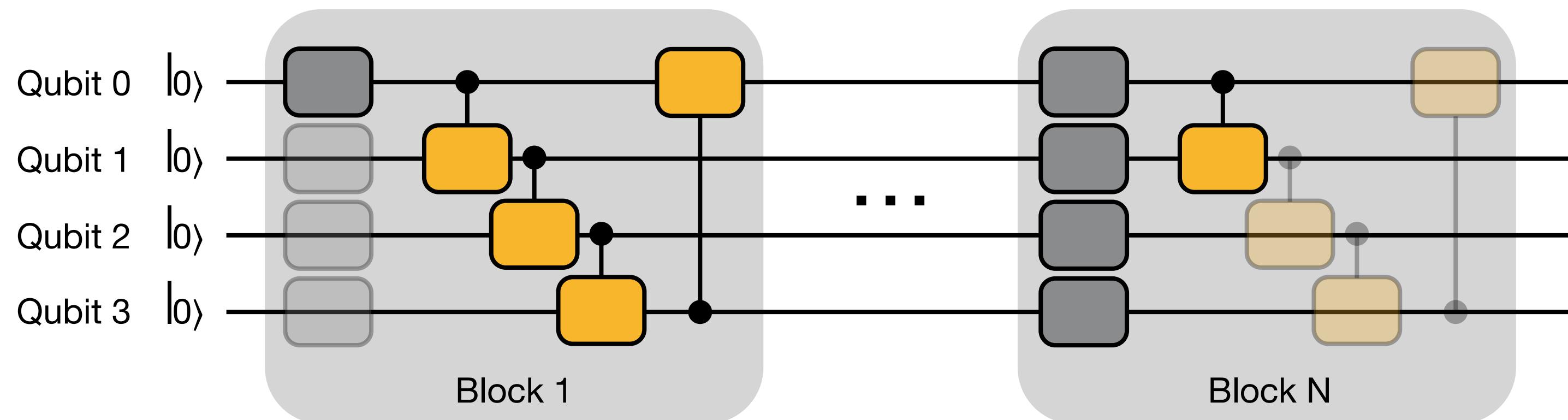
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



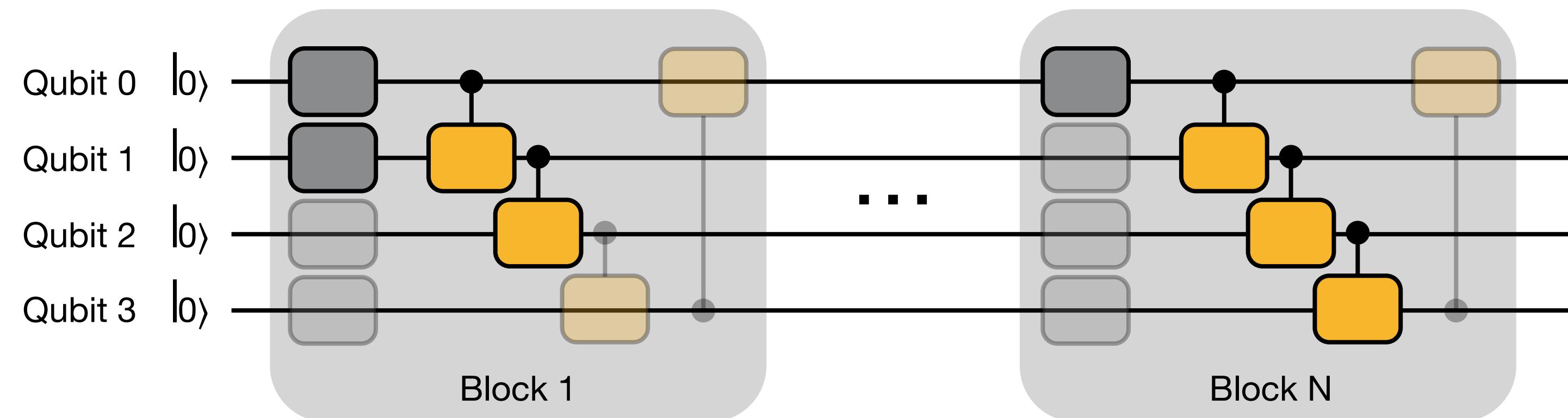
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



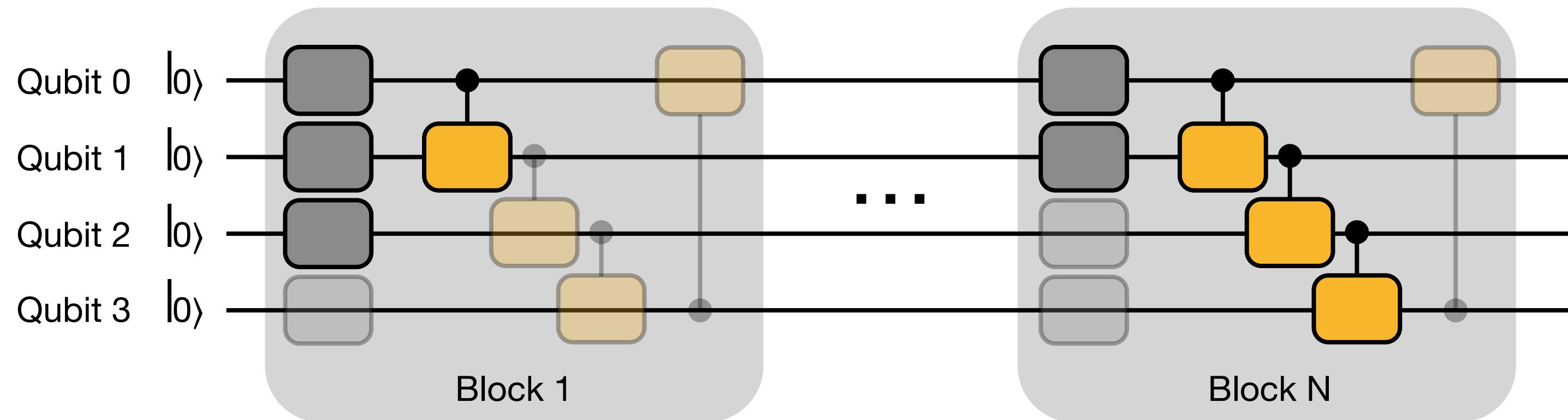
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



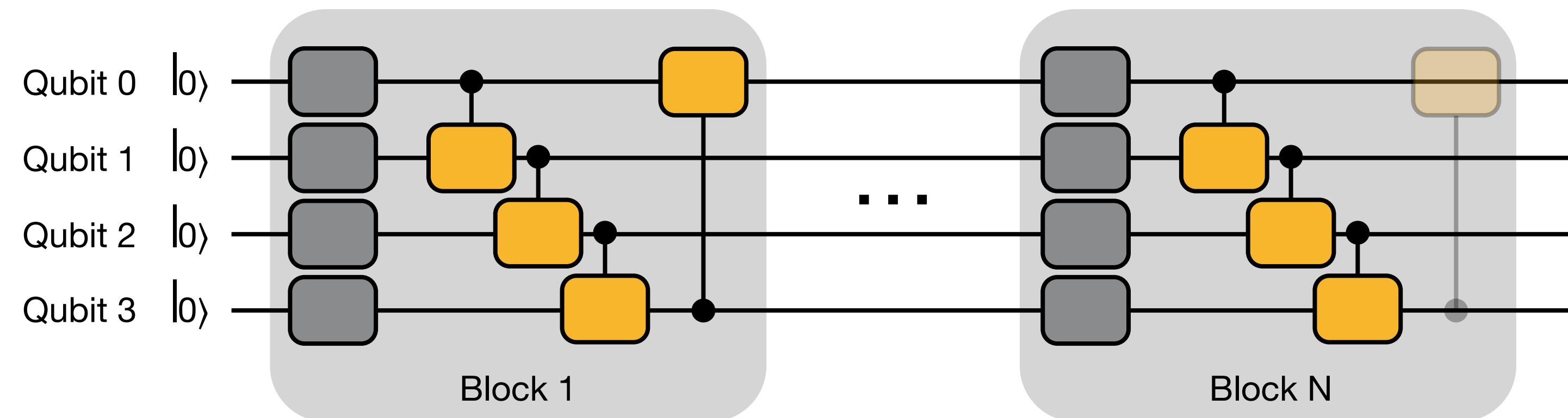
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



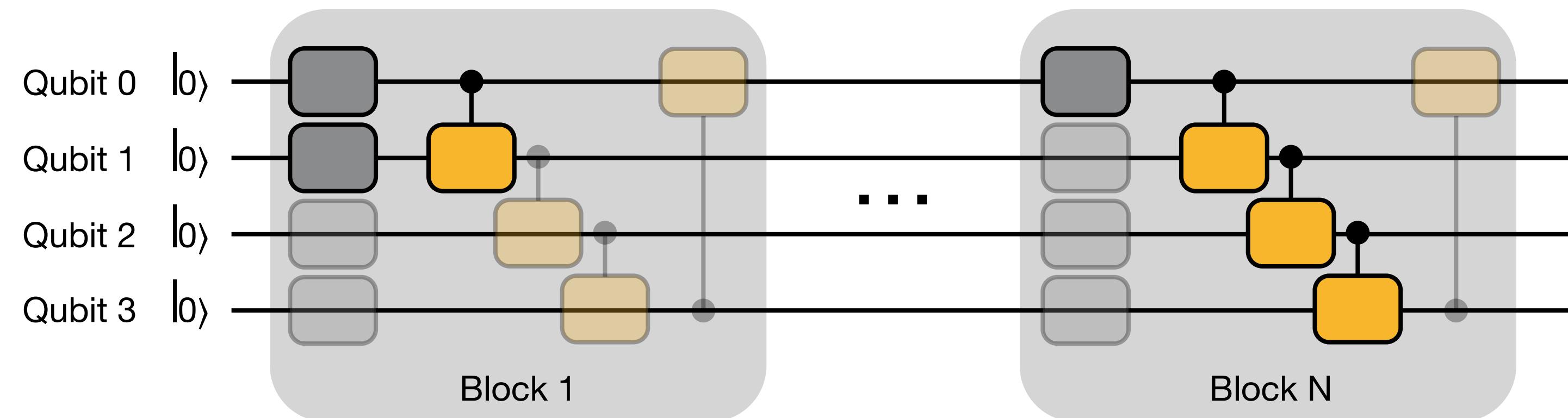
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



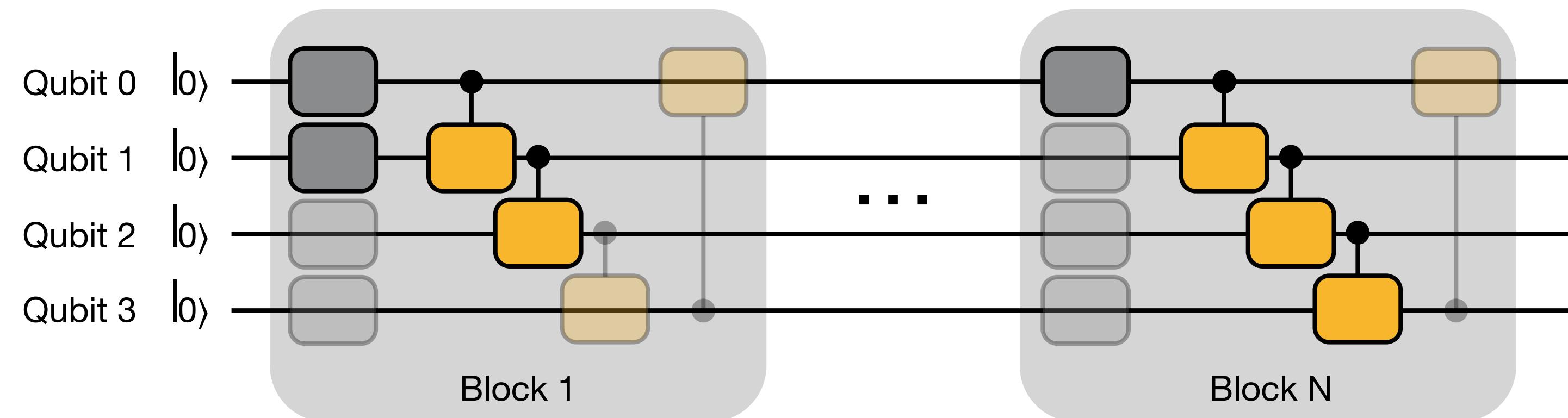
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



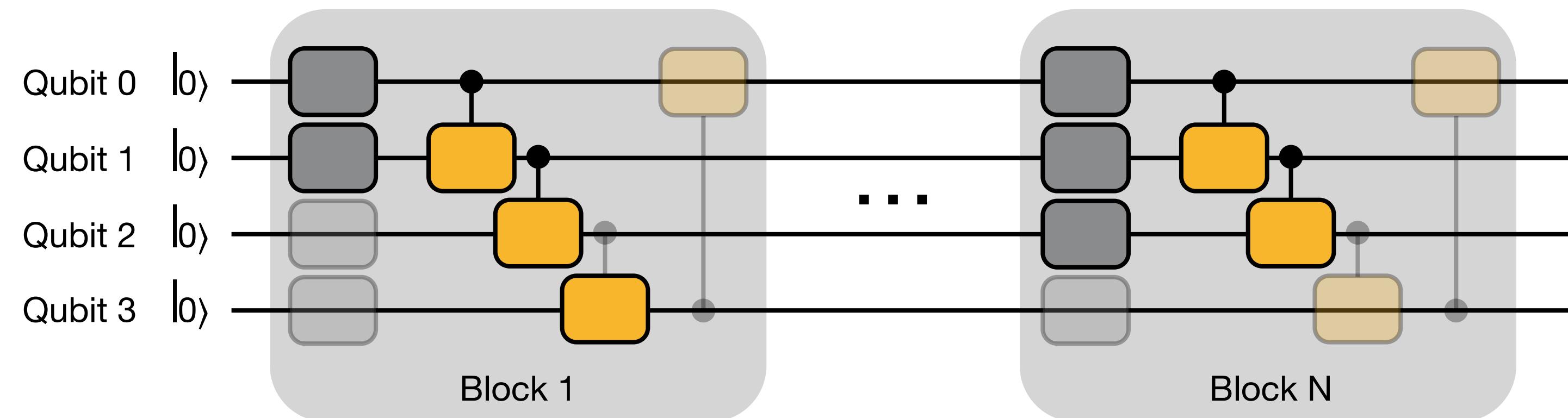
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



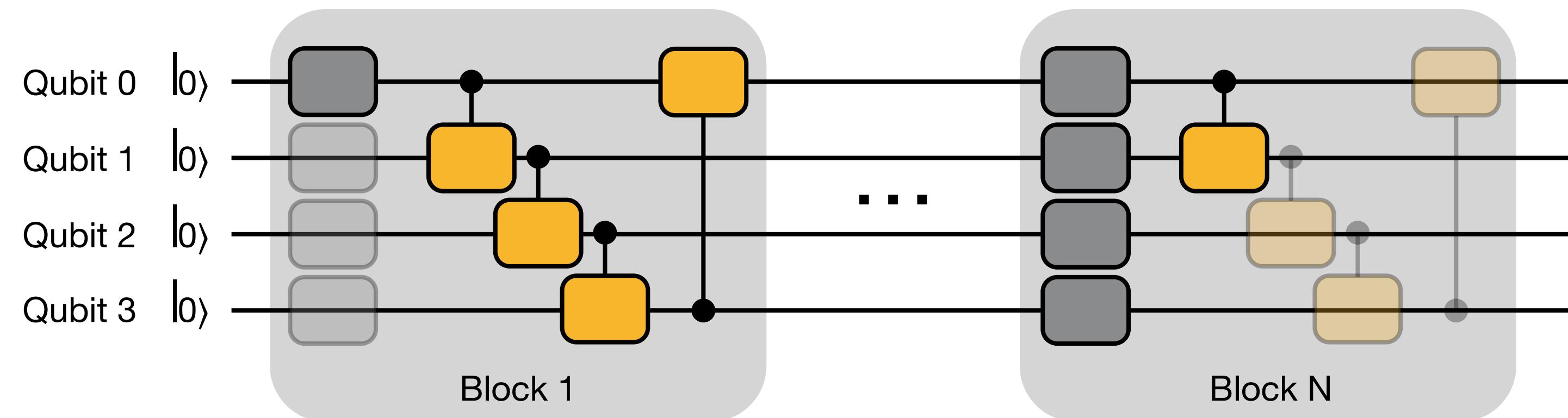
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



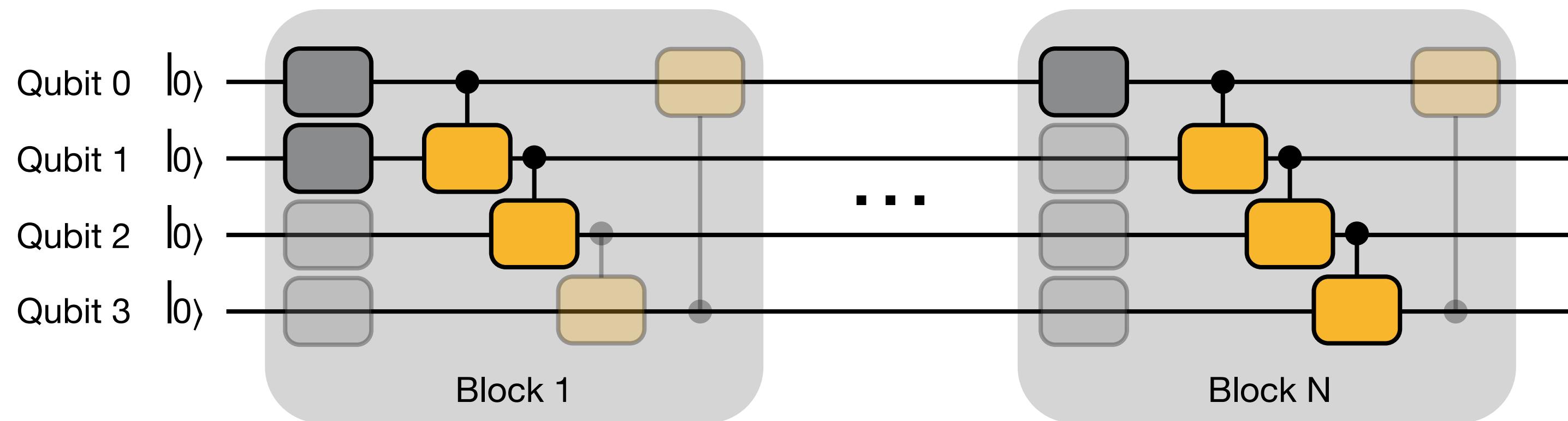
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



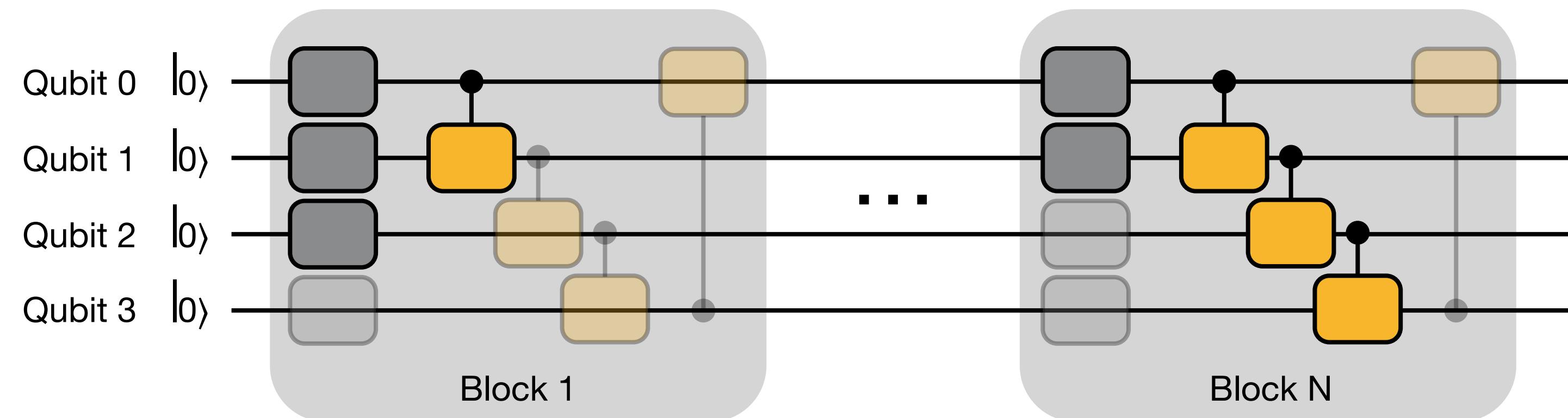
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



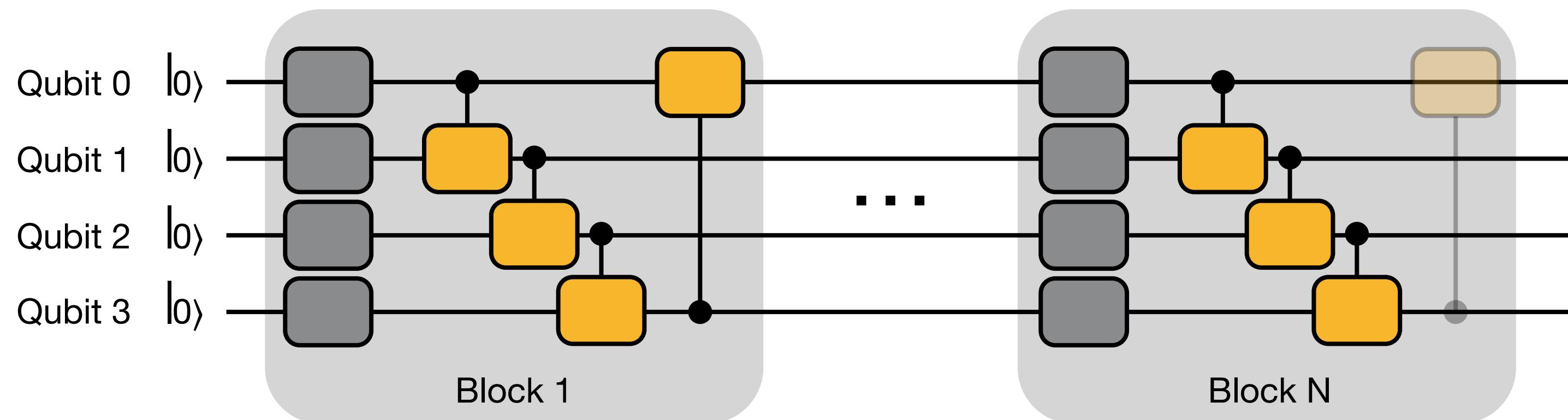
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



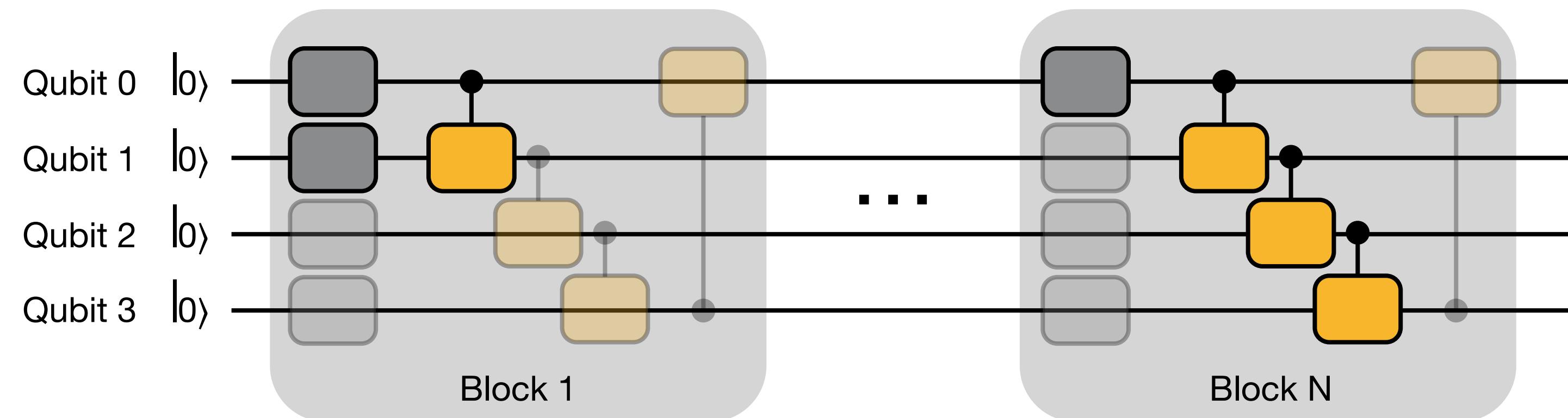
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



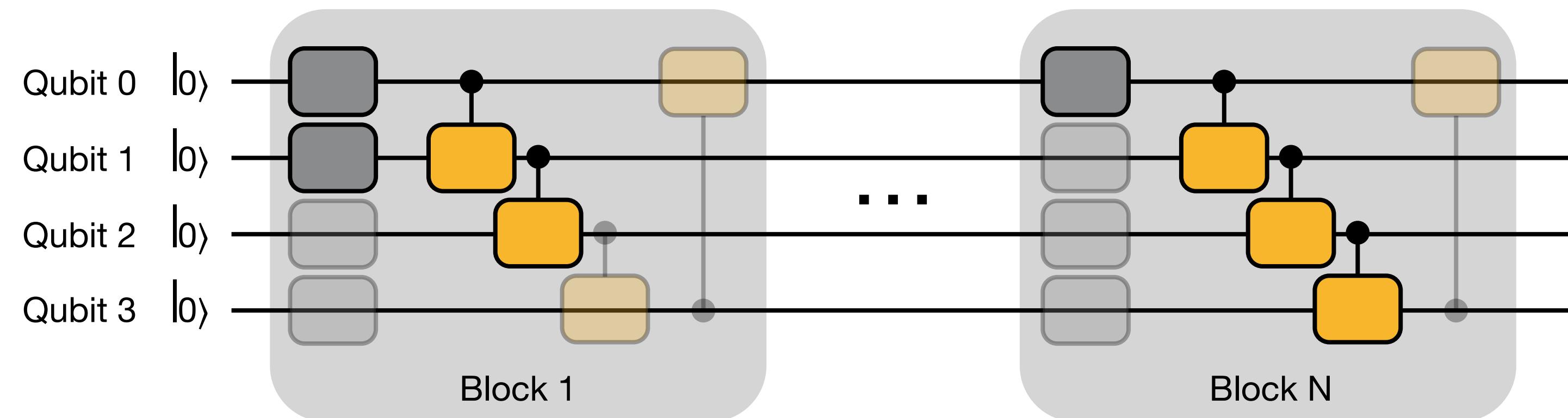
# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train

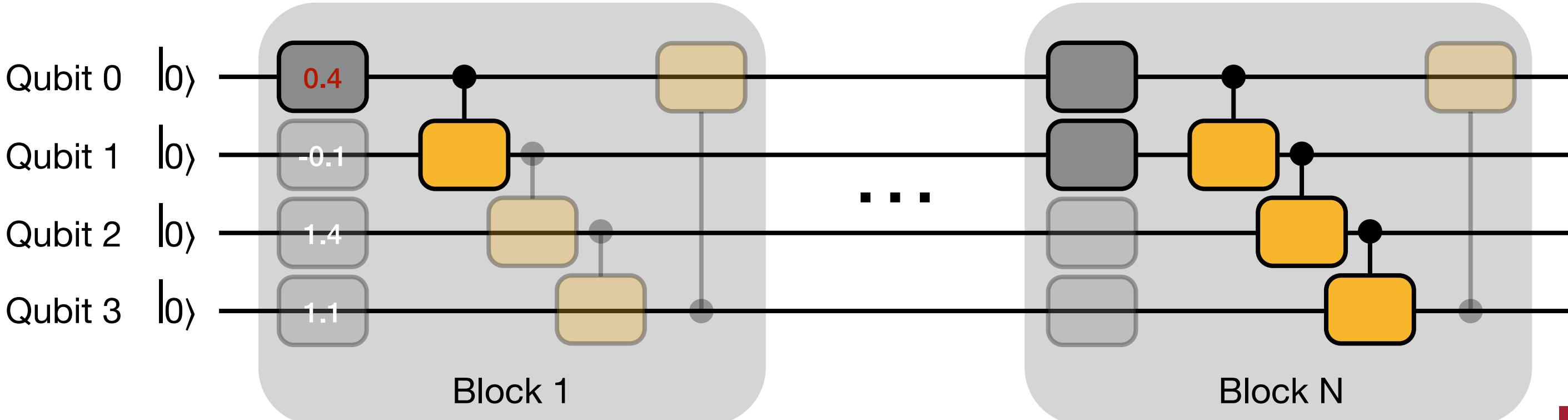
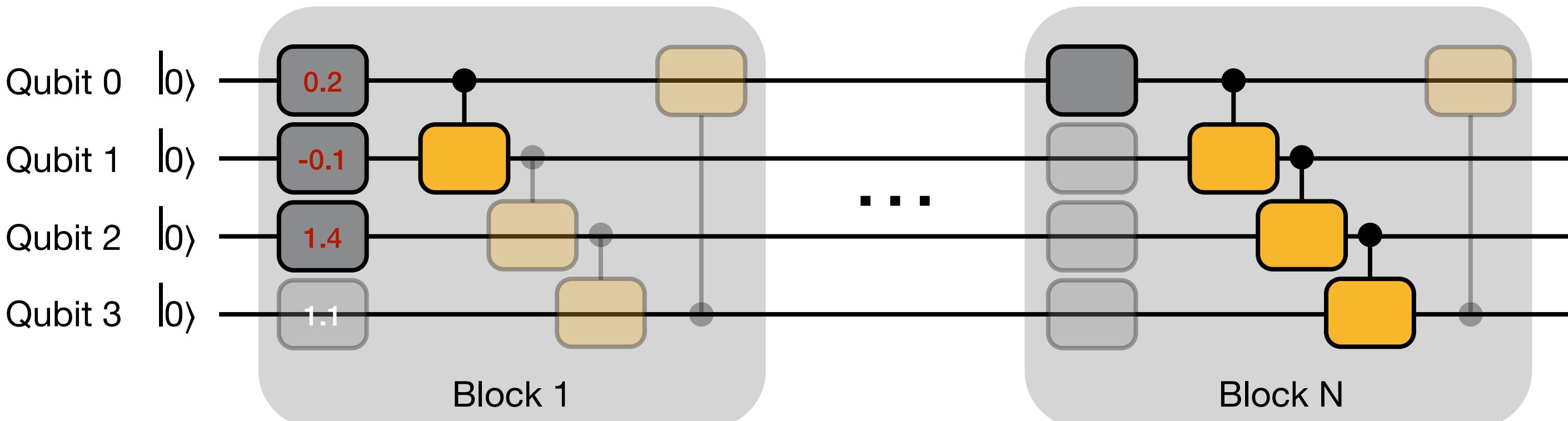
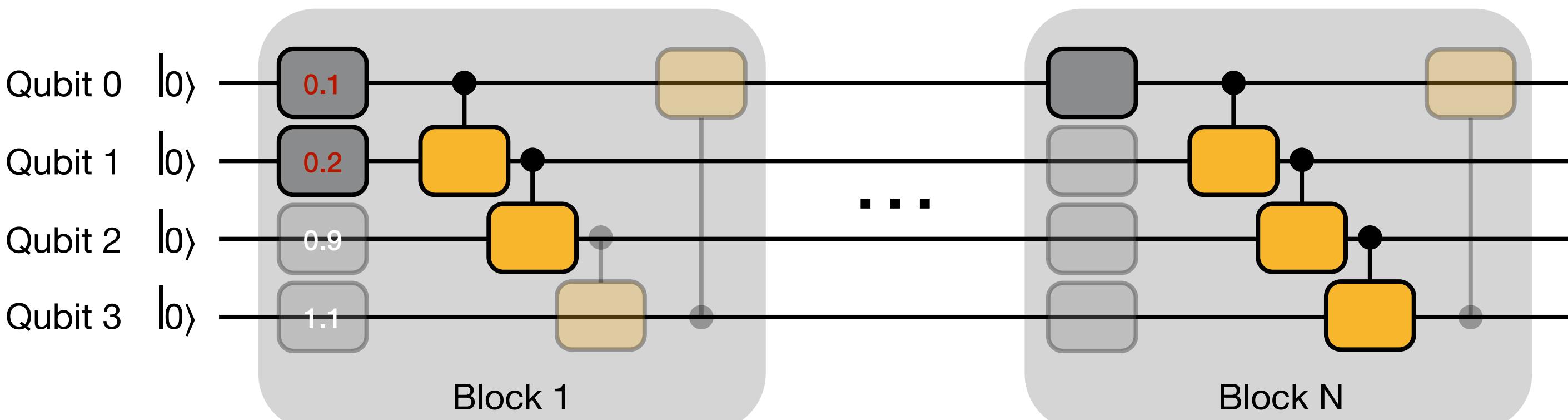


# Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train

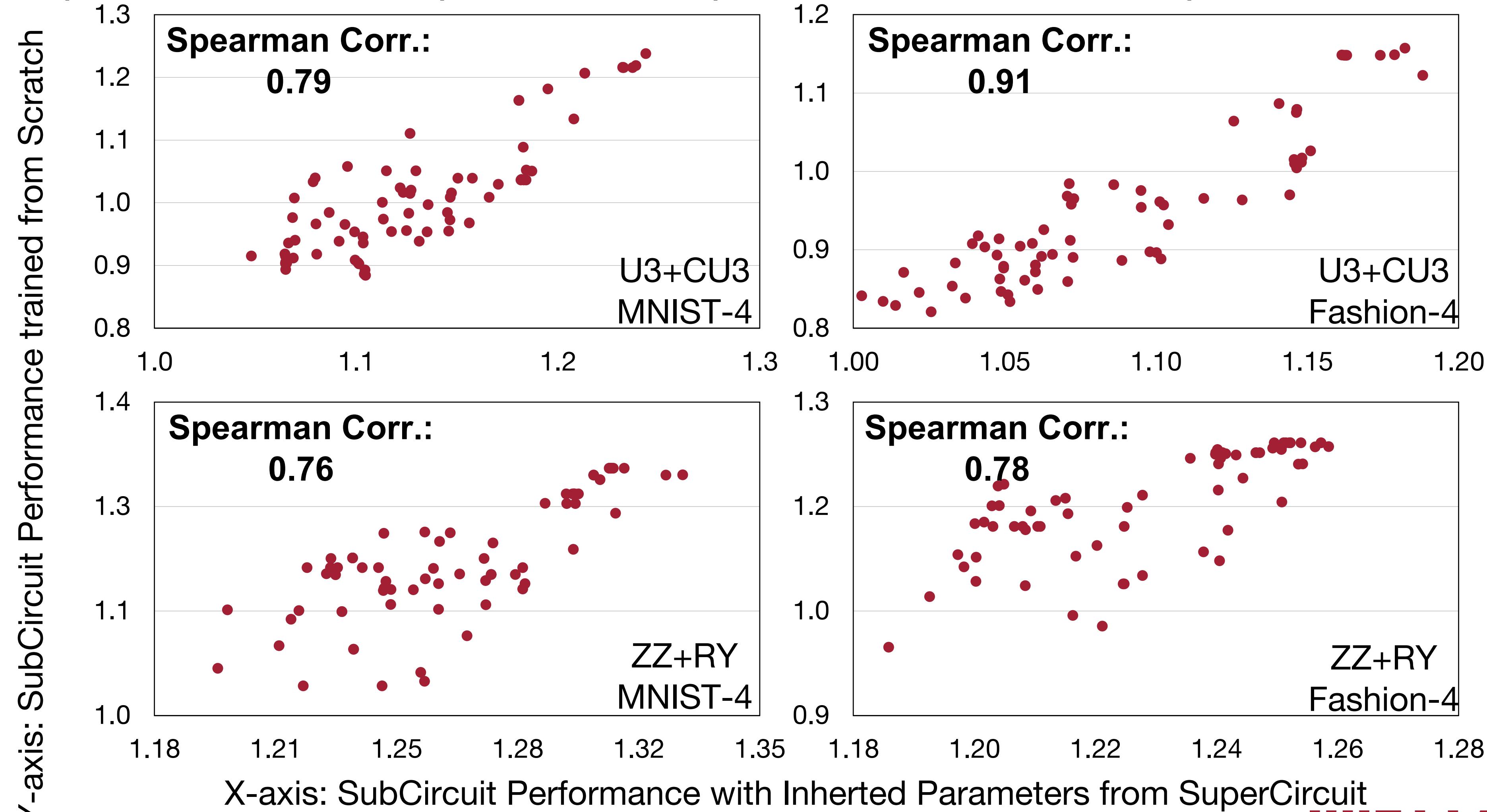


# Train SuperCircuit for Multiple Steps



# How Reliable is the SuperCircuit?

- Inherited parameters from SuperCircuit can provide accurate relative performance

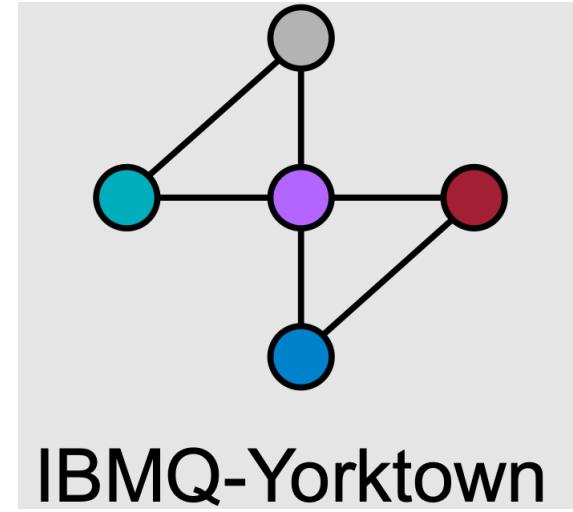


# QuantumNAS

- SuperCircuit Construction and Training
- Noise-Adaptive Evolutionary Co-Search of SubCircuit and Qubit Mapping
- Train the Searched SubCircuit
- Iterative Quantum Gate Pruning

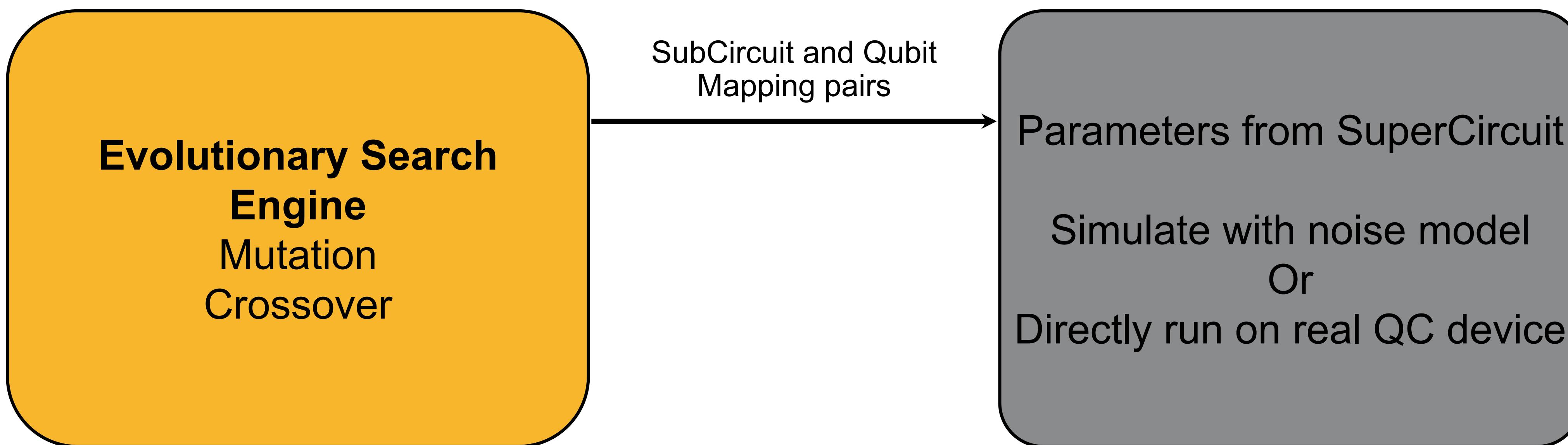
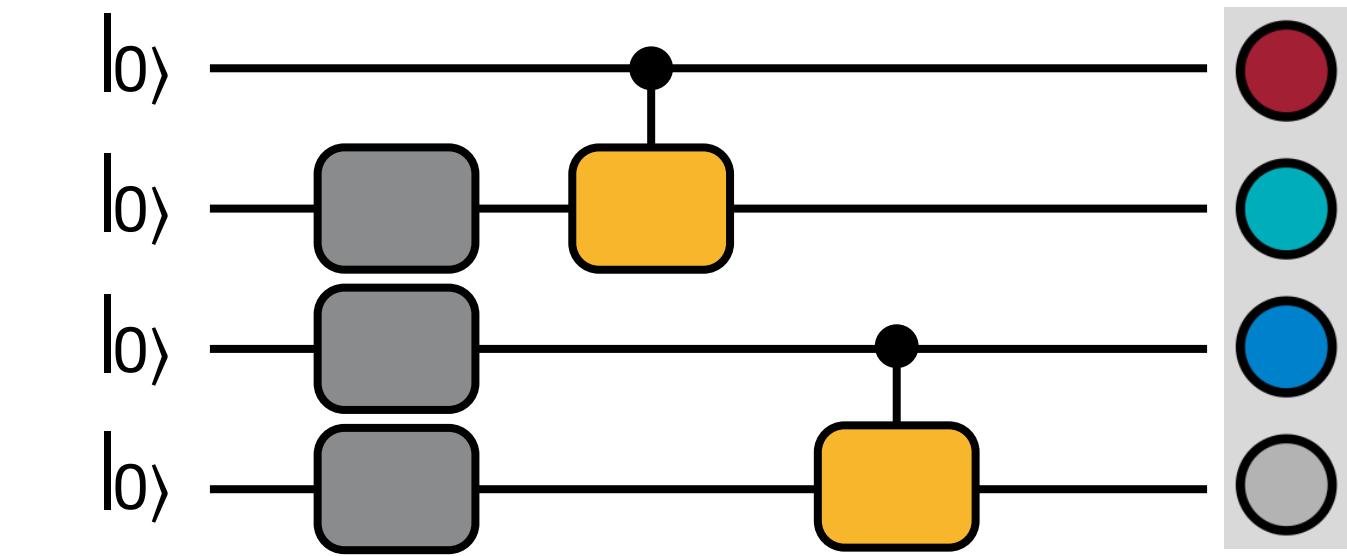
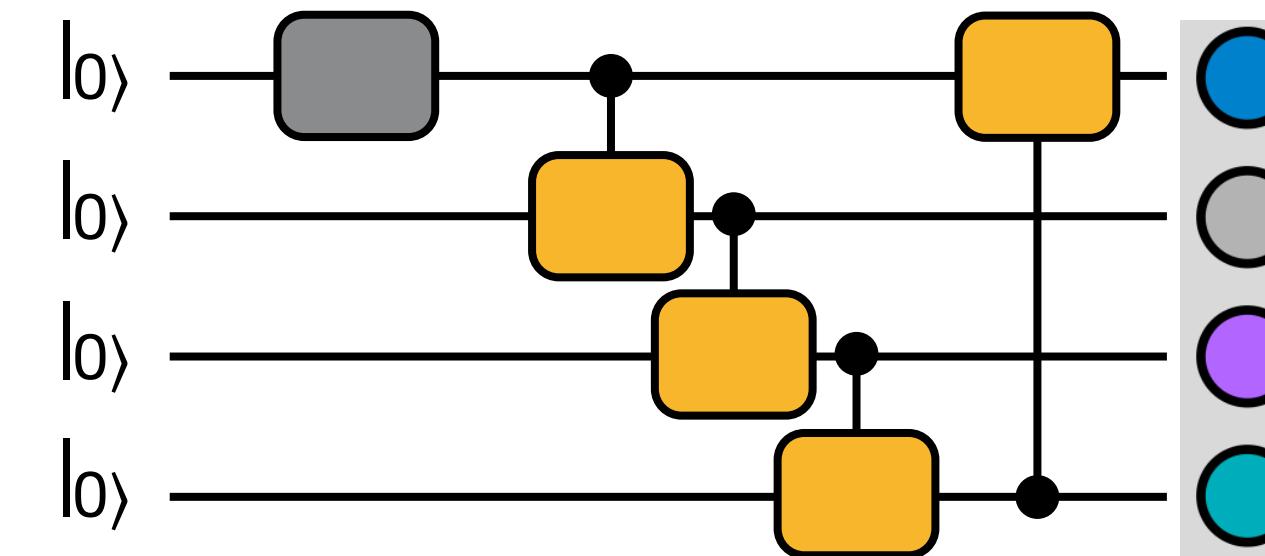
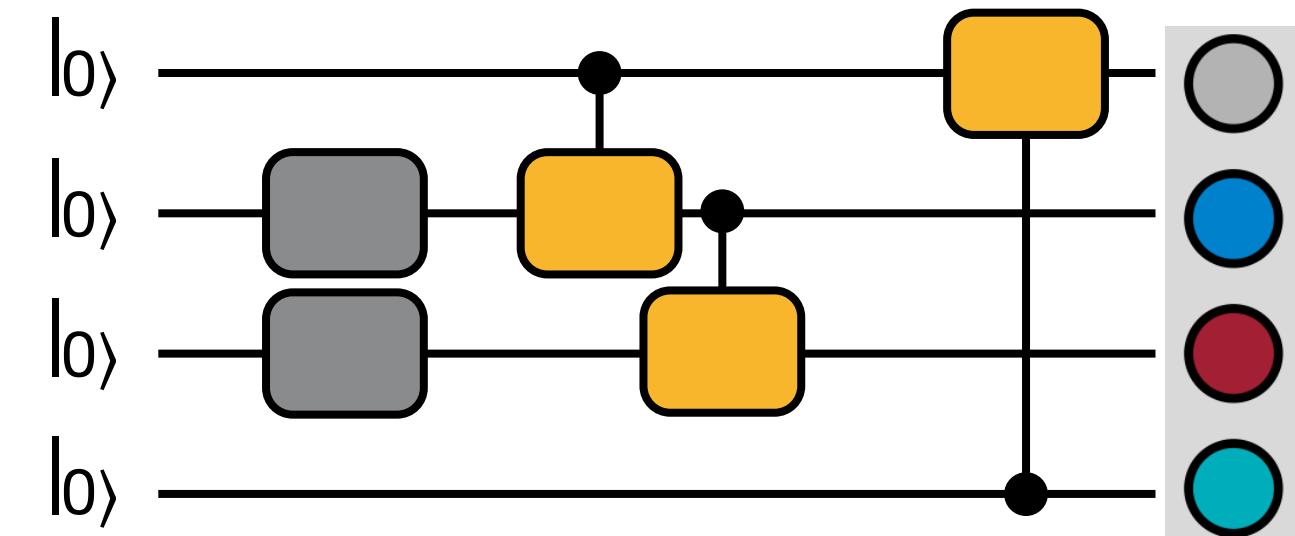
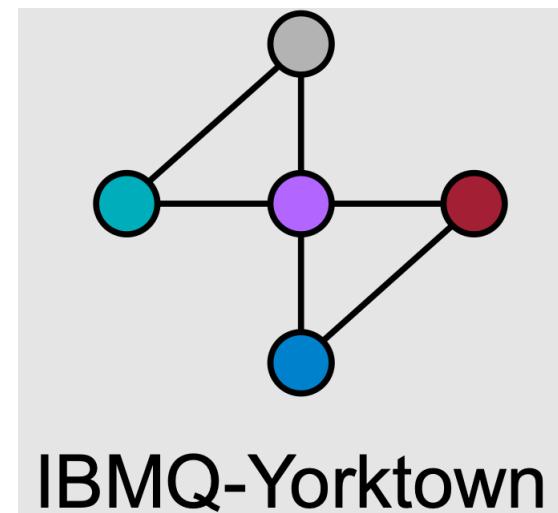
# Noise-Adaptive Evolutionary Co-Search

- Search the best SubCircuit and its qubit mapping on target device



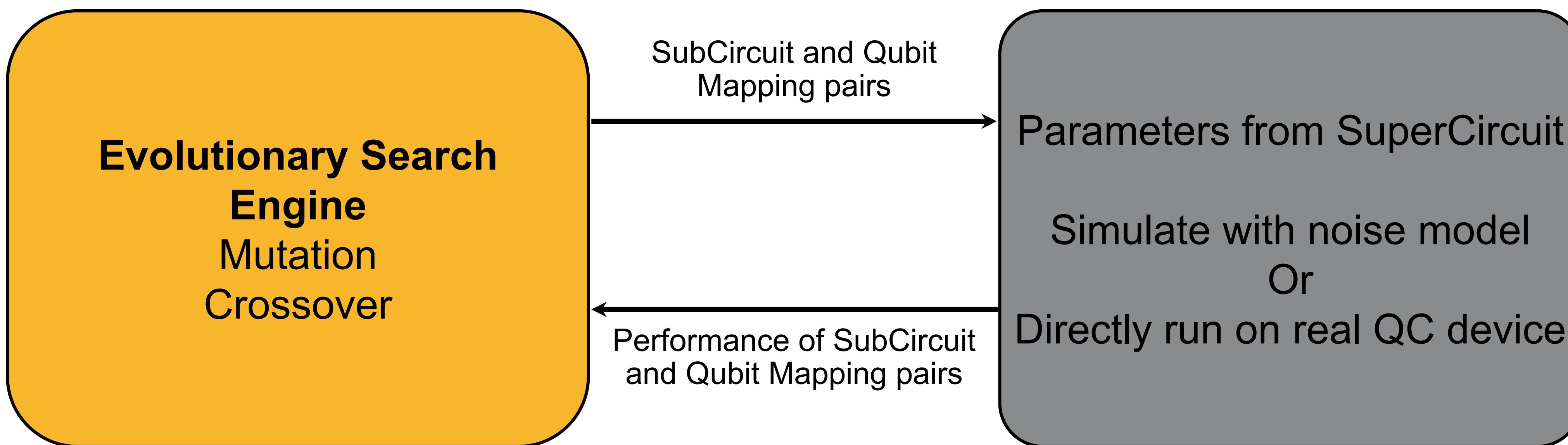
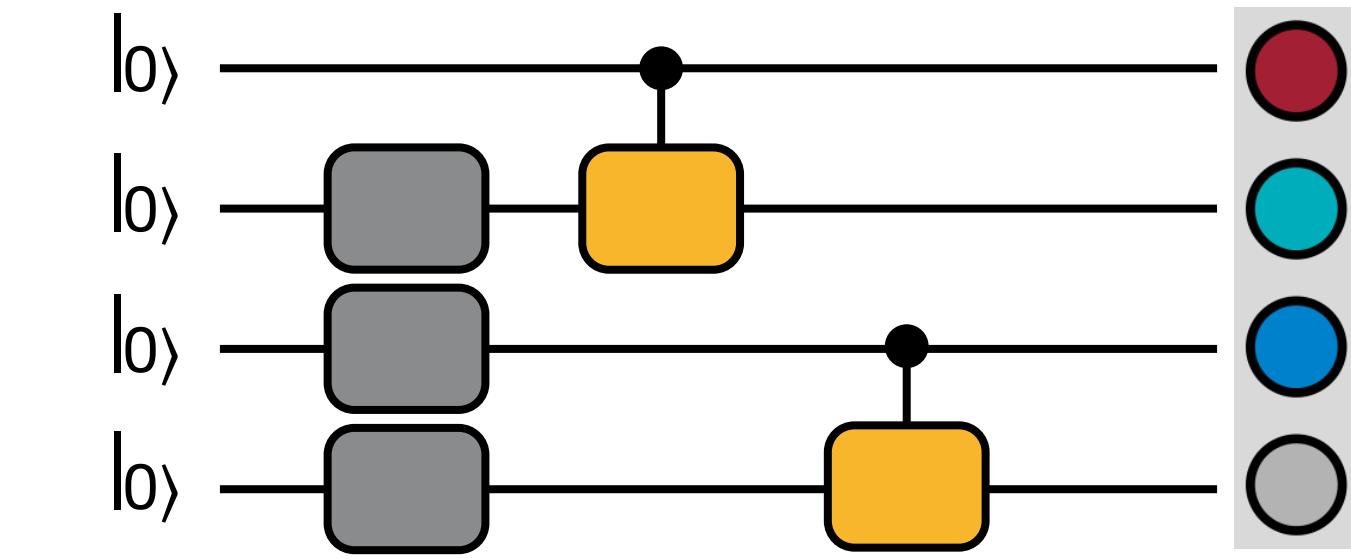
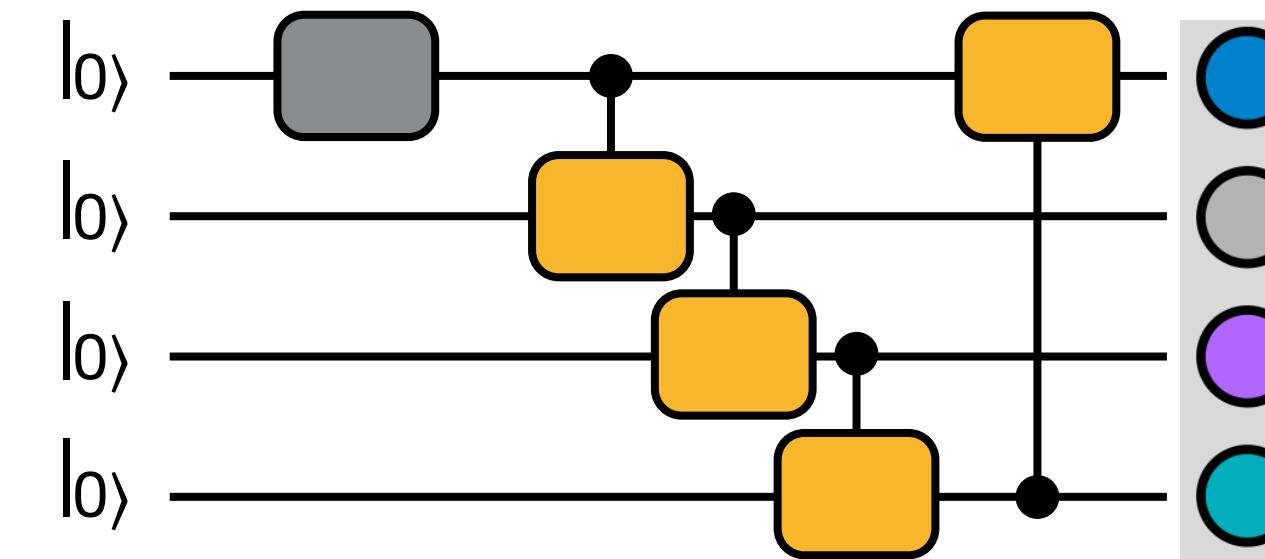
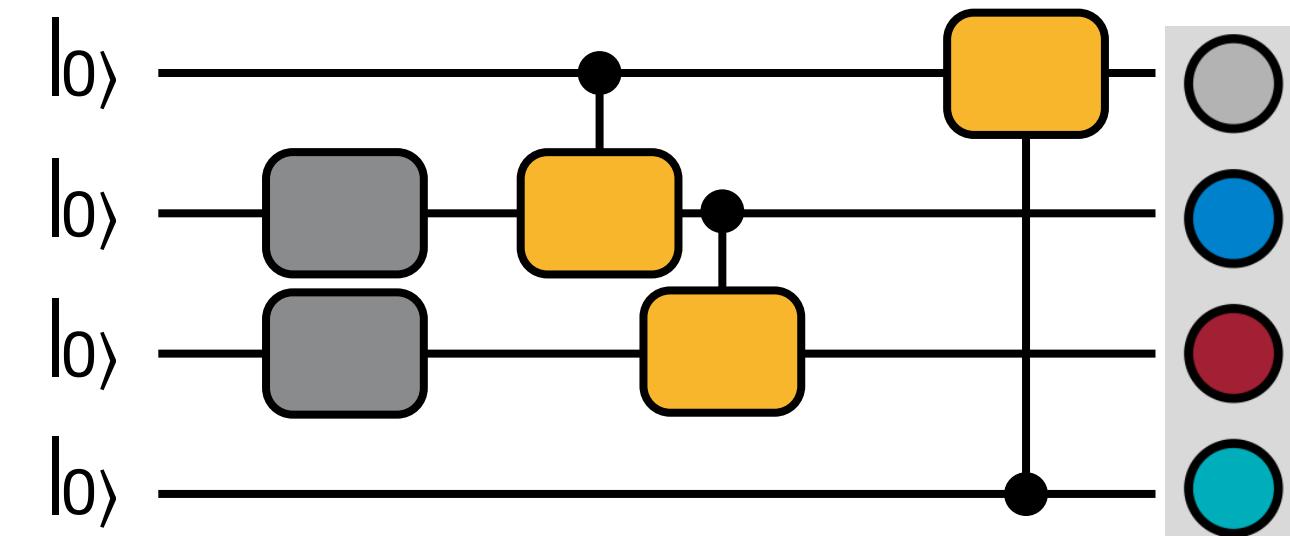
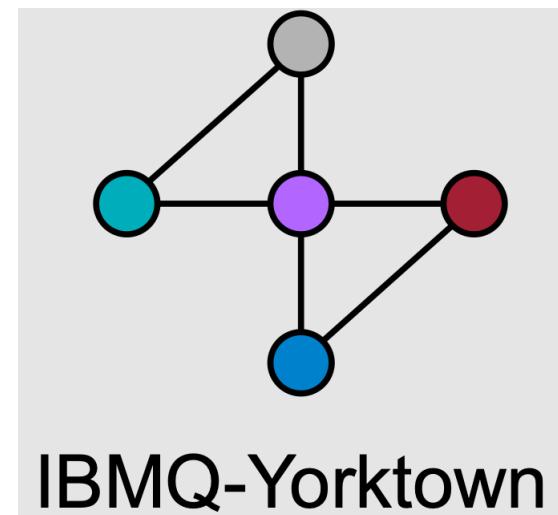
# Noise-Adaptive Evolutionary Co-Search

- Search the best SubCircuit and its qubit mapping on target device



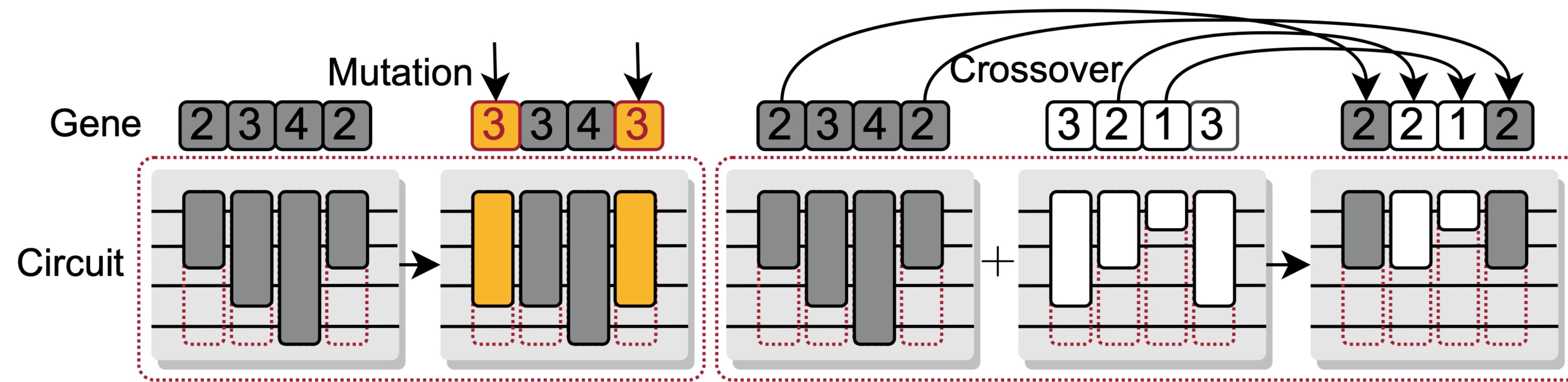
# Noise-Adaptive Evolutionary Co-Search

- Search the best SubCircuit and its qubit mapping on target device



# Mutation and Crossover

- Mutation and crossover create new SubCircuit candidates



# QuantumNAS

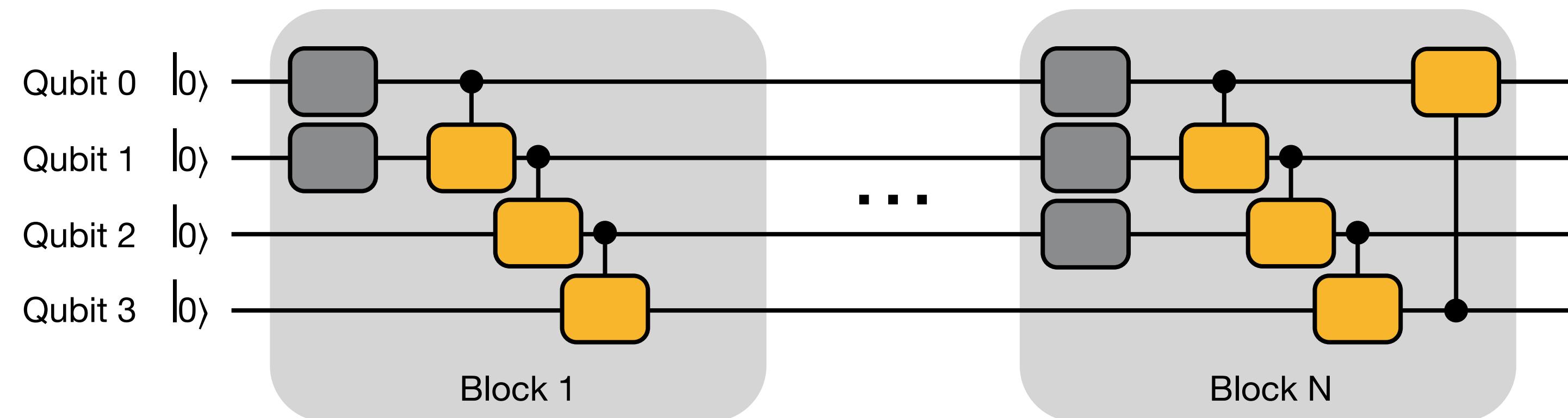
- SuperCircuit Construction and Training
- Noise-Adaptive Evolutionary Co-Search of SubCircuit and Qubit Mapping
- Train the Searched SubCircuit
- Iterative Quantum Gate Pruning

# QuantumNAS

- SuperCircuit Construction and Training
- Noise-Adaptive Evolutionary Co-Search of SubCircuit and Qubit Mapping
- Train the Searched SubCircuit
- Iterative Quantum Gate Pruning

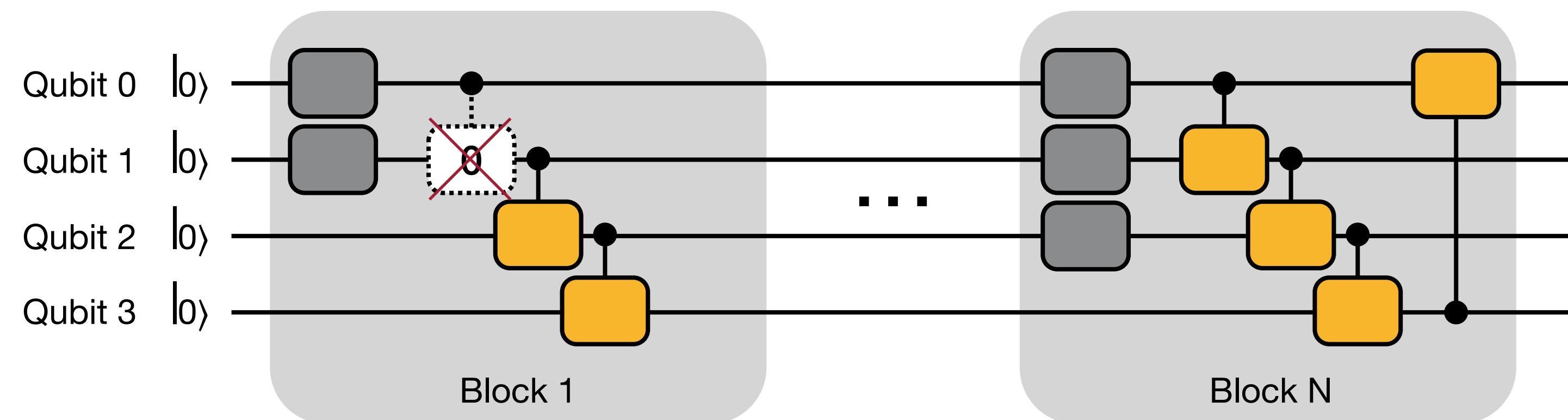
# Iterative Pruning

- Some gates have parameters close to 0
  - Rotation gate with angle close to 0 has small impact on the results
- Iteratively prune small-magnitude gates and fine-tune the remaining parameters



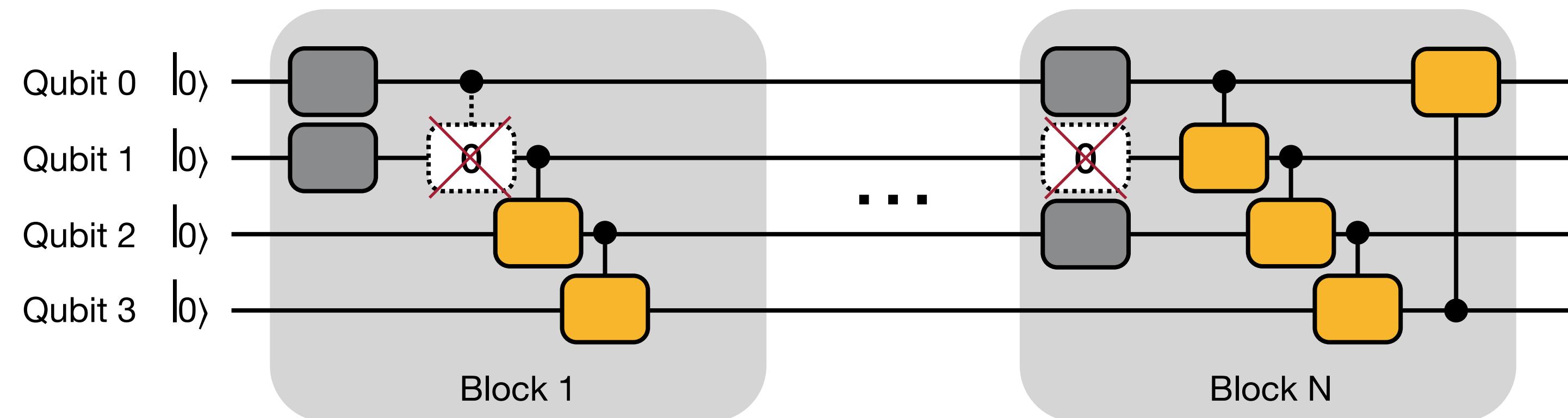
# Iterative Pruning

- Some gates have parameters close to 0
  - Rotation gate with angle close to 0 has small impact on the results
- Iteratively prune small-magnitude gates and fine-tune the remaining parameters



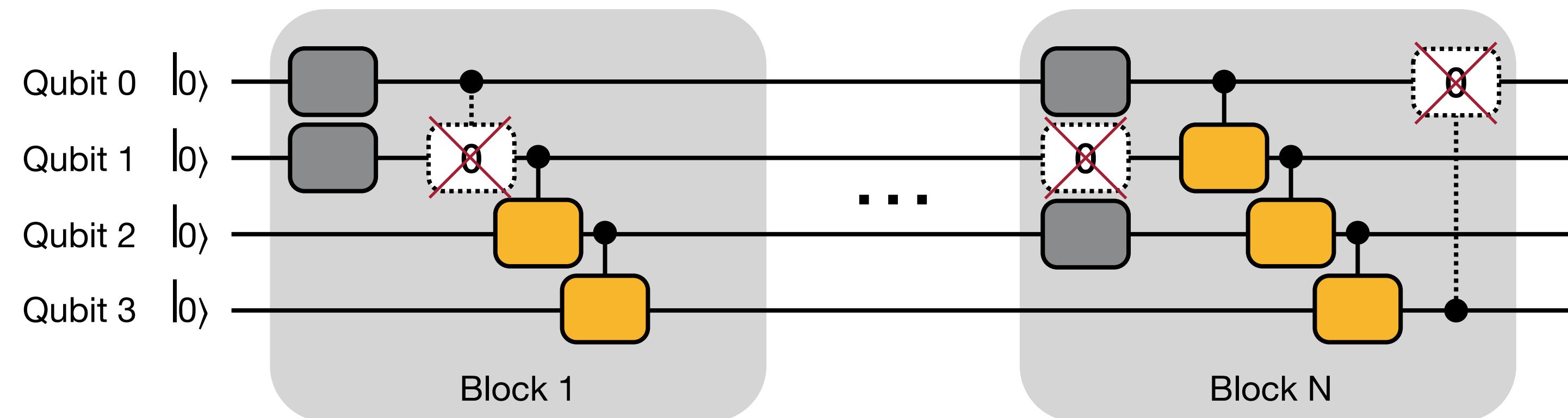
# Iterative Pruning

- Some gates have parameters close to 0
  - Rotation gate with angle close to 0 has small impact on the results
- Iteratively prune small-magnitude gates and fine-tune the remaining parameters



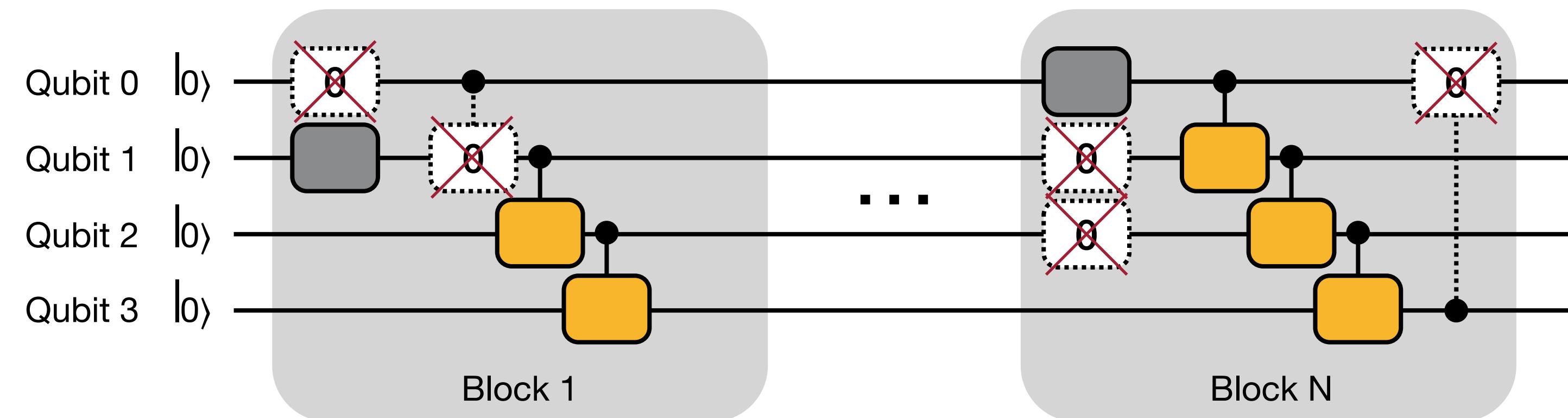
# Iterative Pruning

- Some gates have parameters close to 0
  - Rotation gate with angle close to 0 has small impact on the results
- Iteratively prune small-magnitude gates and fine-tune the remaining parameters



# Iterative Pruning

- Some gates have parameters close to 0
  - Rotation gate with angle close to 0 has small impact on the results
- Iteratively prune small-magnitude gates and fine-tune the remaining parameters

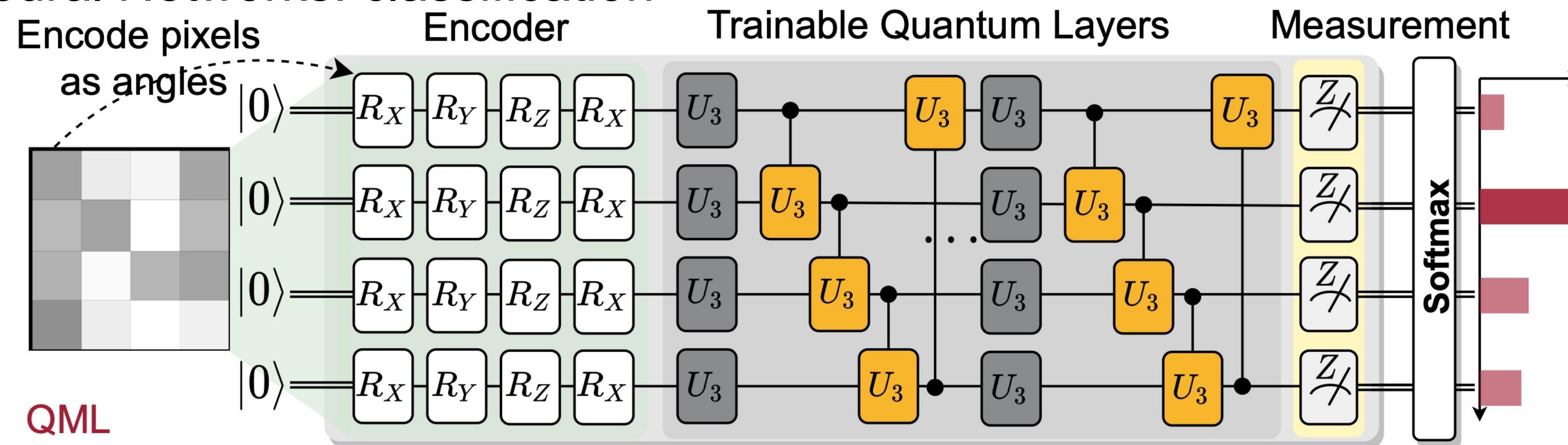


# Evaluation Setups: Benchmarks and Devices

- Benchmarks
  - QML classification tasks: MNIST 10-class, 4-class, 2-class, Fashion 4-class, 2-class, Vowel 4-class
  - VQE task molecules: H<sub>2</sub>, H<sub>2</sub>O, LiH, CH<sub>4</sub>, BeH<sub>2</sub>
- Quantum Devices
  - IBMQ
  - #Qubits: 5 to 65
  - Quantum Volume: 8 to 128

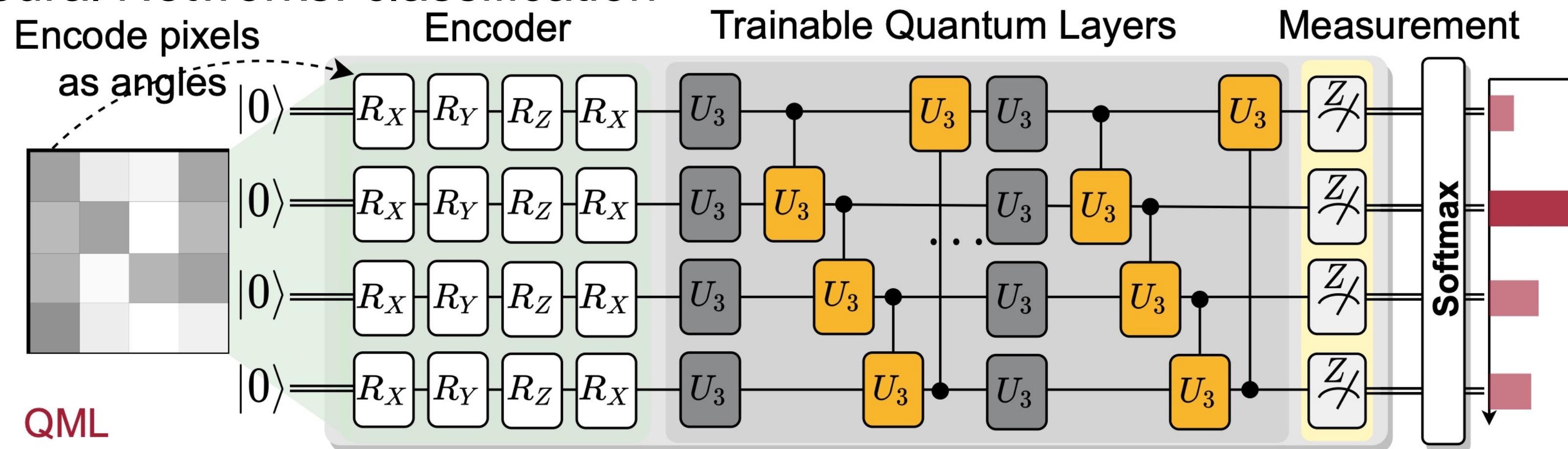
# Benchmarks: QNN and VQE

- Quantum Neural Networks: classification

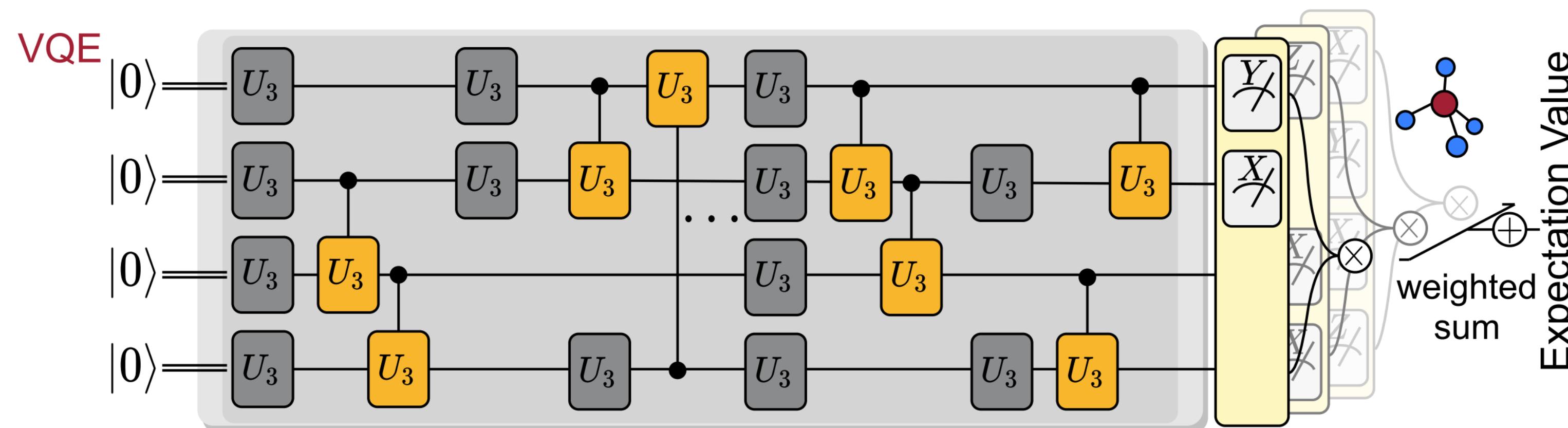


# Benchmarks: QNN and VQE

- Quantum Neural Networks: classification

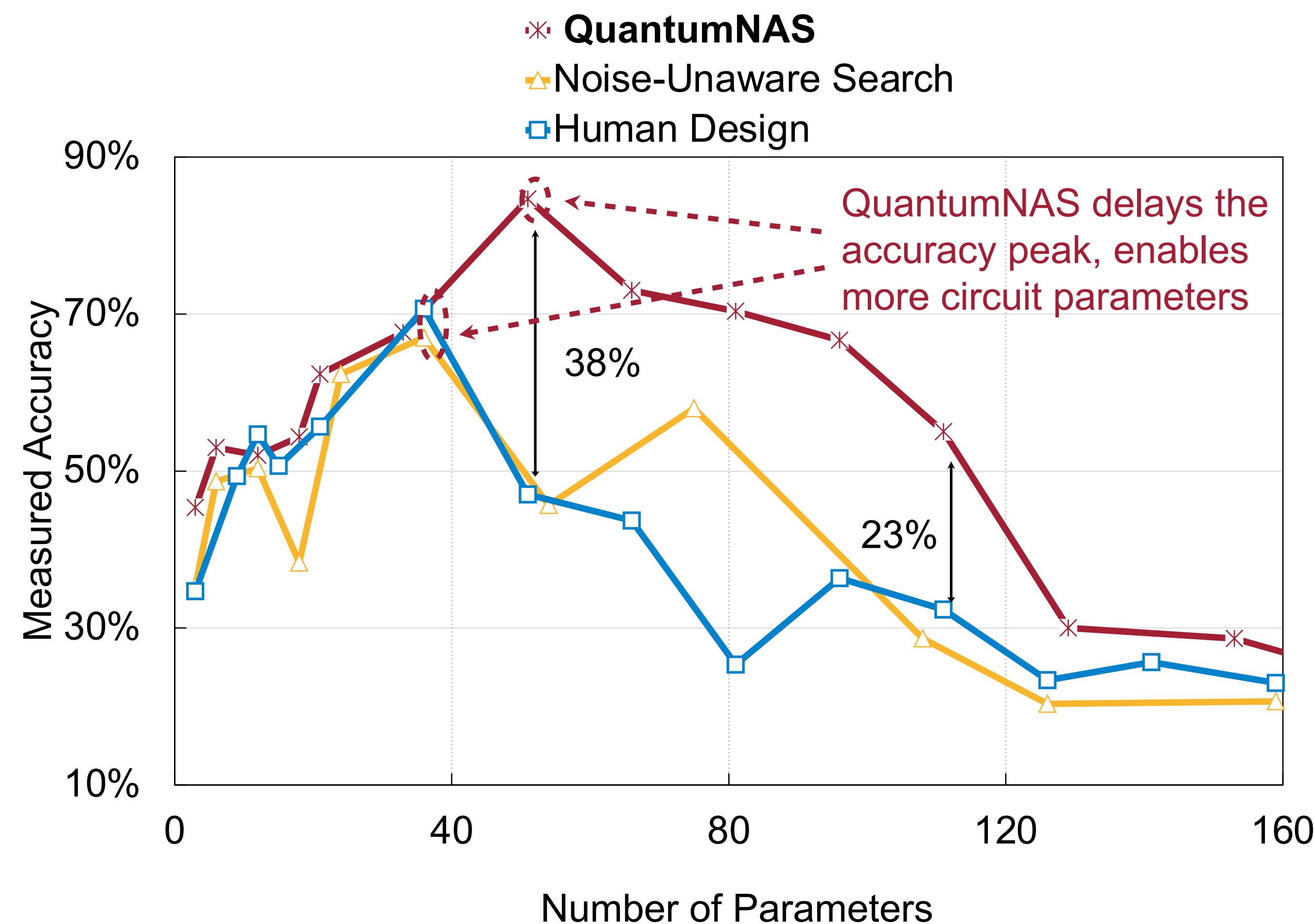


- Variational Quantum Eigensolver: finds the ground state energy of molecule Hamiltonian



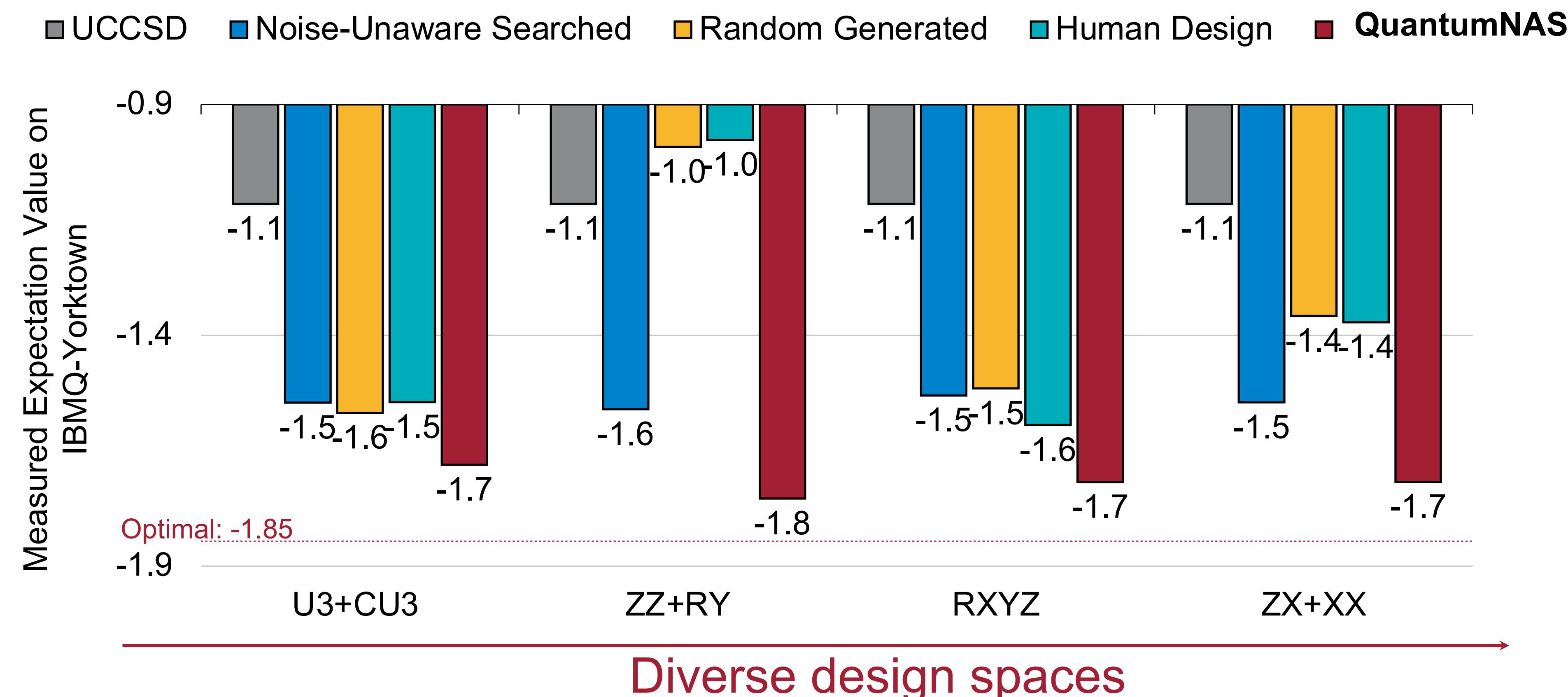
# QML Results

- 4-classification: MNIST-4 U3+CU3 on IBMQ-Yorktown



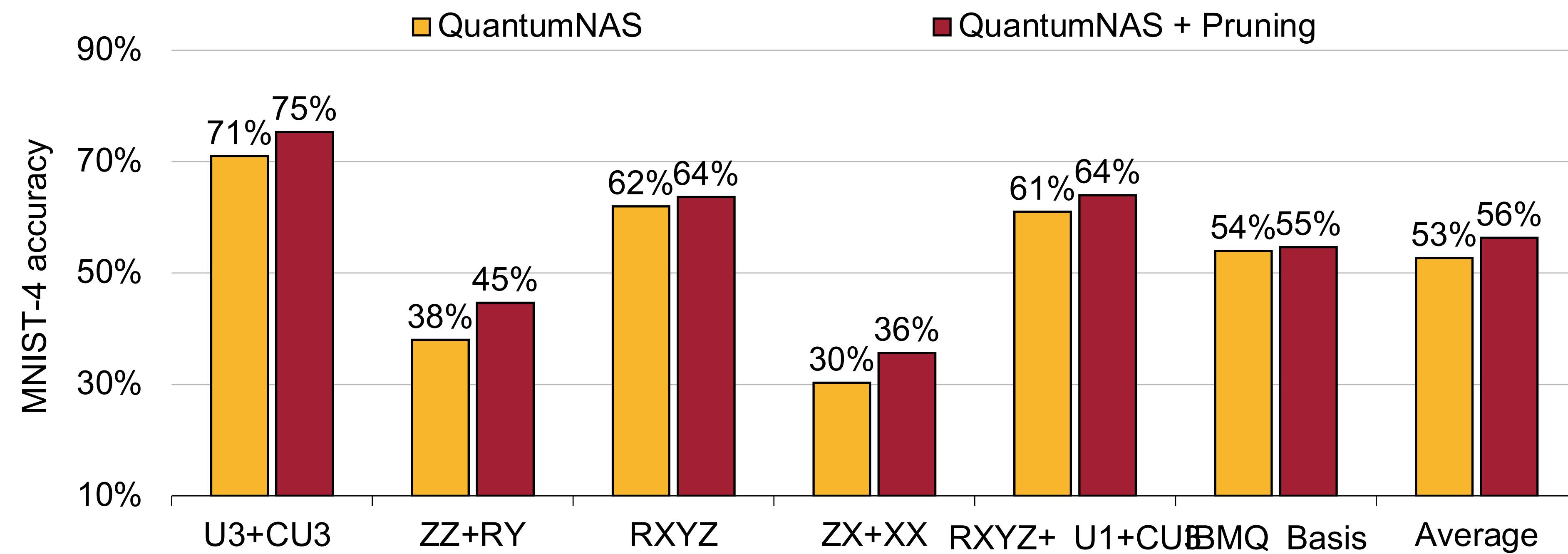
# Consistent Improvements on Diverse Design Spaces

- H2 in different design spaces on IBMQ-Yorktown



# Effectiveness of Quantum Gate Pruning

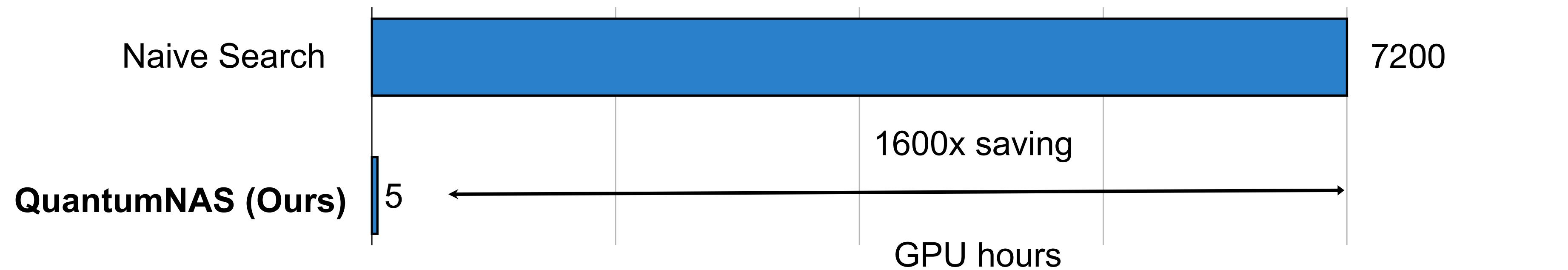
- For MNIST-4, Quantum gate pruning improves accuracy by 3% on average



# Time Cost with TorchQuantum

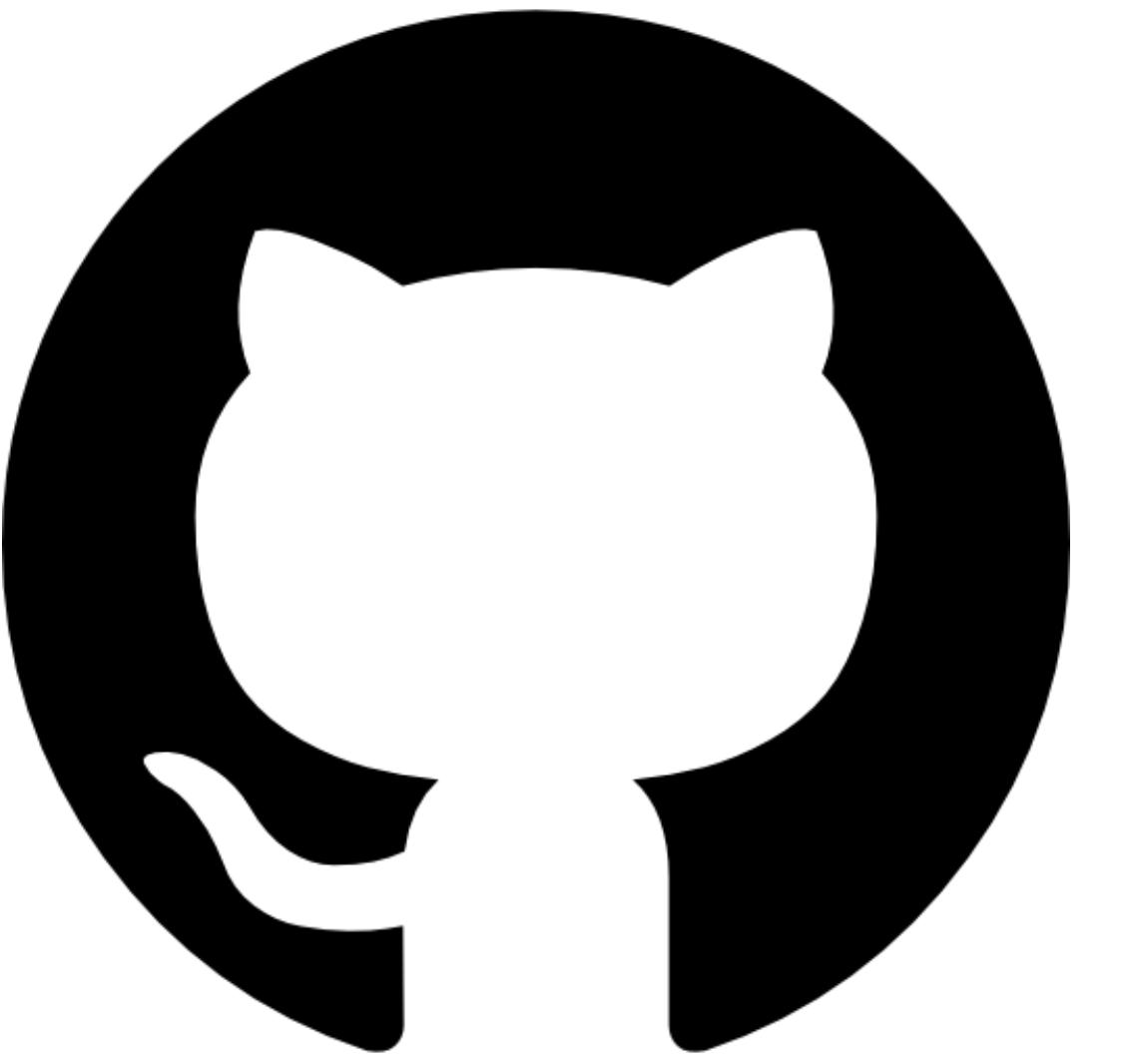
- On 1 Nvidia Titan RTX 2080 ti GPU

#qubits	Step	SuperCircuit Training	Noise-Adaptive Co-search	SubCircuit Training	Deployment on Real QC
4 Qubits		0.5h	3h	0.5h	0.5h
15 Qubits		5h	5h	5h	1h
21 Qubits		20h	10h	15h	1h



# Hands-On Section

## 2.1 QuantumNAS



# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations

1.3 TQ for State Prep

1.4 TQ for VQE

1.4 TQ for QNN

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning

2.2 QuantumNAT: Noise Injection and Quantization

2.3 QOC: On-Chip Training

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression

## Section 3

### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control

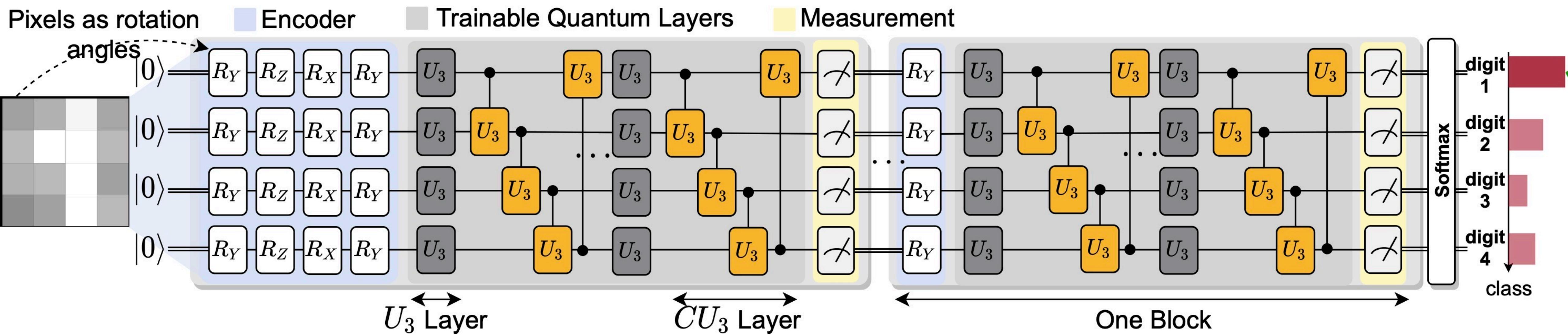
3.2 Variational Pulse Learning

# QuantumNAS vs. QuantumNAT

- **QuantumNAS** finds noise robust circuit **architecture**
- **QuantumNAT** finds noise robust circuit **parameters**

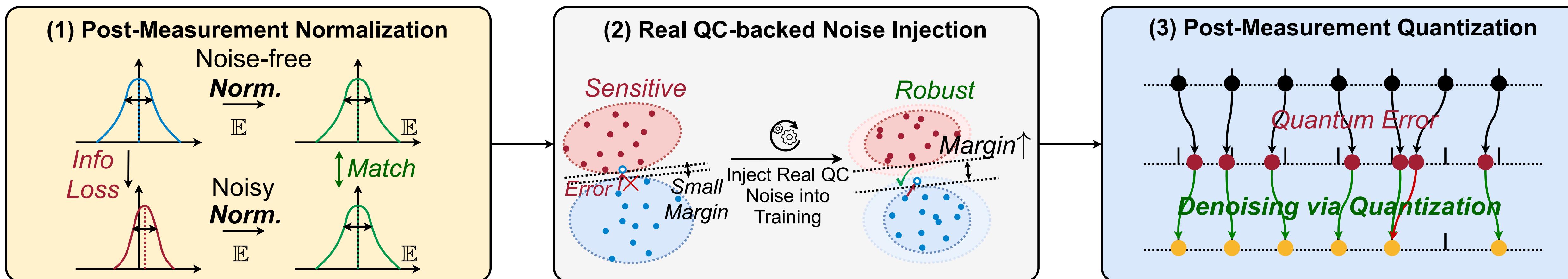
# PQC Circuit in QuantumNAT

- QNN with multiple nodes
  - Encoder
  - Trainable Quantum Layers
  - Measurements

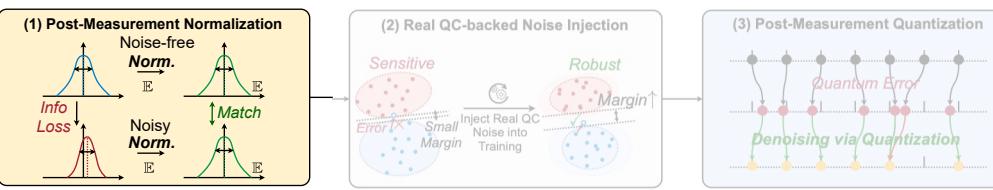


# Three Techniques in QuantumNAT

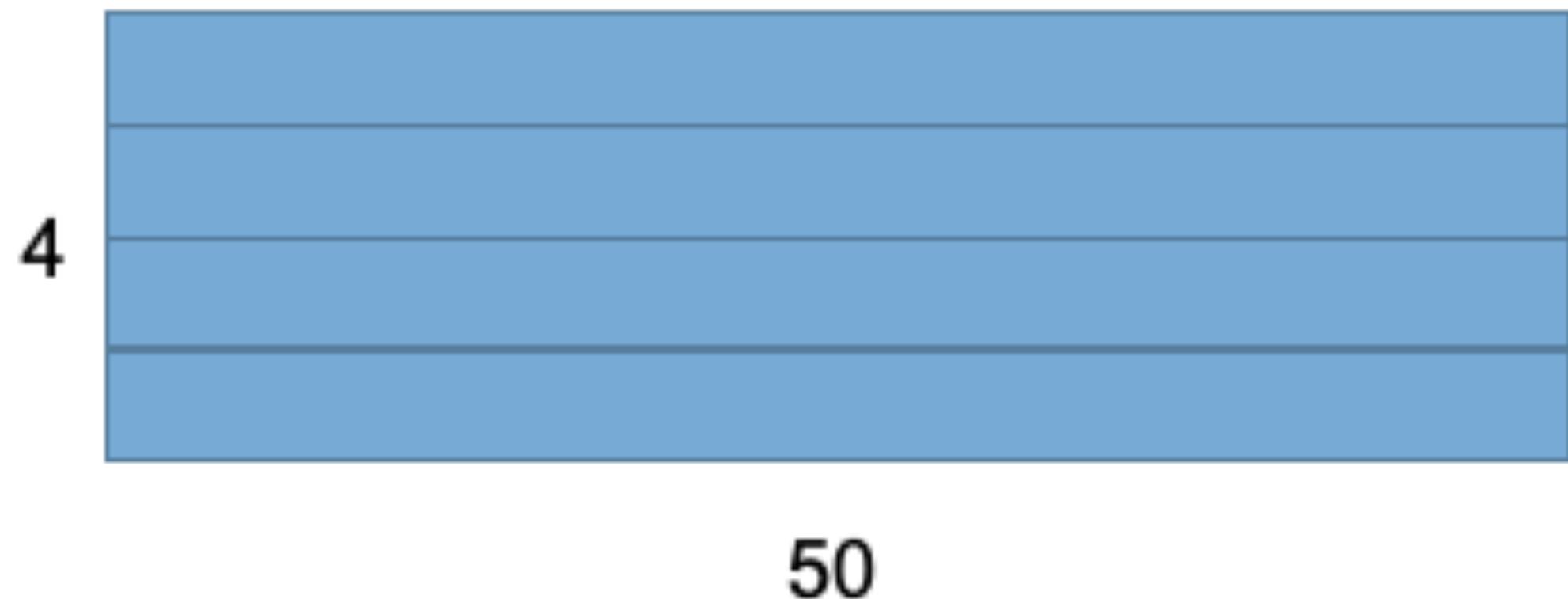
- QuantumNAT:
  - Normalization: mitigate noise impact
  - Noise injection: make the parameters aware of noise
  - Quantization: mitigate noise impact



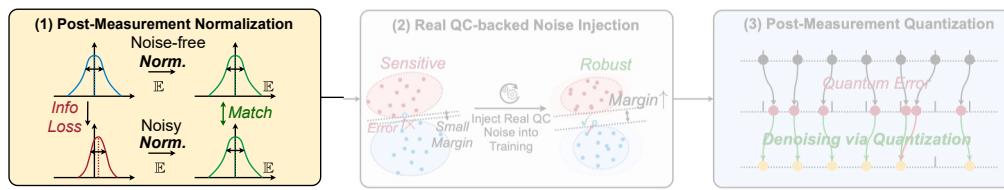
# Post-Measurement Normalization



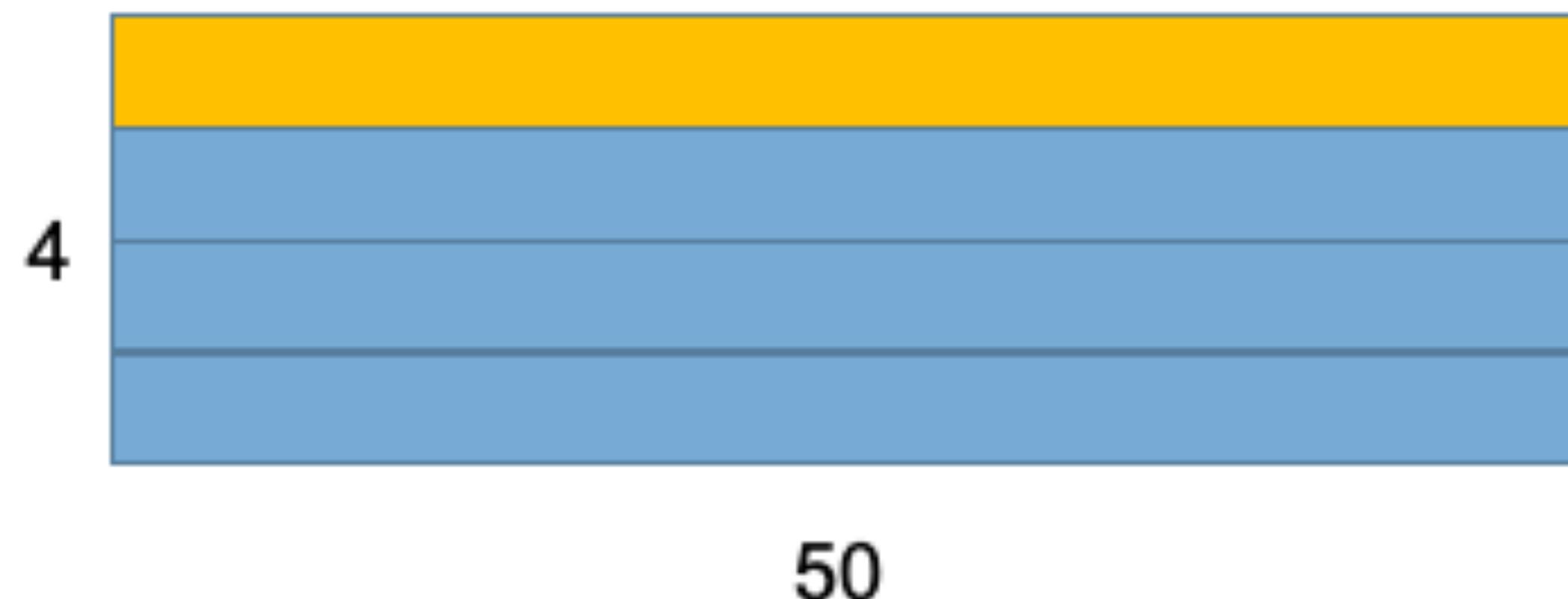
- Normalize the measurement outcome
  - Along the **batch** dimension
  - For example, we train the 4-qubit PQC with batch=50 then we have results as 50 \* 4 values



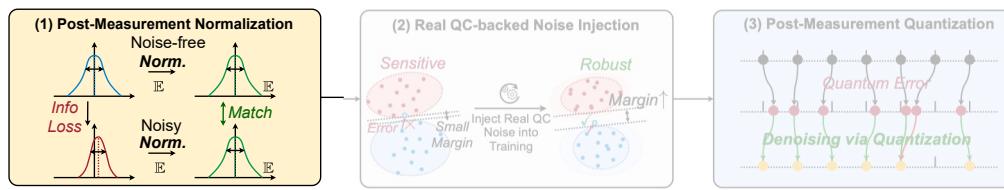
# Post-Measurement Normalization



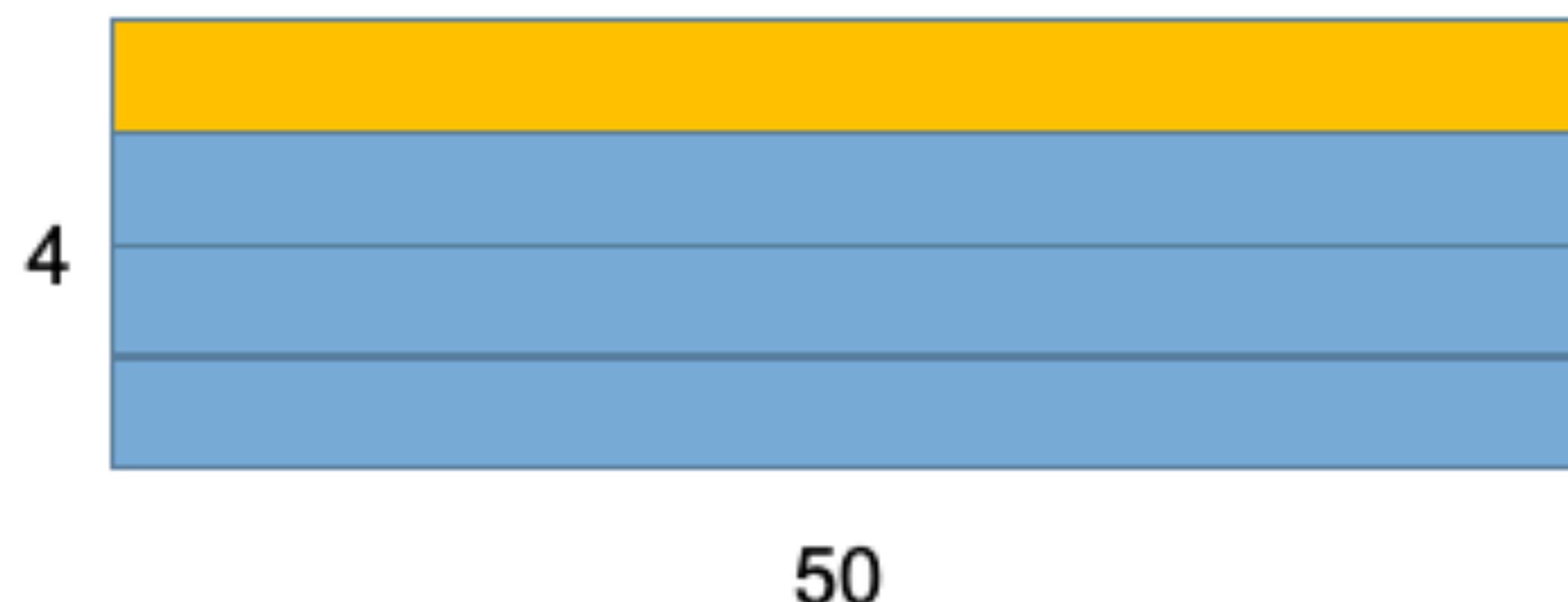
- Normalize the measurement outcome
  - Along the **batch** dimension
  - For example, we train the 4-qubit PQC with batch=50 then we have results as  $50 * 4$  values
    - Compute the mean and std on of the measurement outcome on each qubit across batch dim



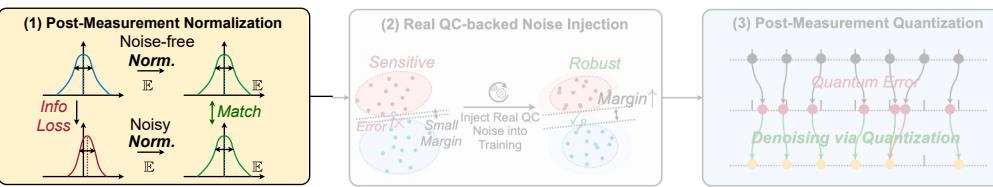
# Post-Measurement Normalization



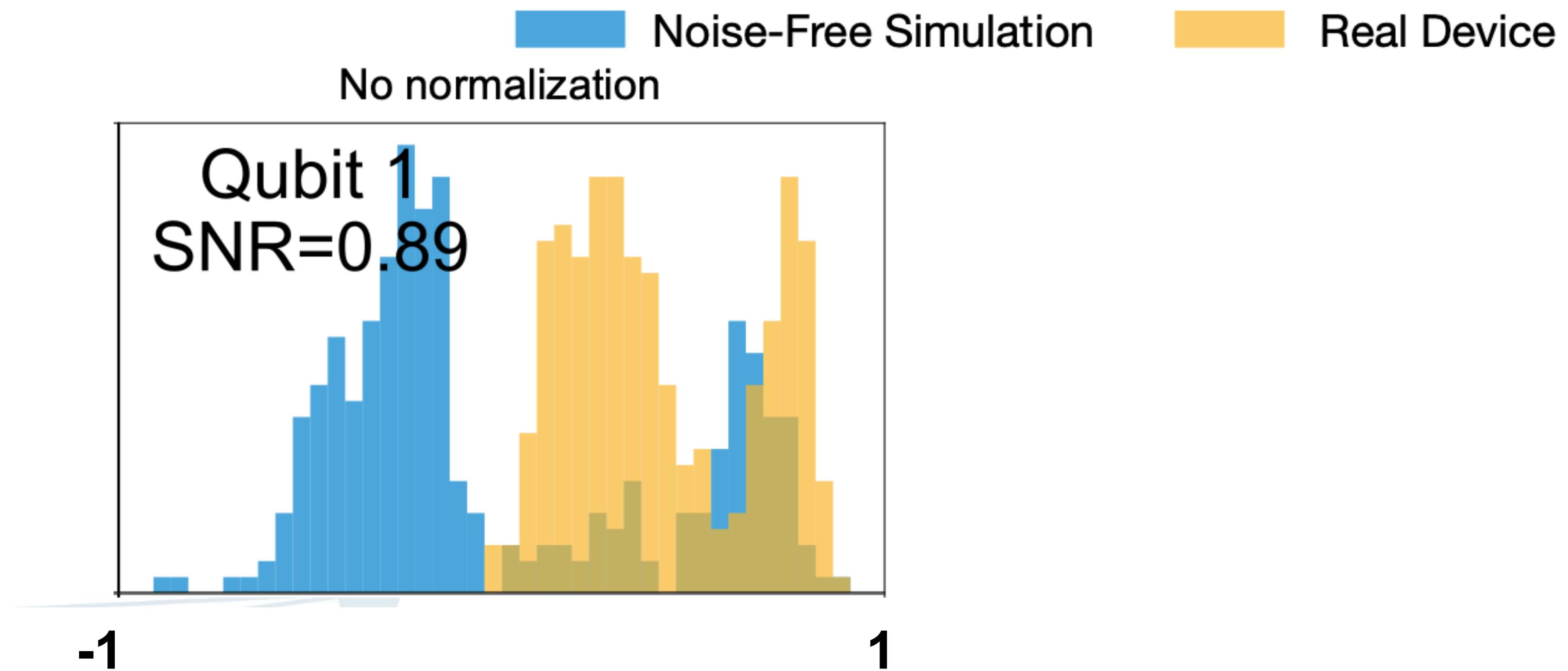
- Normalize the measurement outcome
  - Along the **batch** dimension
  - For example, we train the 4-qubit PQC with batch=50 then we have results as 50 \* 4 values
    - Compute the mean and std on of the measurement outcome on each qubit across batch dim
    - Normalize the measurement outcome with the computed mean and std



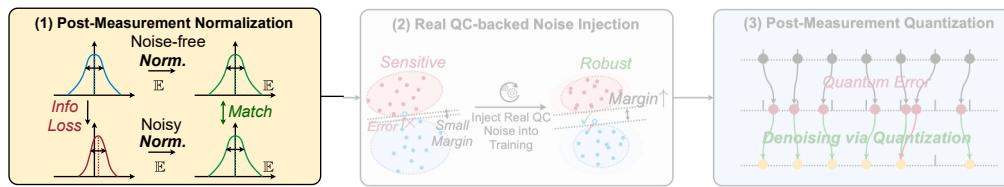
# Post-Measurement Normalization



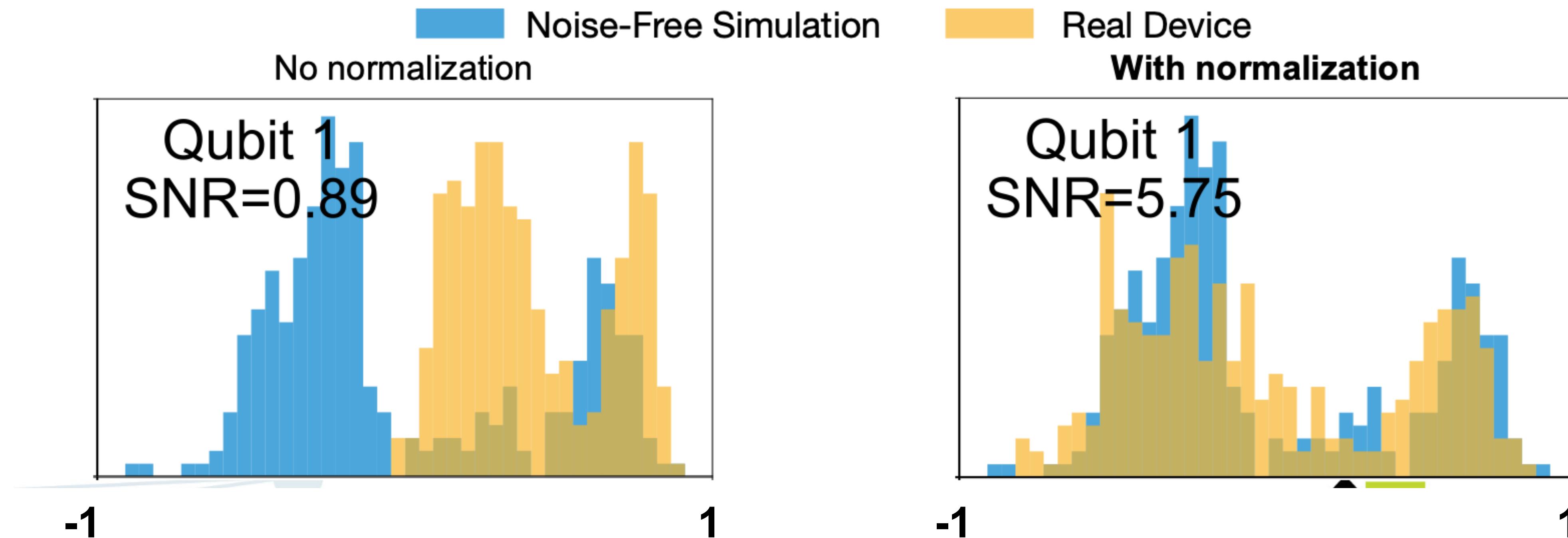
- Normalize the measurement outcome
  - Along the **batch** dimension
- Measurement outcome distribution of 50 quantum circuits:



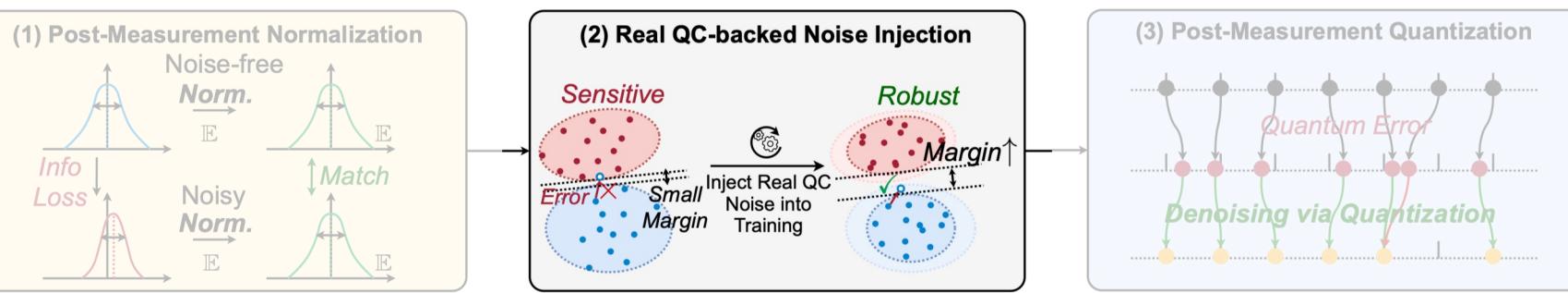
# Post-Measurement Normalization



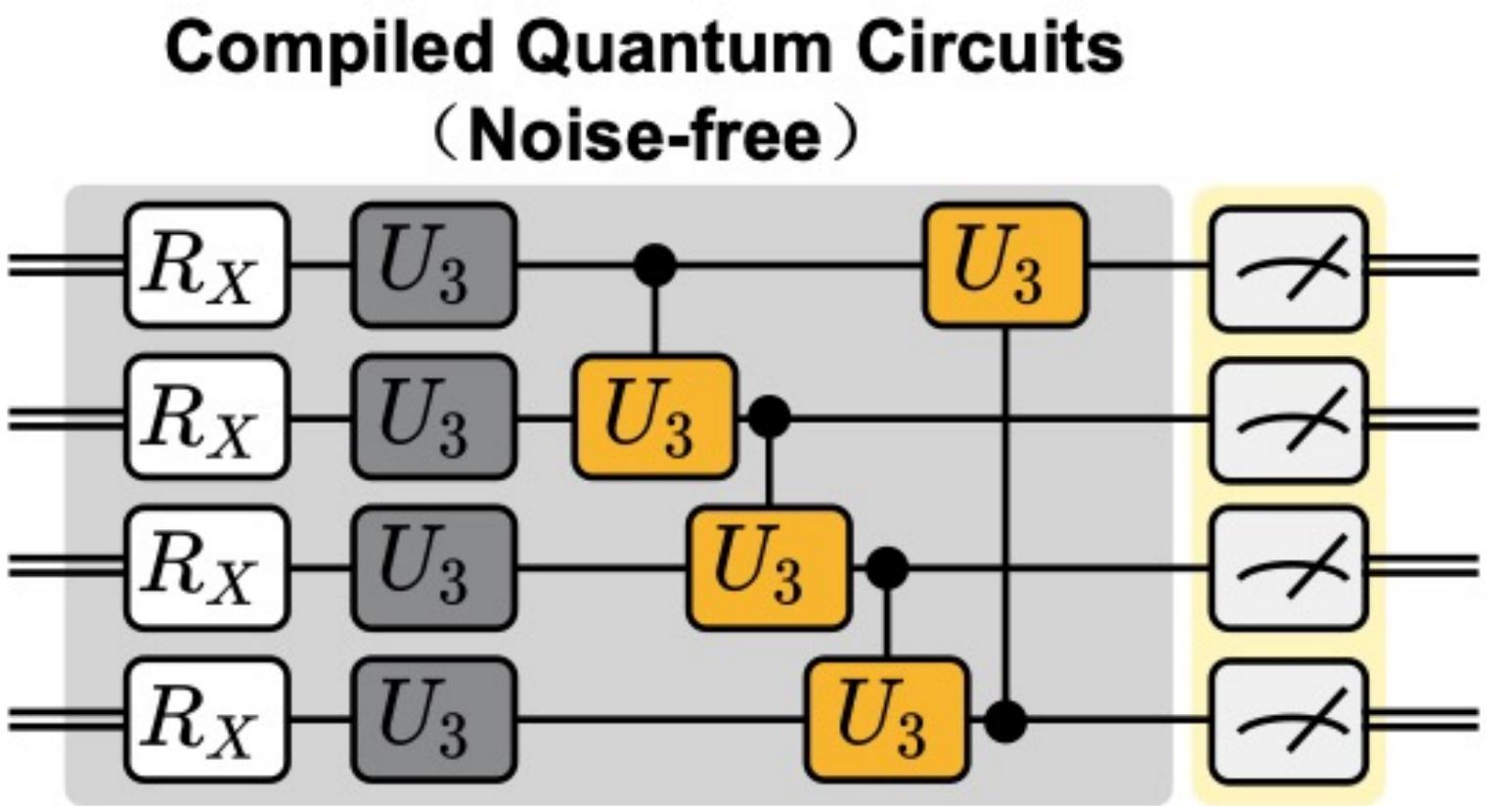
- Normalize the measurement outcome
  - Along the **batch** dimension
- Measurement outcome distribution of 50 quantum circuits:



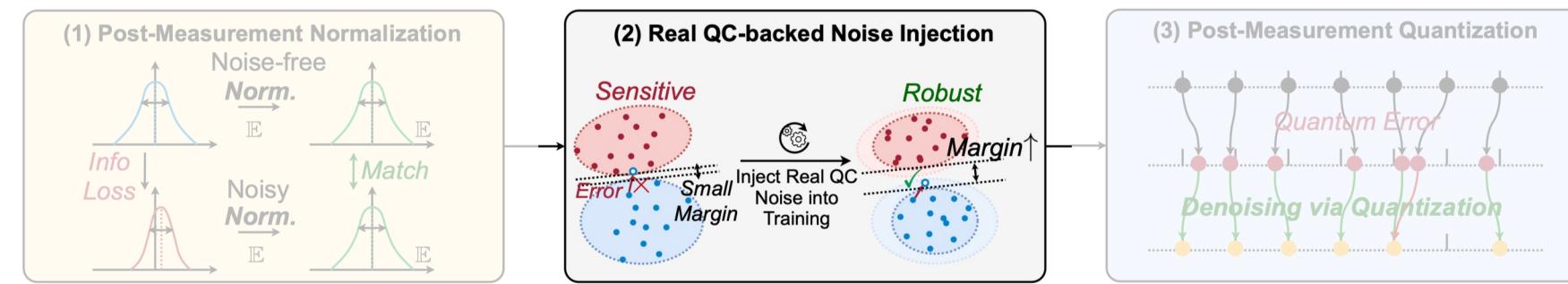
# Noise Injection



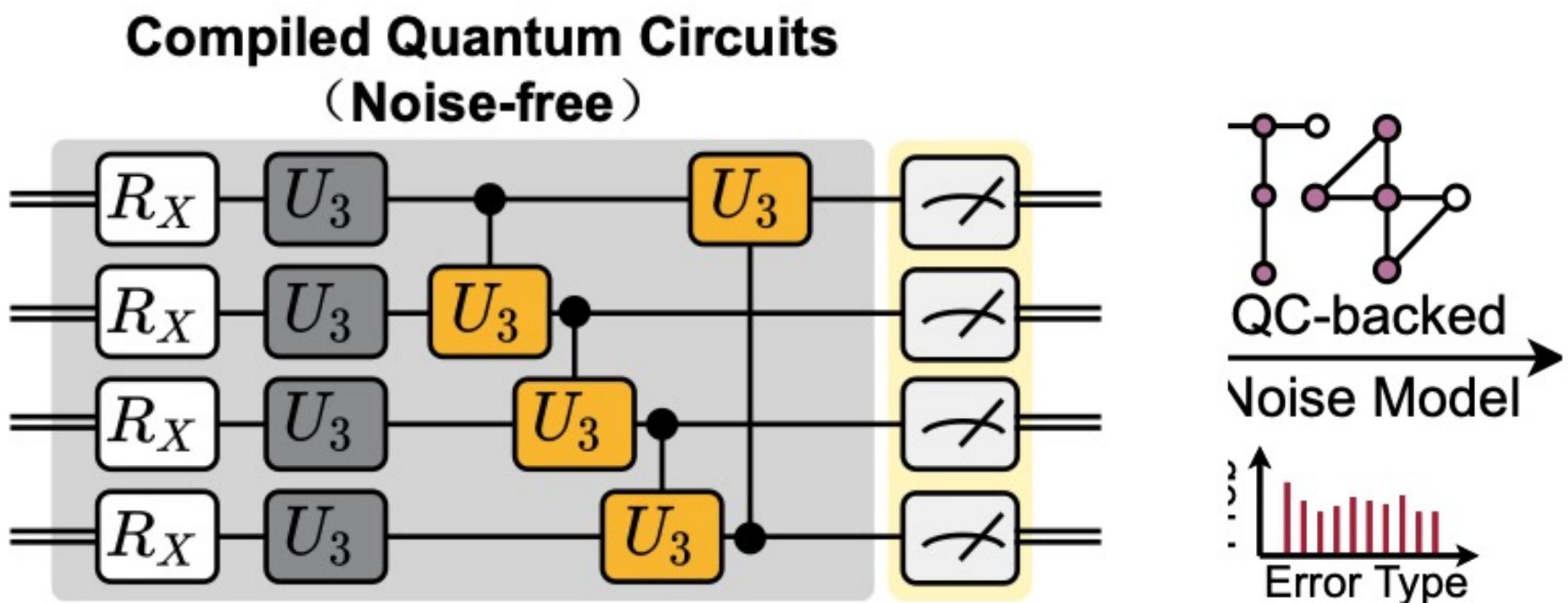
- Inject noise during training on classical simulator
  - Pauli error
  - Readout error



# Noise Injection



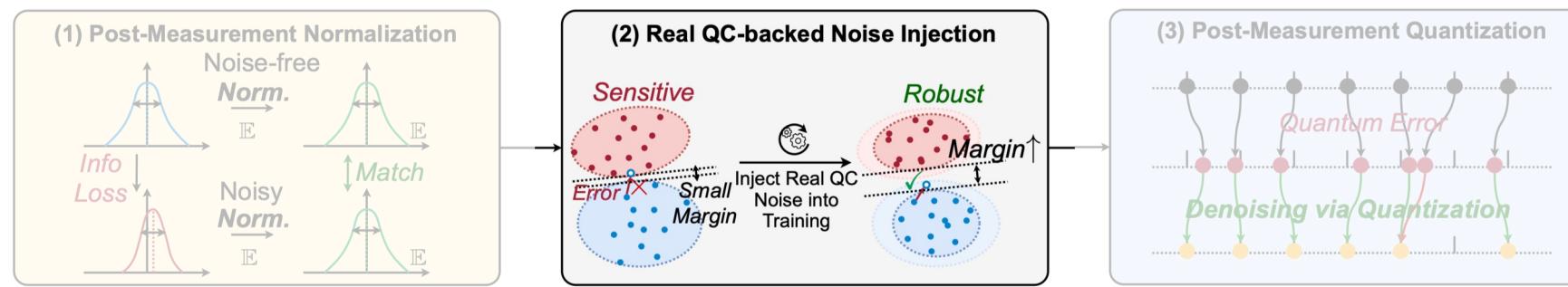
- Inject noise during training on classical simulator
  - Pauli error
  - Readout error



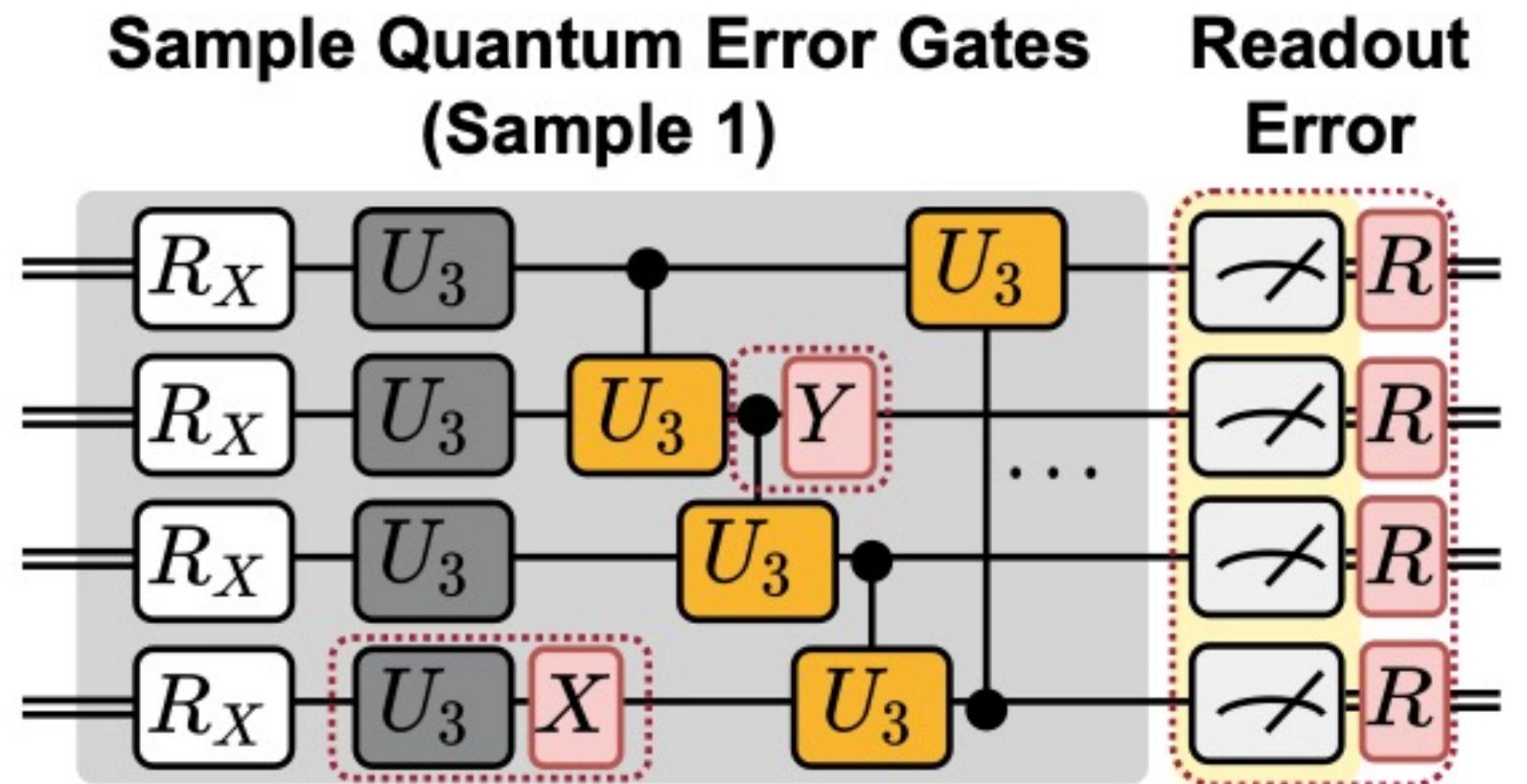
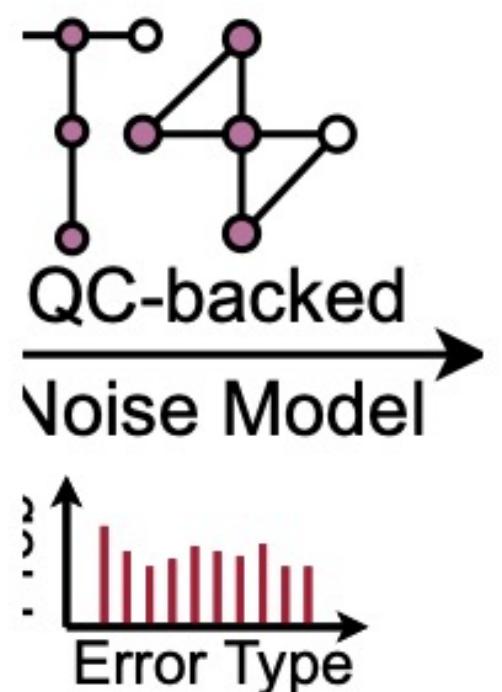
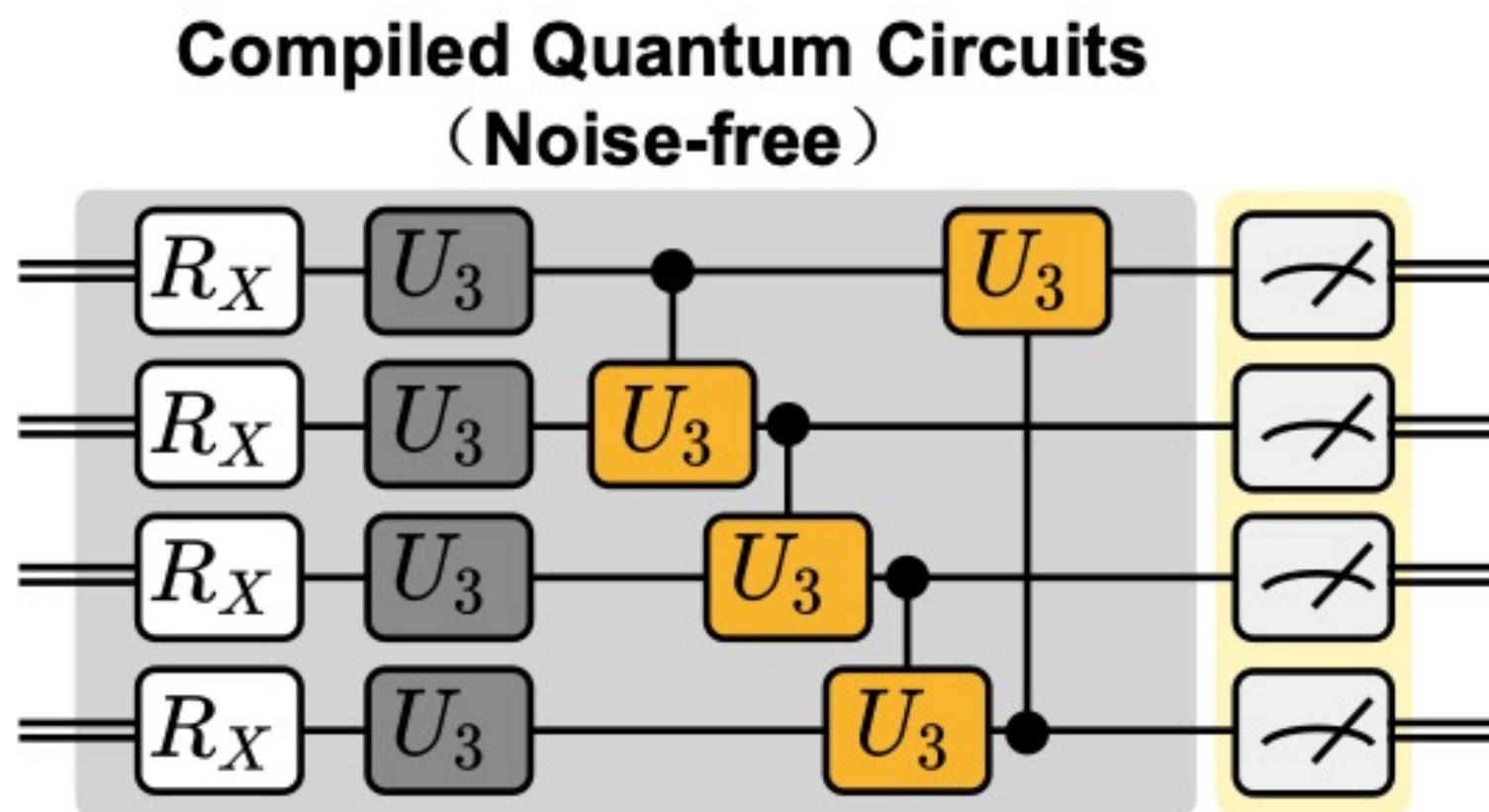
**Pauli error: SX gate: {X: 0.00096, Y: 0.00096, Z: 0.00096, None: 0.99712}**

**Readout Error Matrix:**  
0.984, 0.016  
0.022, 0.978

# Noise Injection



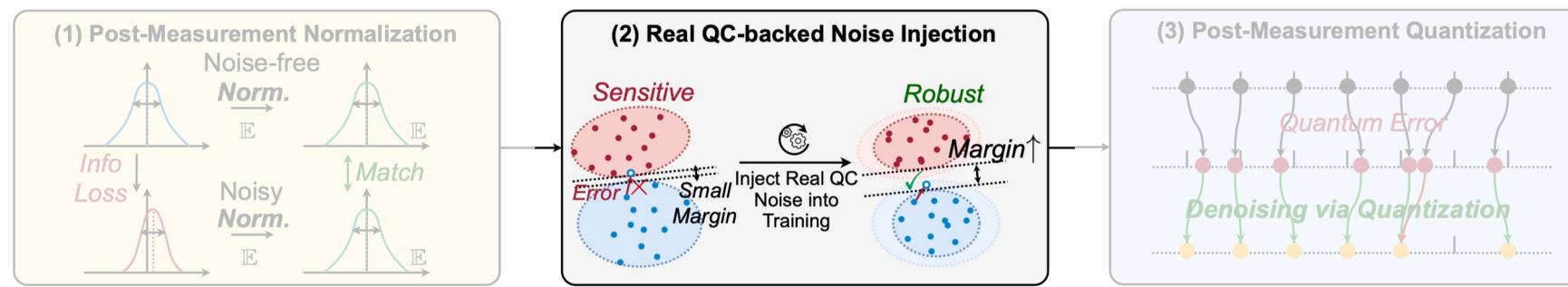
- Inject noise during training on classical simulator
  - Pauli error
  - Readout error



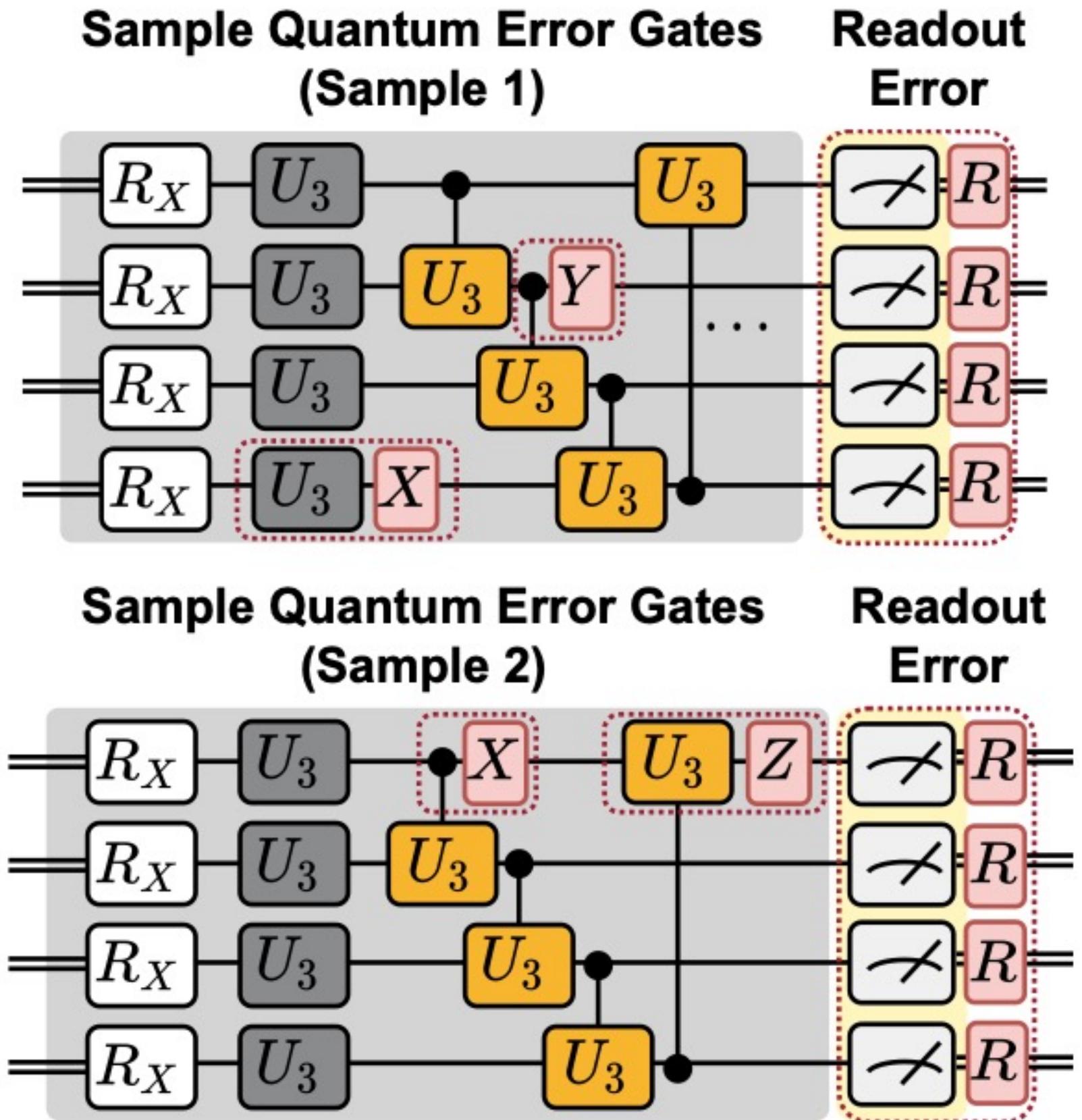
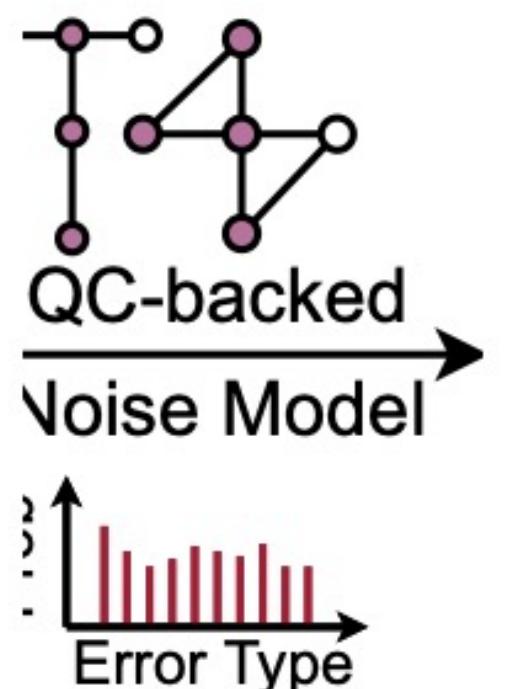
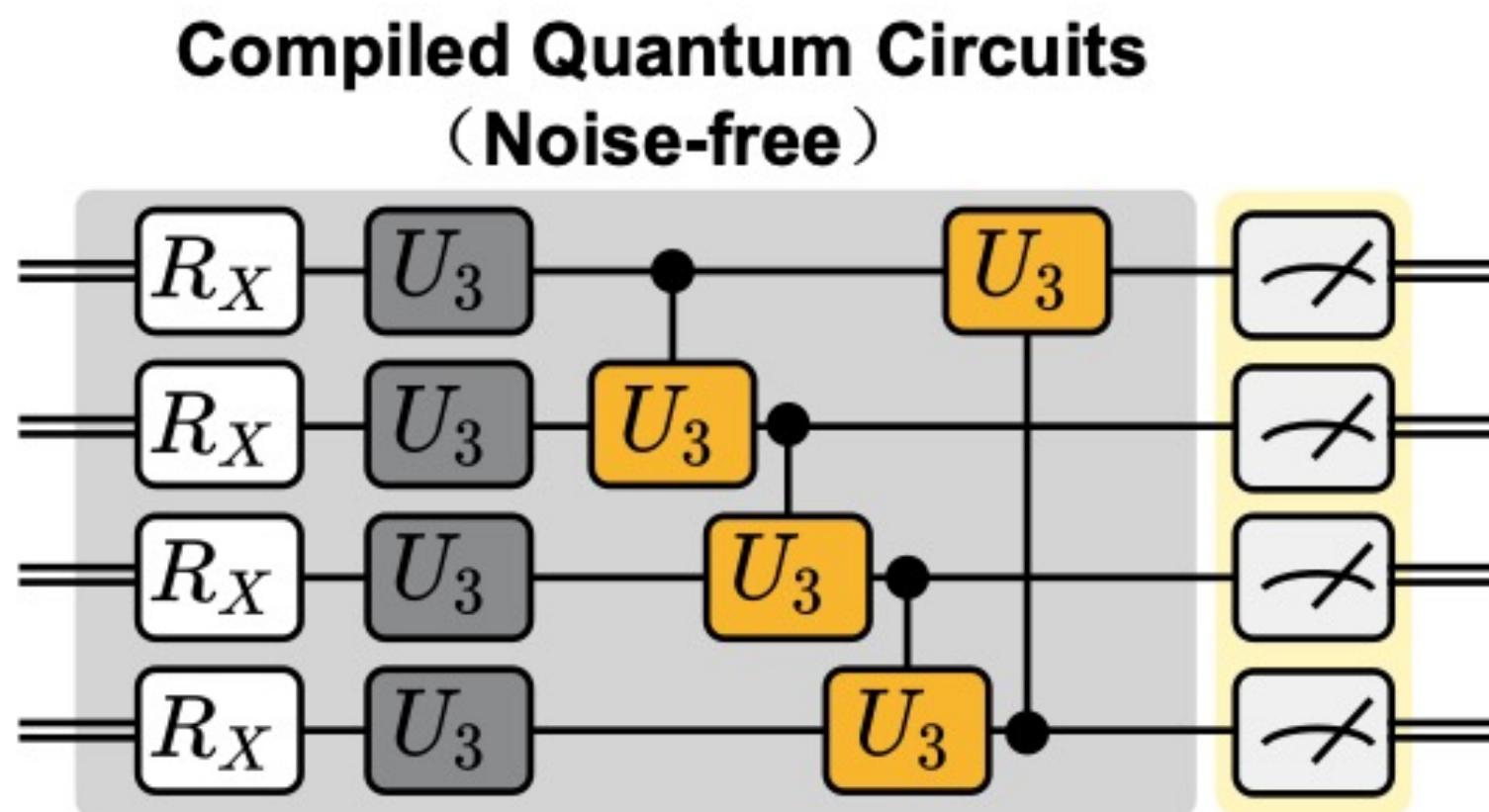
**Pauli error: SX gate: {X: 0.00096, Y: 0.00096, Z: 0.00096, None: 0.99712}**

**Readout Error Matrix:**  
0.984, 0.016  
0.022, 0.978

# Noise Injection



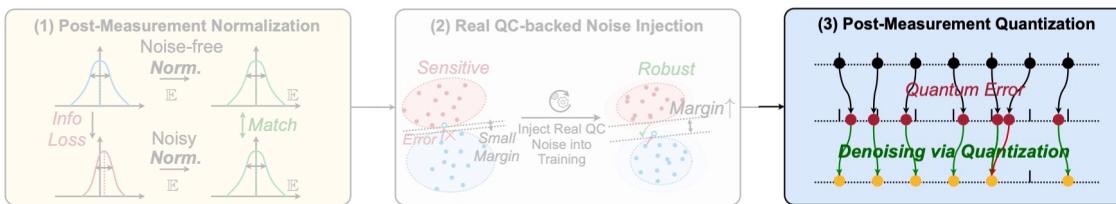
- Inject noise during training on classical simulator
  - Pauli error
  - Readout error



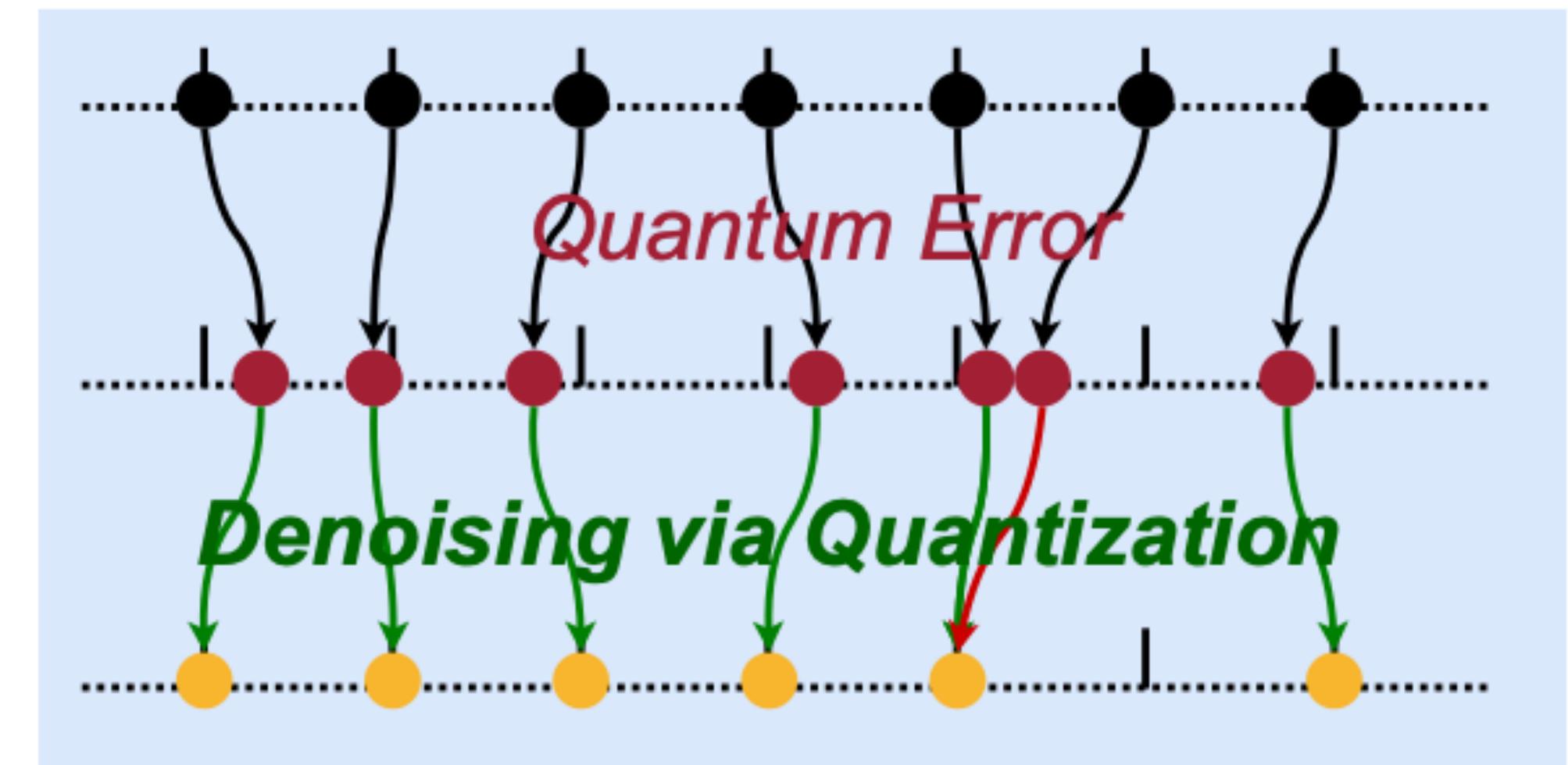
**Pauli error: SX gate: {X: 0.00096, Y: 0.00096, Z: 0.00096, None: 0.99712}**

**Readout Error Matrix:**  
0.984, 0.016  
0.022, 0.978

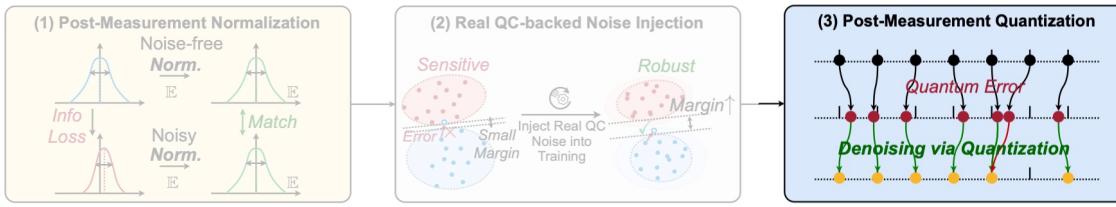
# Post-Measurement Quantization



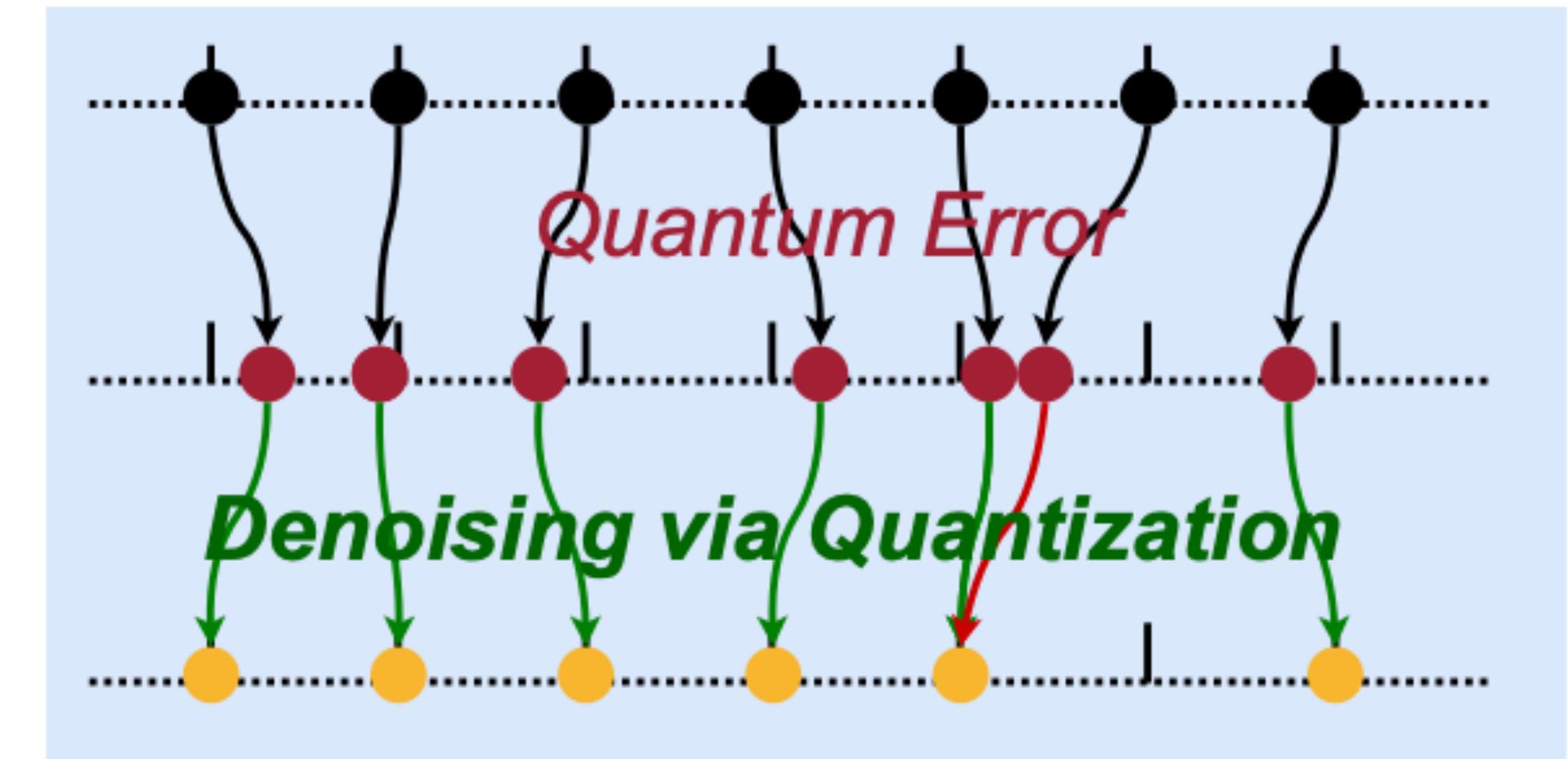
- Quantize measurement outcomes
  - Denoising effect
  - Small errors will be mitigated



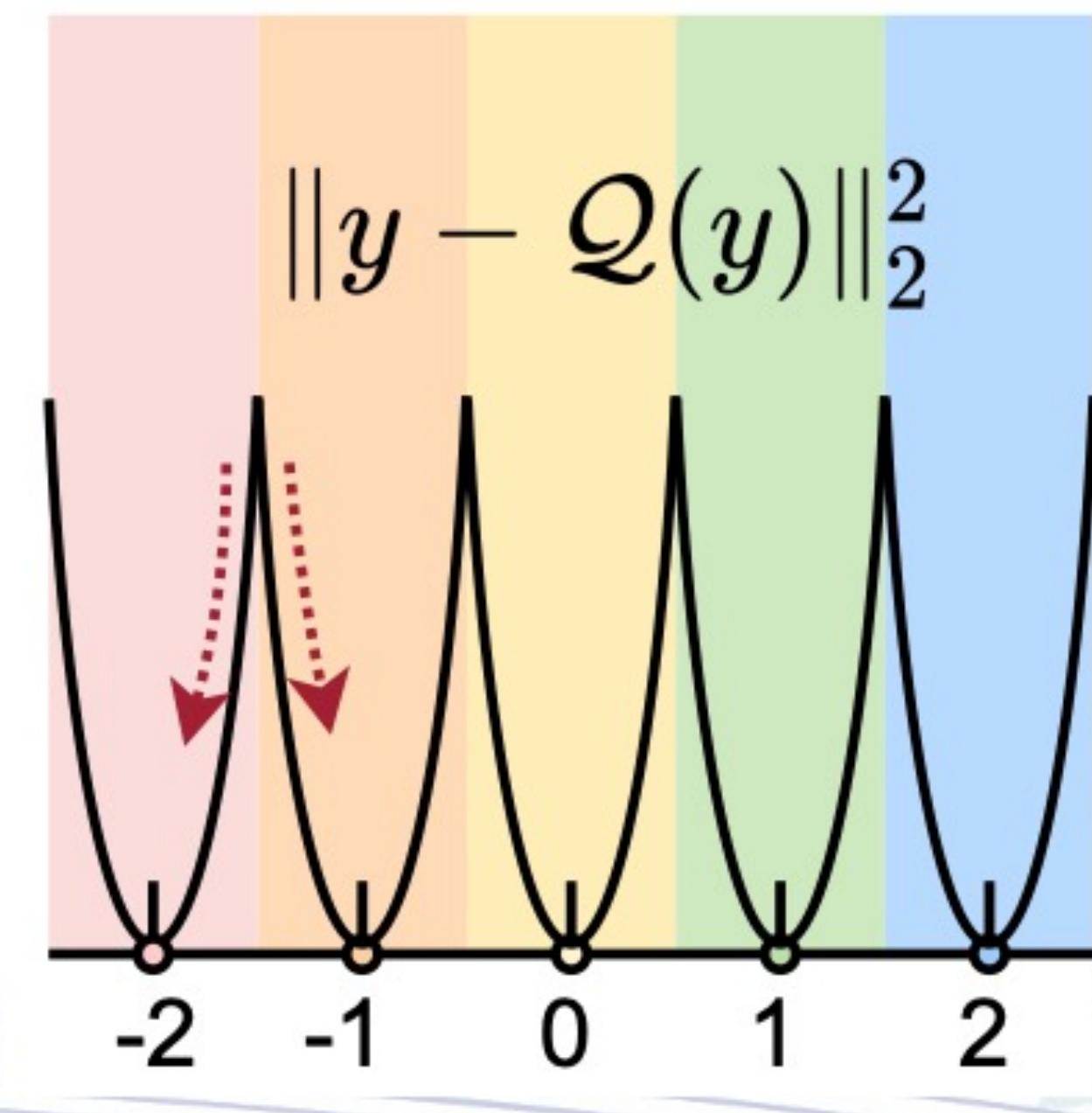
# Post-Measurement Quantization



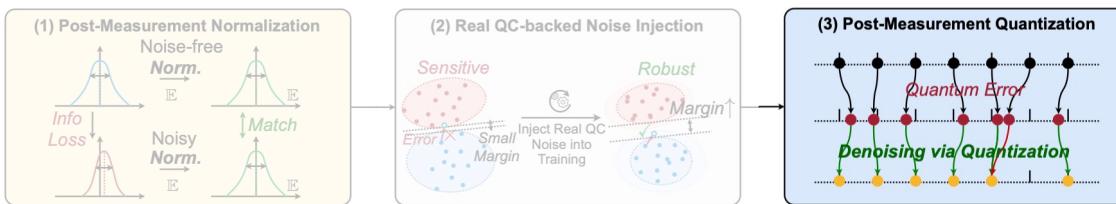
- Quantize measurement outcomes
  - Denoising effect
  - Small errors will be mitigated



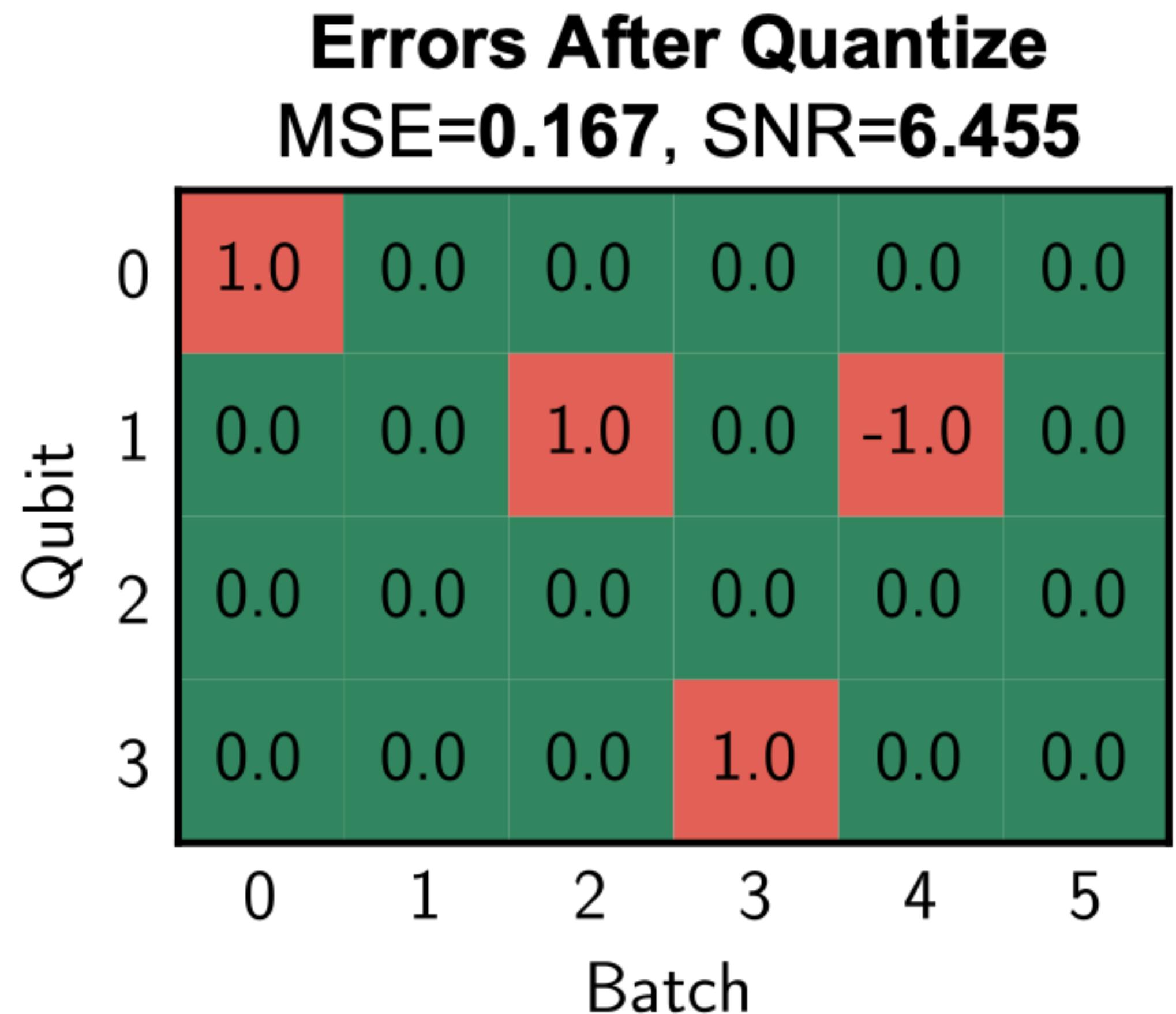
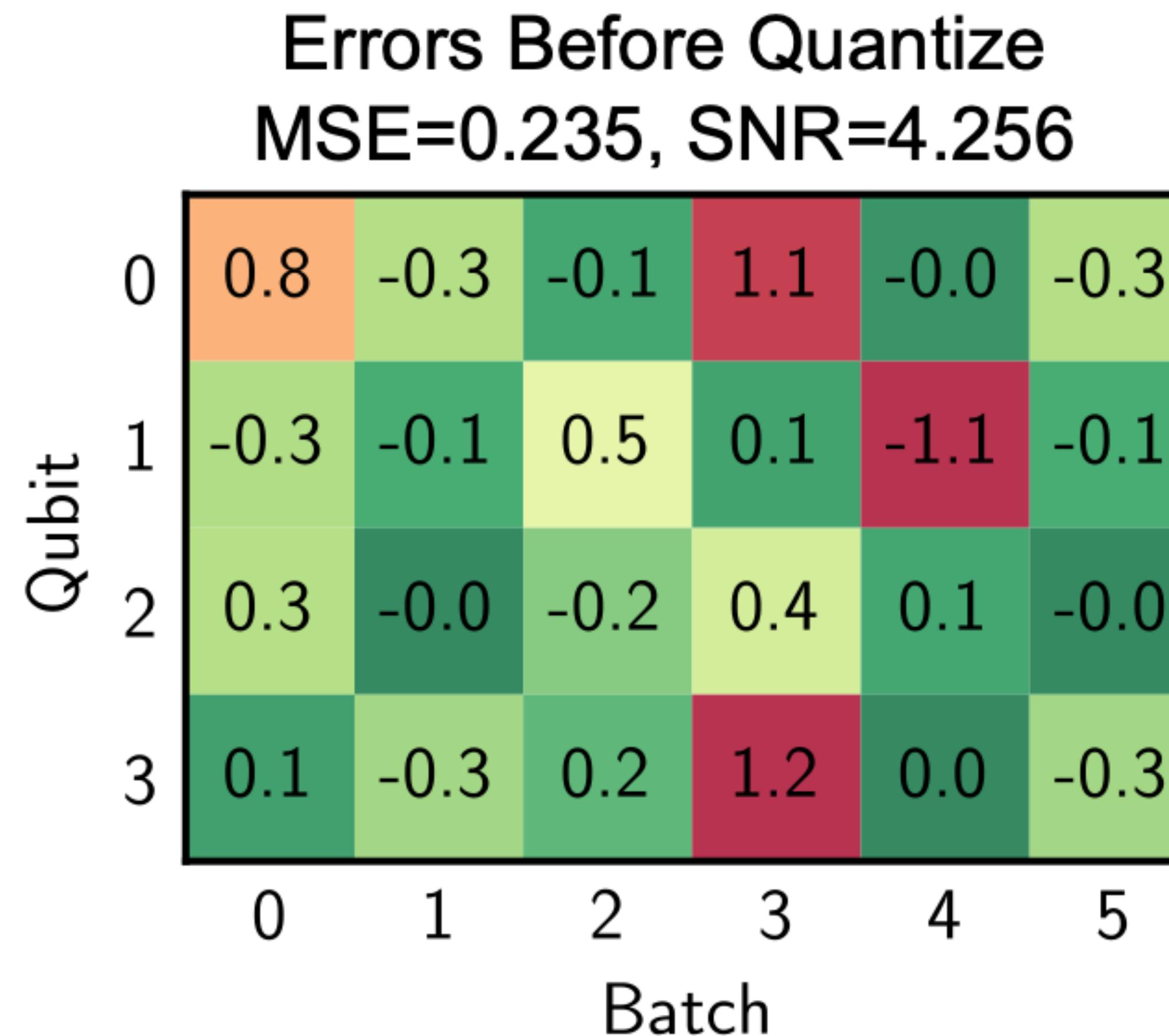
- **Loss** term to encourage measurement outcomes to be close to **centroids**



# Post-Measurement Quantization



- Quantization reduces errors and improves SNR



# Evaluation

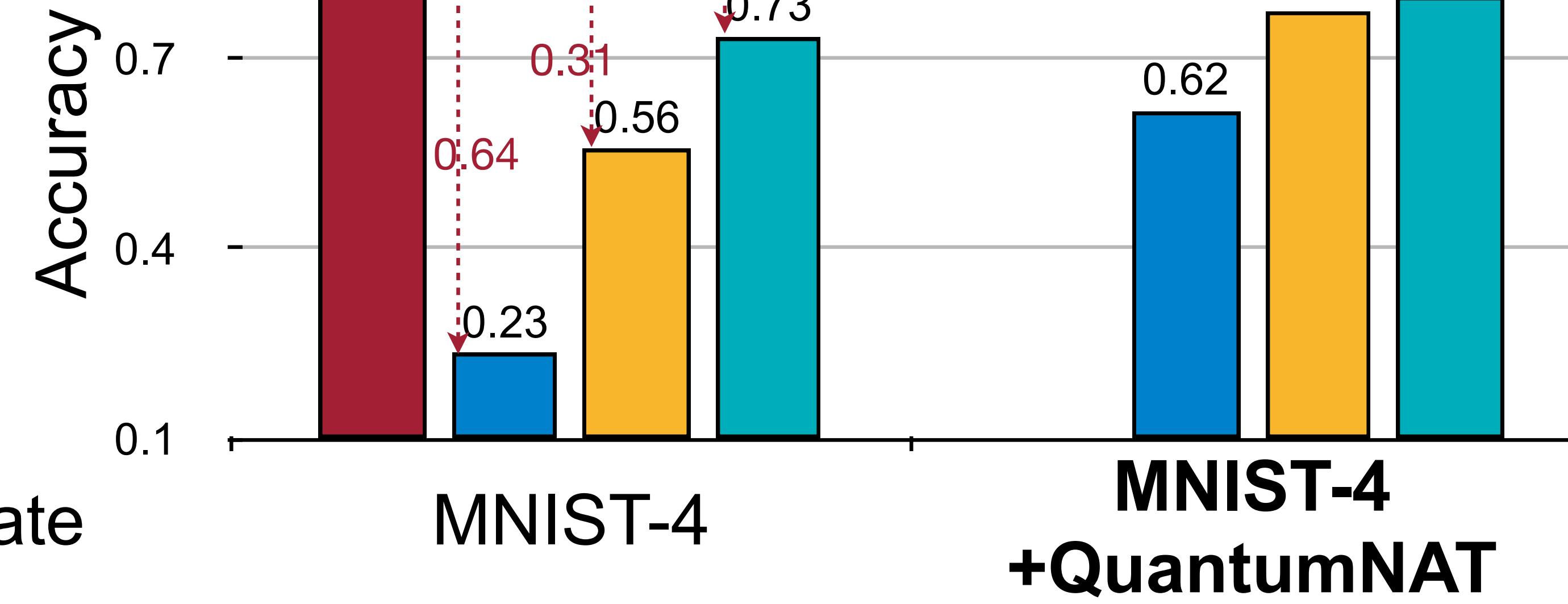
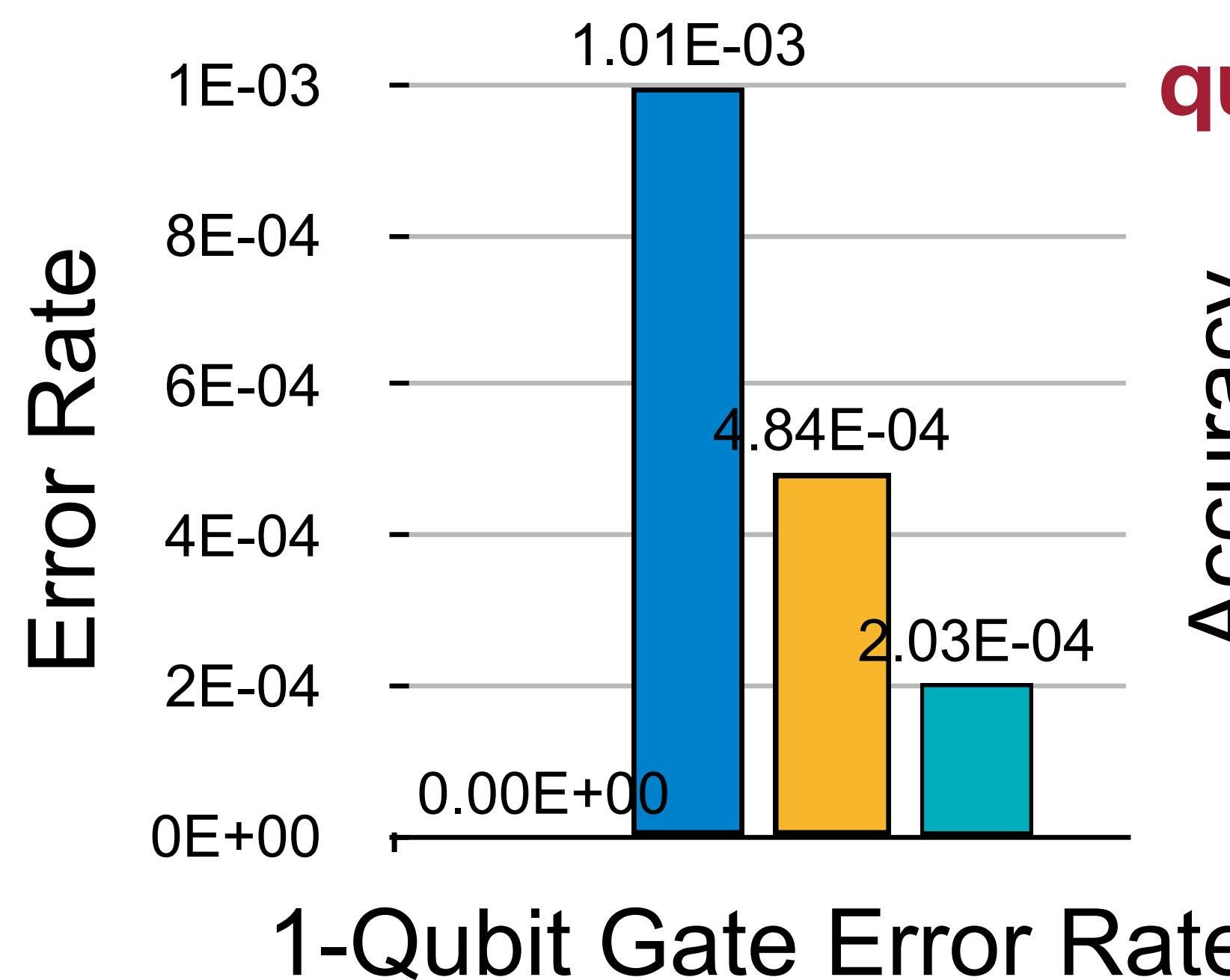
- Benchmarks
  - Quantum Machine Learning task:
    - MNIST 10-class, 4-class, 2-class
    - Fashion MNIST 10-class, 4-class, 2-class
    - Vowel 4-class
    - Cifar-2 class
  - Quantum Devices
    - IBMQ
    - #Qubits: 5 to 15
    - Quantum Volume: 8 to 32

# Evaluation

- QuantumNAT significantly improves real measurement accuracy

Noise-Free Simulation    IBMQ-Yorktown    Lima    Santiago

Severe accuracy drop because of  
quantum errors on real devices



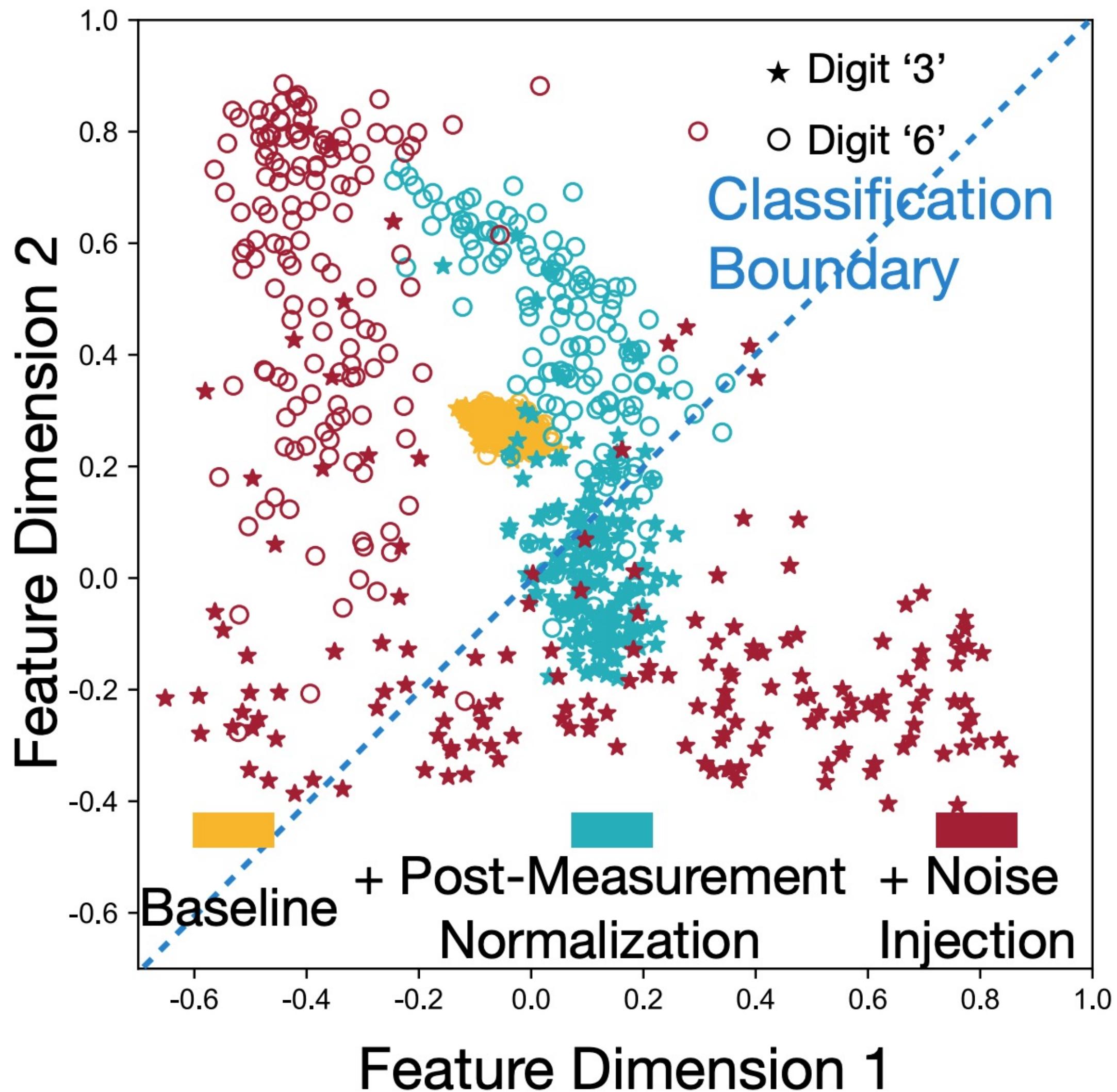
# Consistent Improvements on Various Benchmarks

- On IBMQ santiago

Method	MNIST-4	FMNIST-4	Vowel-4	MNIST-2	FMNIST-2	Cifar-2
Baseline	0.30	0.32	0.28	0.84	0.78	0.51
+ Normalization	0.41	0.61	0.29	0.87	0.68	0.56
+ Noise Injection	0.61	0.70	0.44	0.93	0.86	0.57
+ Quantization	<b>0.68</b>	<b>0.75</b>	<b>0.48</b>	<b>0.94</b>	<b>0.88</b>	<b>0.59</b>

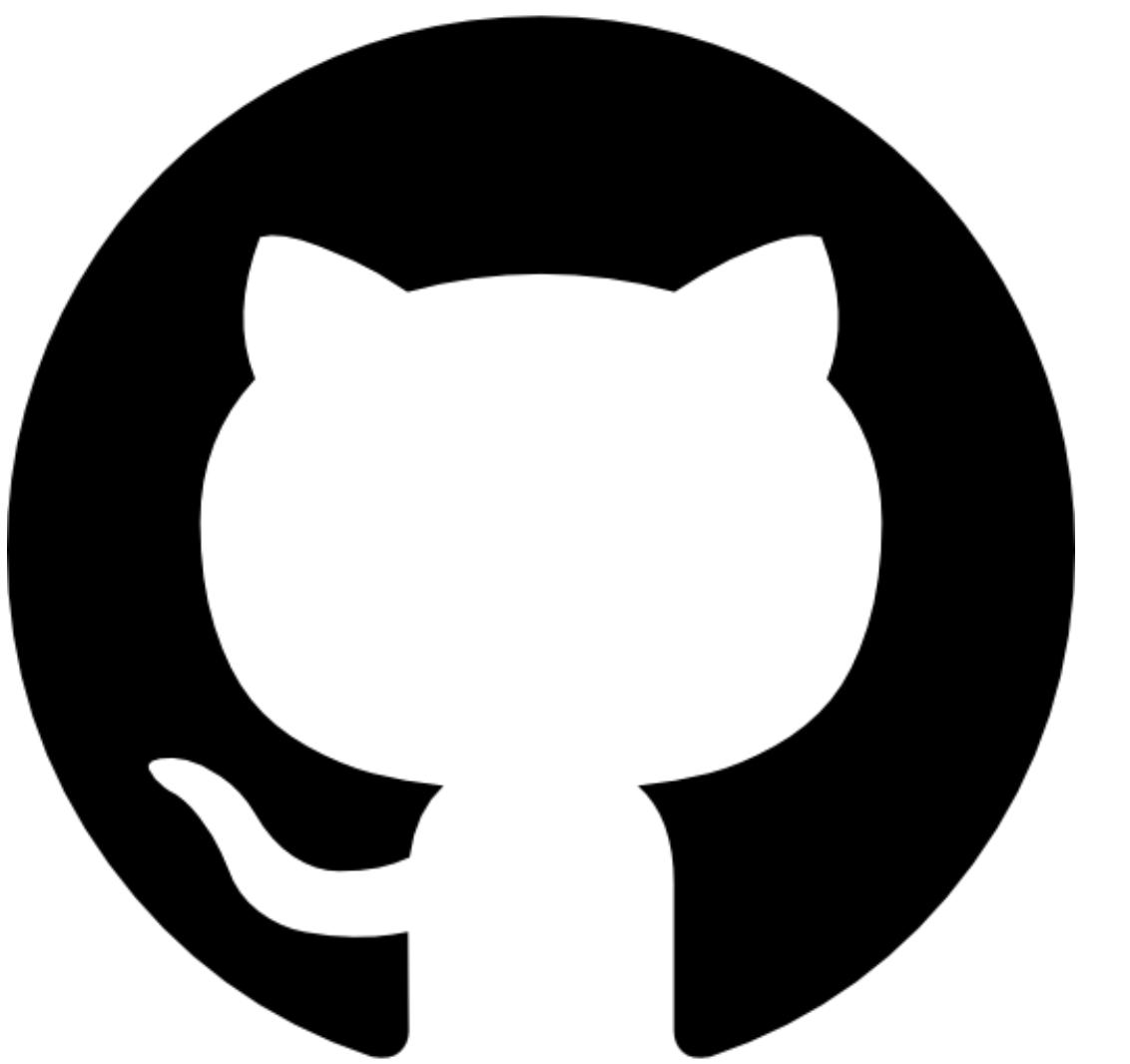
# Visualization

- QuantumNAT stretches the distribution of features
  - MNIST-2 classification task



# Hands-On Section

## 2.1 QuantumNAT



# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations

1.3 TQ for State Prep

1.4 TQ for VQE

1.4 TQ for QNN

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning

2.2 QuantumNAT: Noise Injection and Quantization

**2.3 QOC: On-Chip Training**

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression

## Section 3

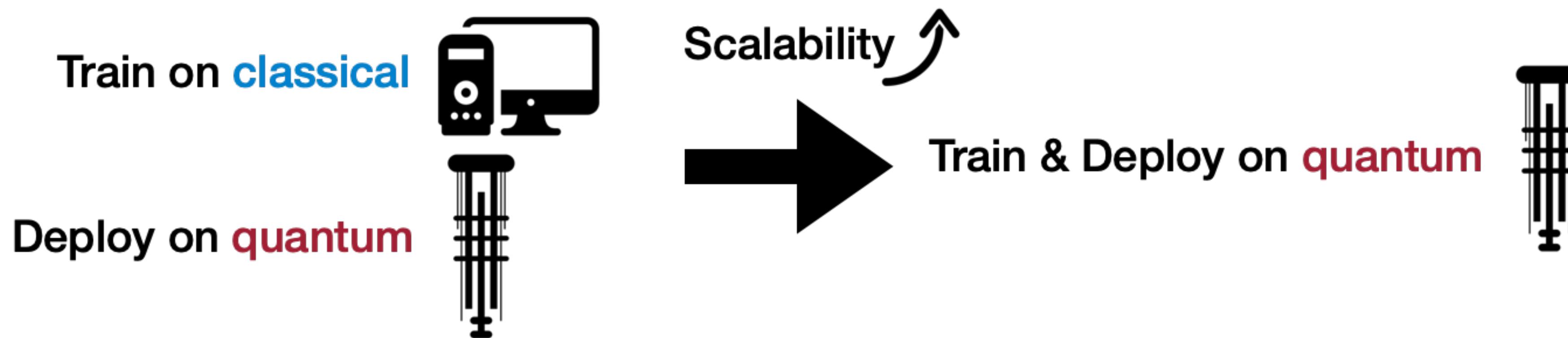
### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control

3.2 Variational Pulse Learning

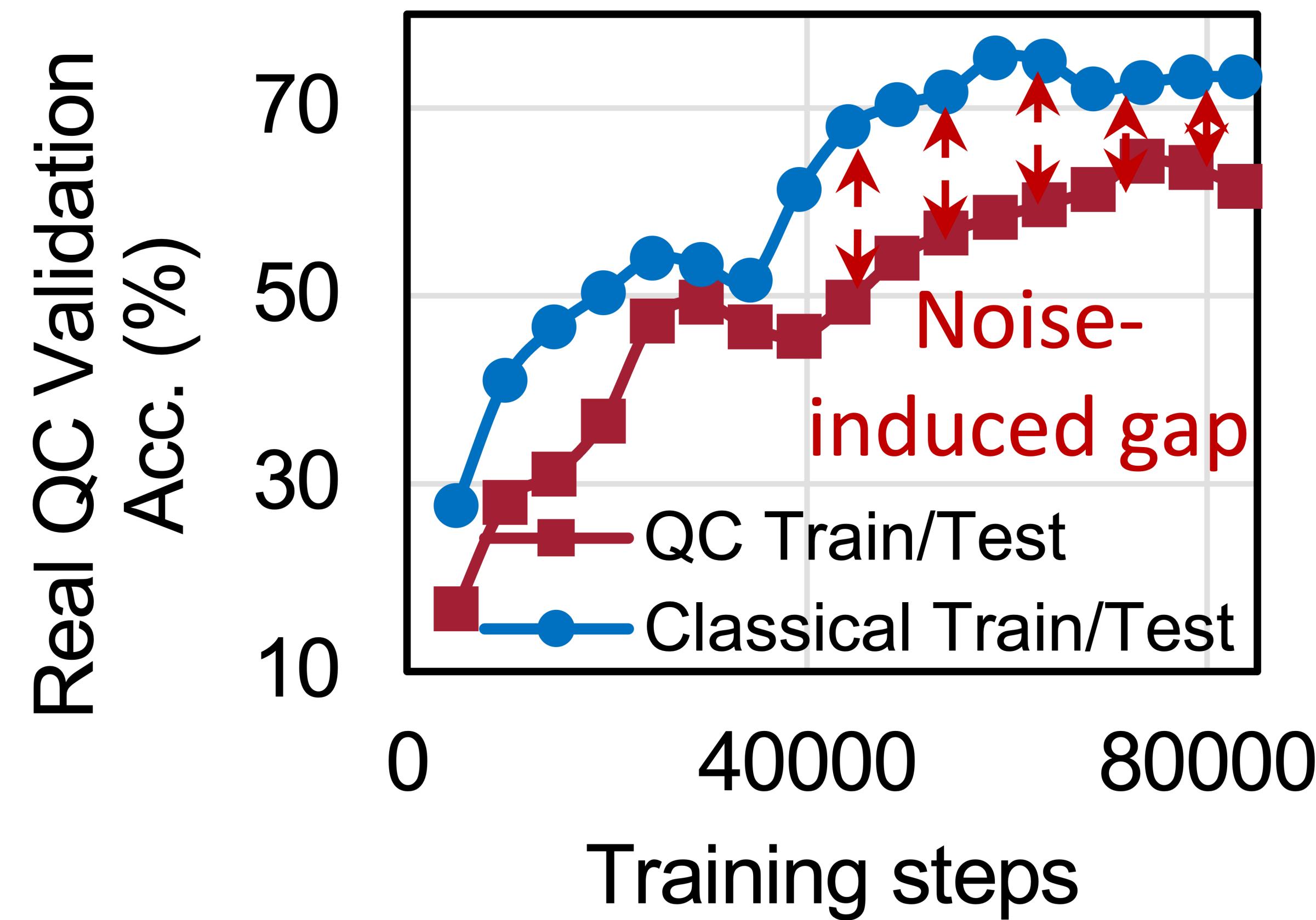
# QOC for High Scalability

- How to further improve the scalability of PQC training?
- Train the parameters directly on real quantum machine



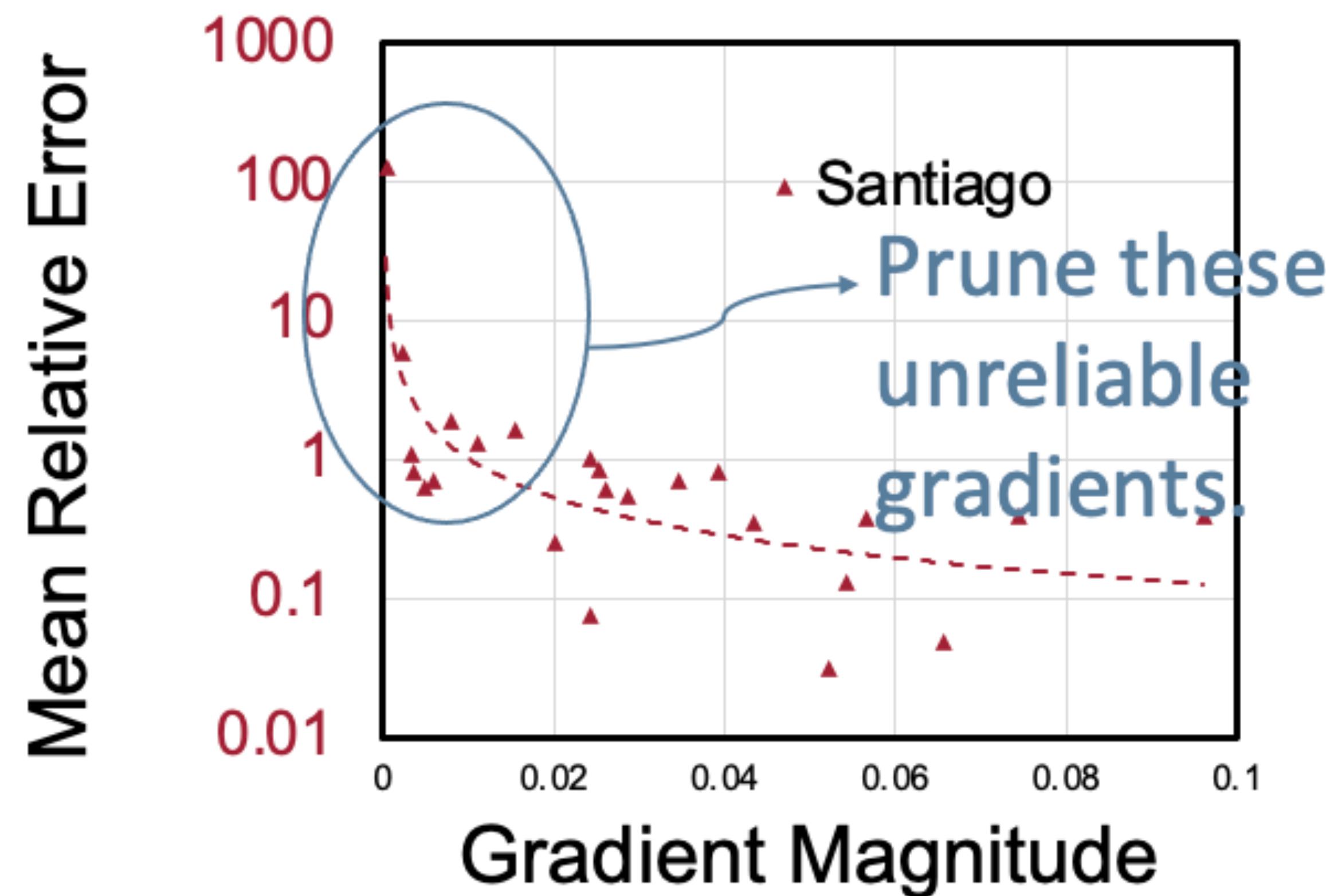
# Challenge of On-chip Training: noise

- Noise **reduces reliability** of on-chip computed gradients



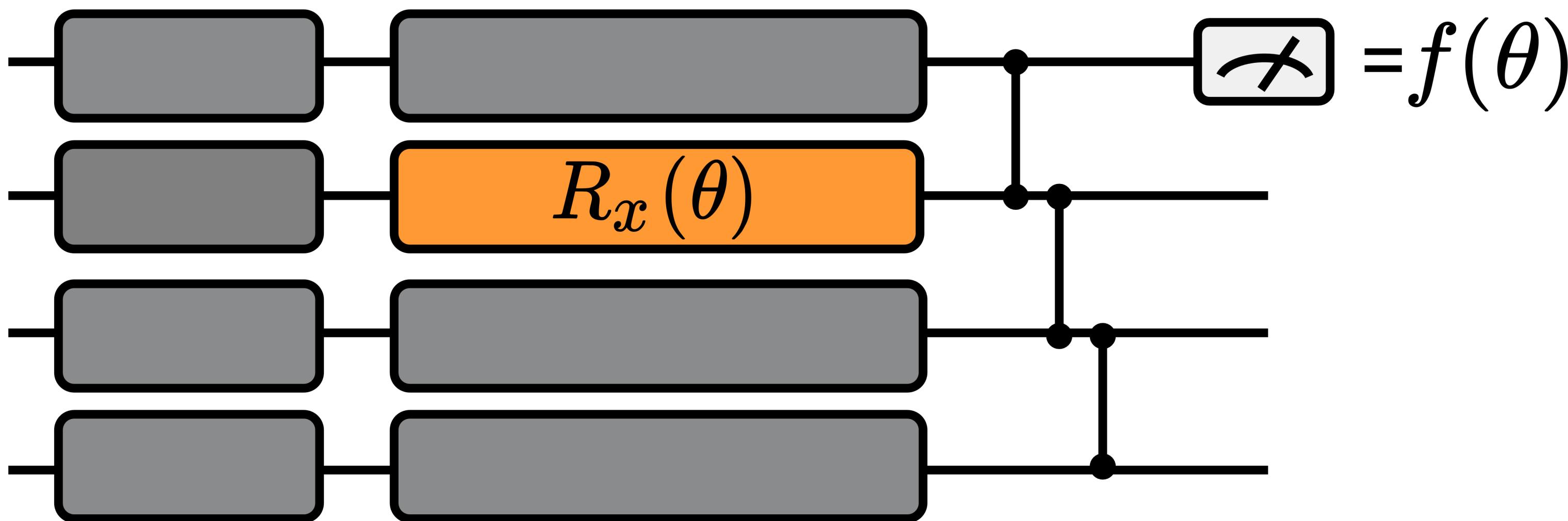
# Challenge of On-chip Training: noise

- Noise **reduces reliability** of on-chip computed gradients
- **Small** magnitude gradients have **large** relative errors



# Parameter Shift Rules

- Calculate the gradient of  $\theta$  w.r.t.  $f(\theta)$ .

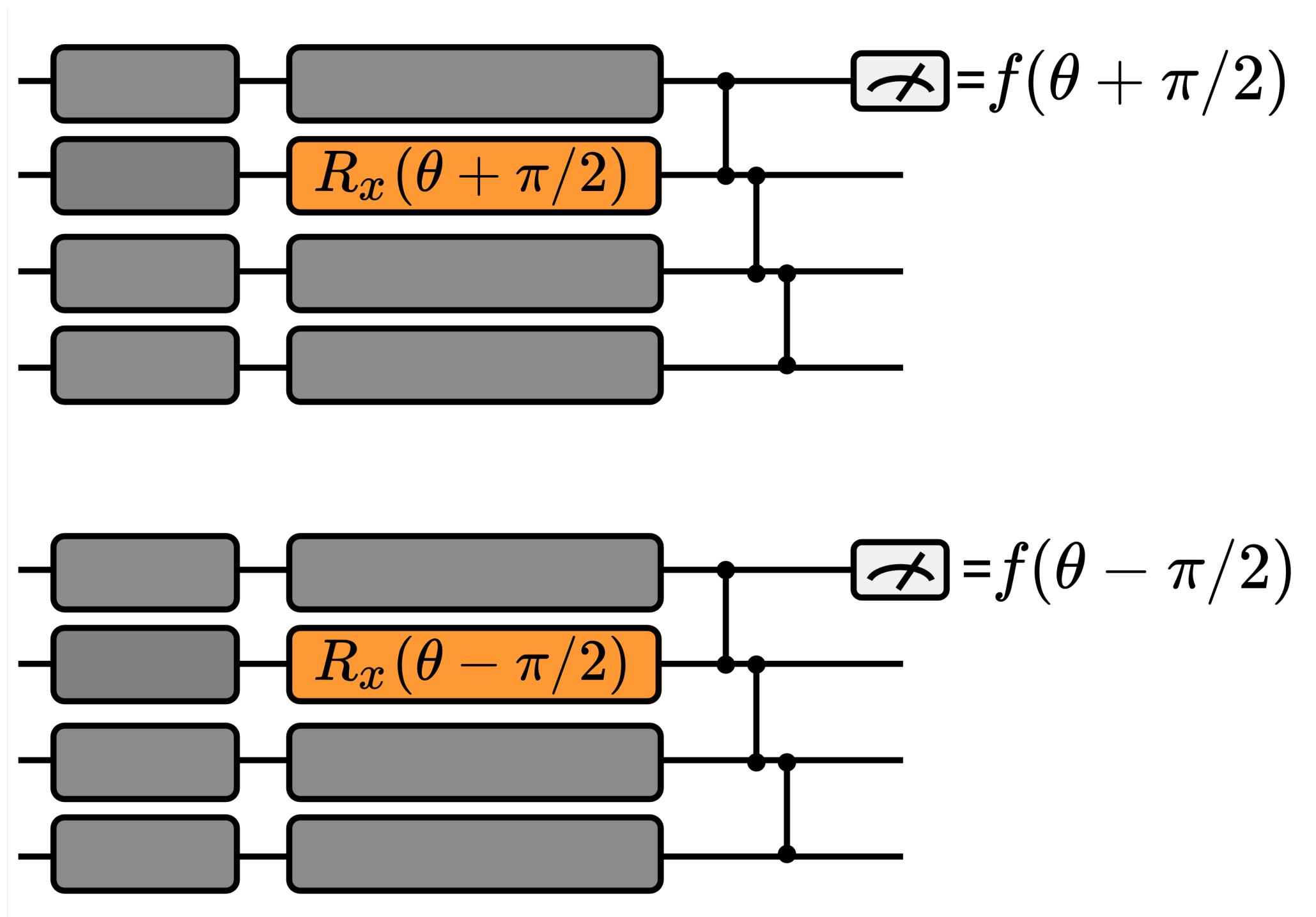


# Parameter Shift Rules

- Calculate the gradient of  $\theta$  w.r.t.  $f(\theta)$ .

# Parameter Shift Rules

- Shift  $\theta$  twice



$$\frac{\partial}{\partial \theta} f(\theta) = \frac{1}{2} \left( f\left(\theta + \frac{\pi}{2}\right) - f\left(\theta - \frac{\pi}{2}\right) \right)$$

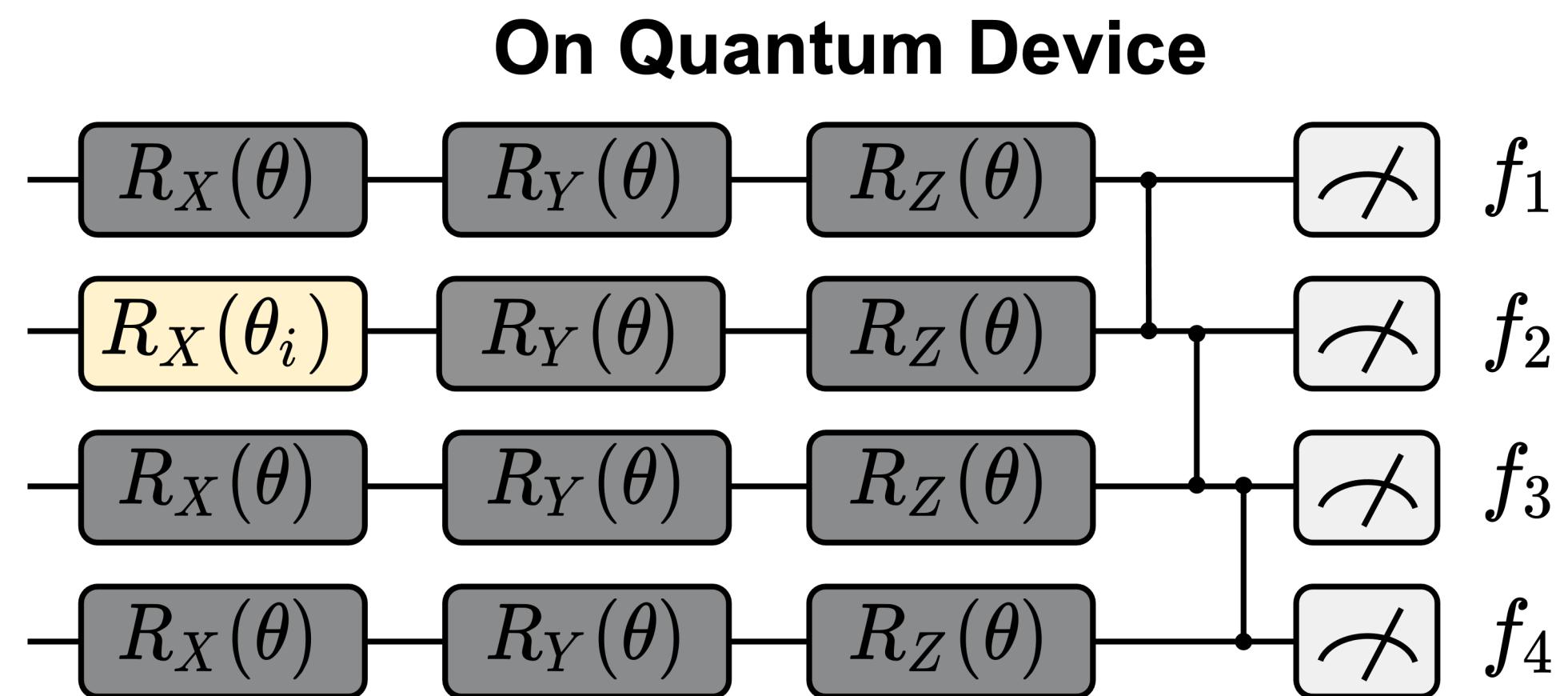
# Parameter Shift Rules

- This formula is valid to all rotation gates
  - RZ, RY, RX, RXX, RZZ
- One gradient requires two runs on real quantum machine

$$\frac{\partial}{\partial \theta} f(\theta) = \frac{1}{2} \left( f\left(\theta + \frac{\pi}{2}\right) - f\left(\theta - \frac{\pi}{2}\right) \right)$$

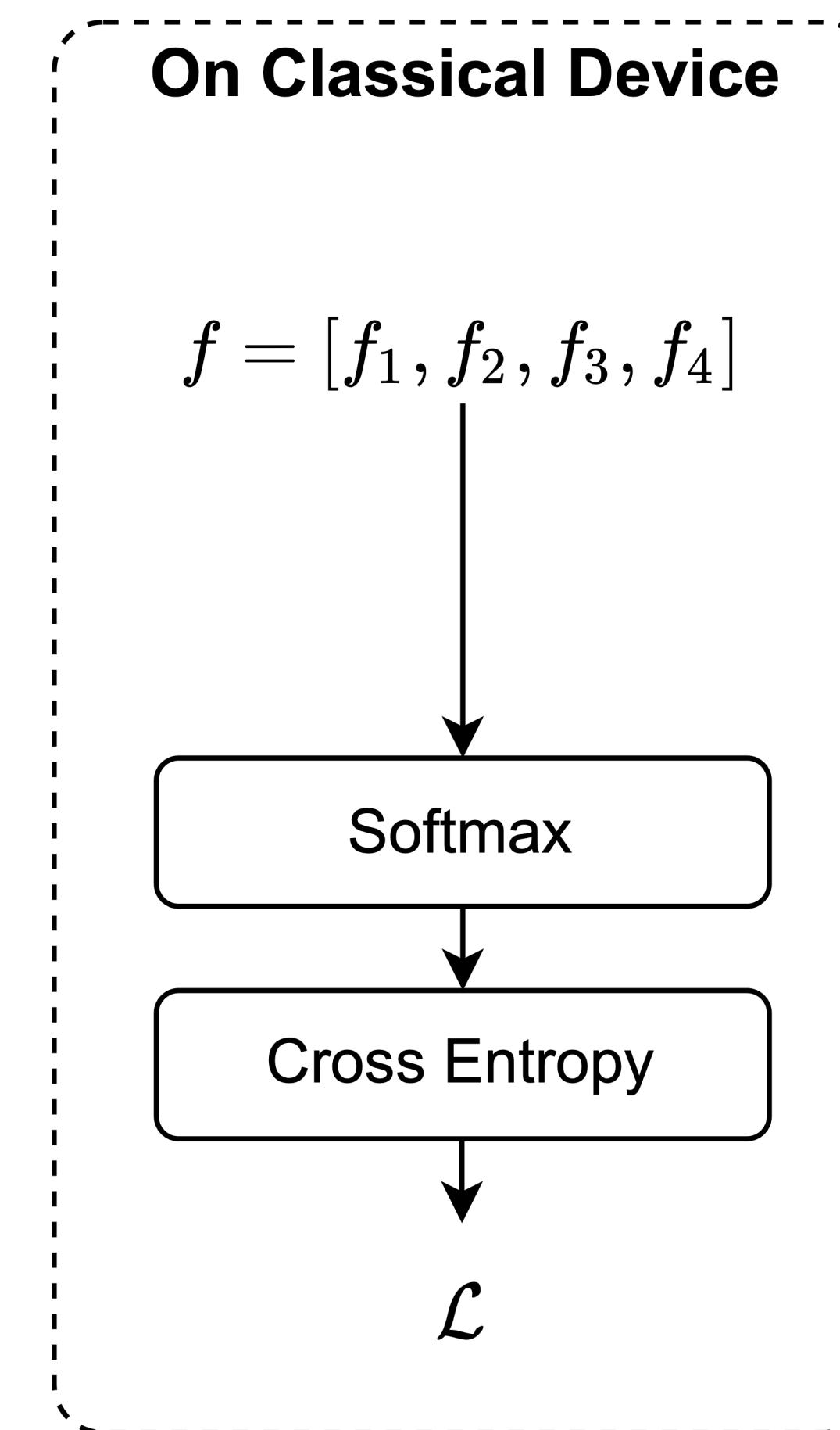
# Calculate Gradients of PQC

- Step 1: Run on QC without shift to obtain  $f$



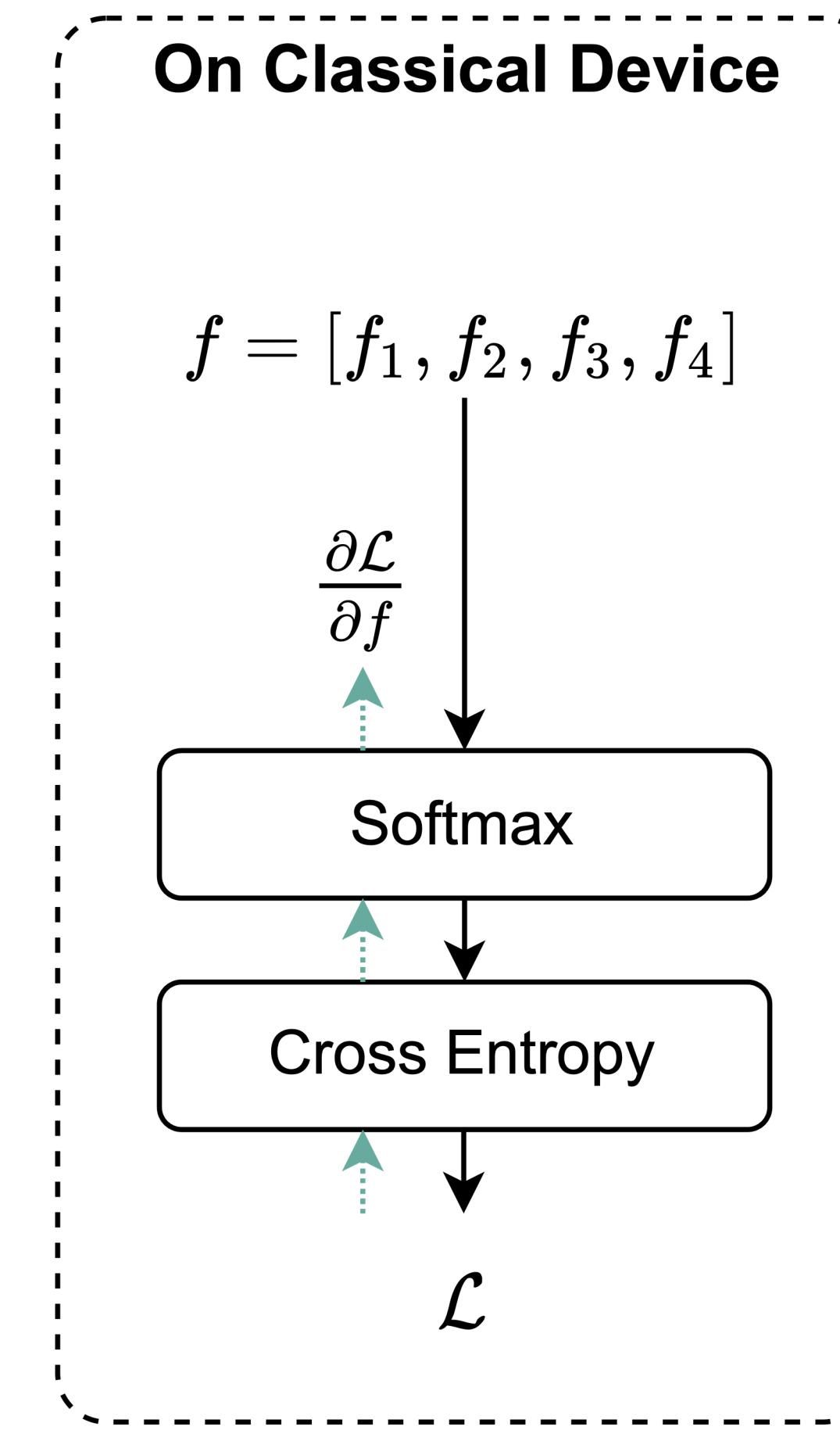
# Calculate Gradients of PQC

- Step 2: Further forward to get *Loss*



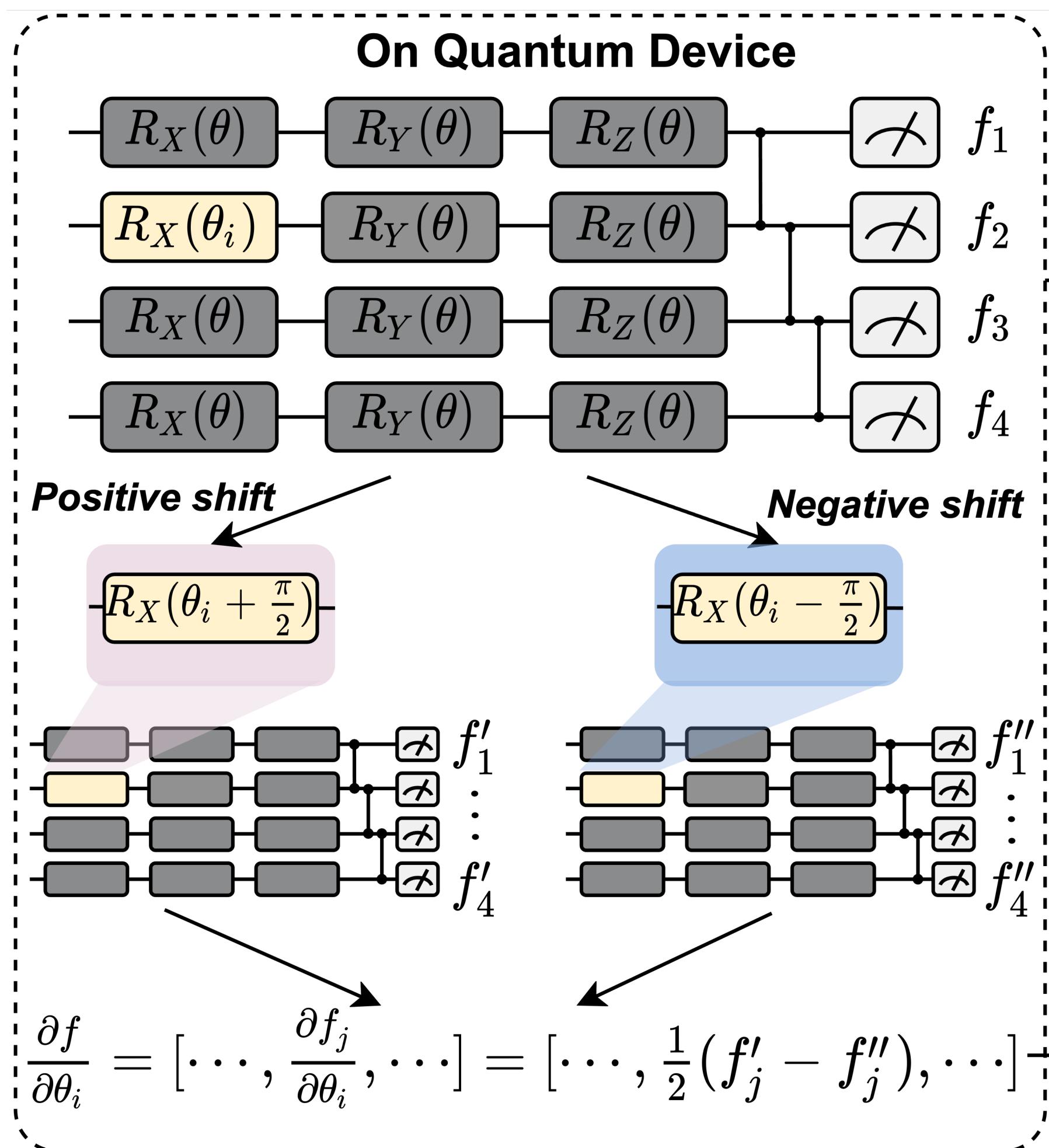
# Calculate Gradients of PQC

- Step 3: Backpropagation to calculate  $\frac{\partial Loss}{\partial f(\theta)}$



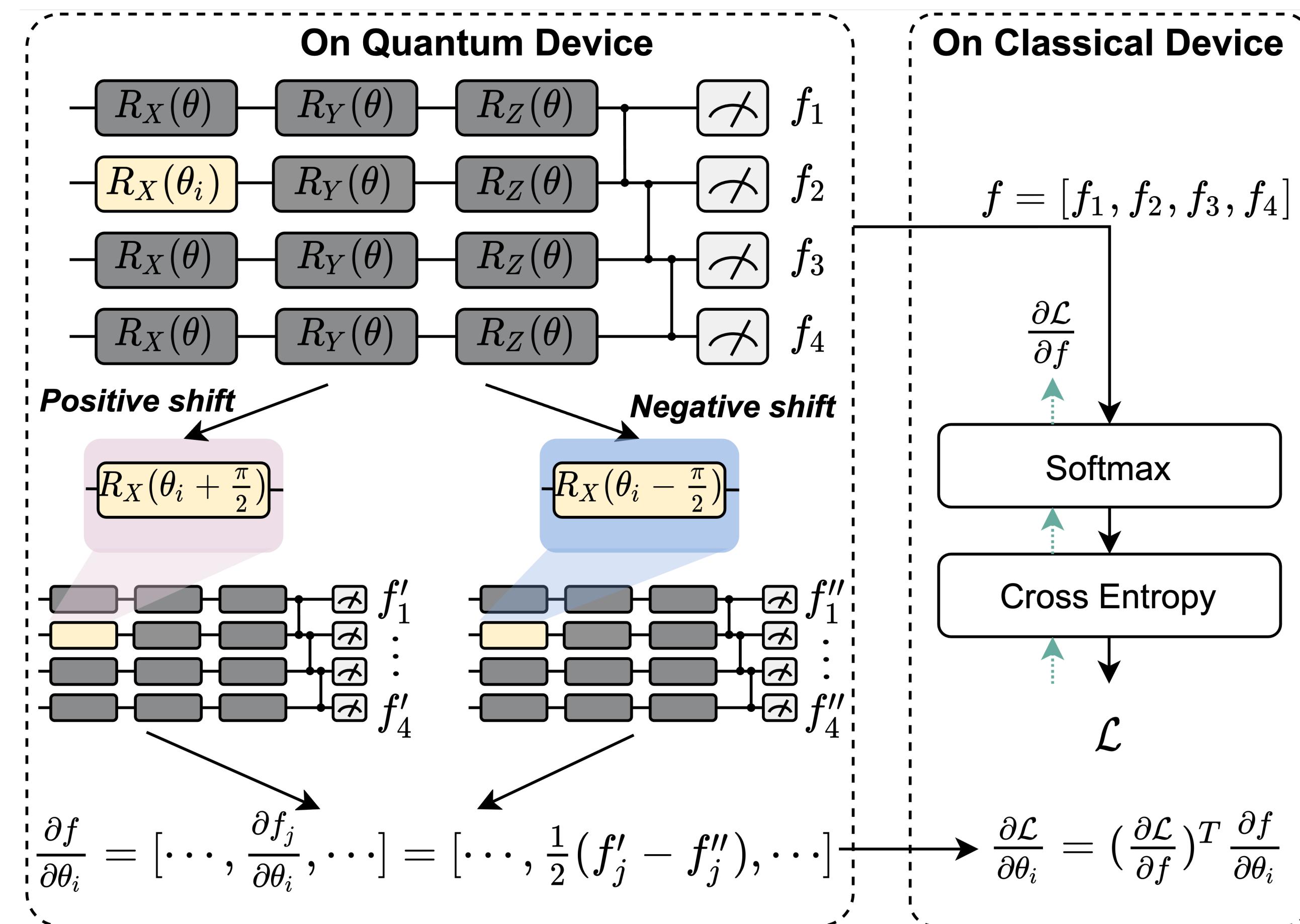
# Calculate Gradients of PQC

- Step 4: Shift twice and run on QC to calculate  $\frac{\partial f(\theta)}{\partial \theta_i}$



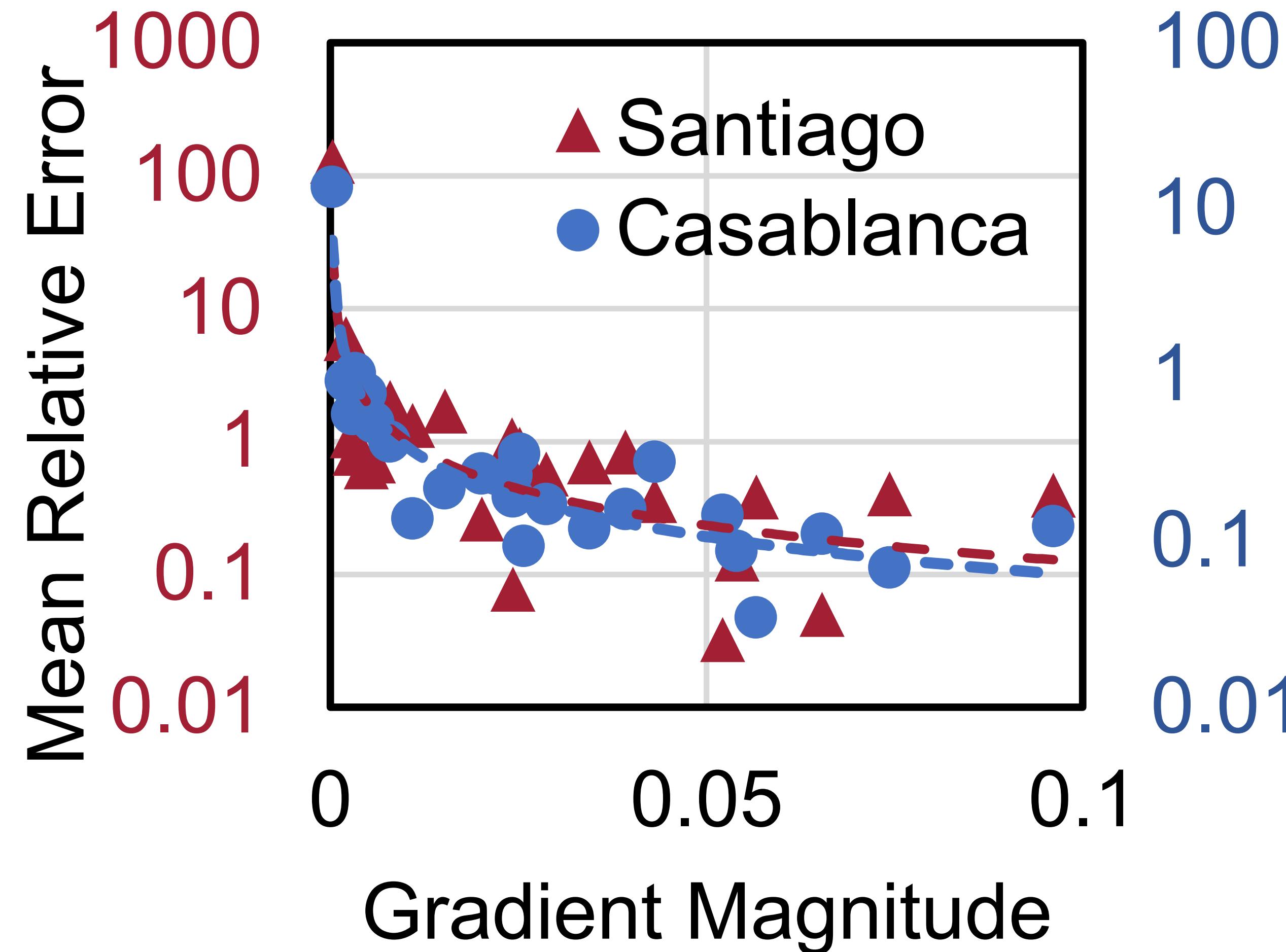
# Calculate Gradients of PQC

- Step 5: By Chain Rule:  $\frac{\partial \text{Loss}}{\partial f(\theta)} \frac{\partial f(\theta)}{\partial \theta_i} = \frac{\partial \text{Loss}}{\partial \theta_i}$ , sum over 4 passes (4 qubits)
- Only **forward** on quantum device



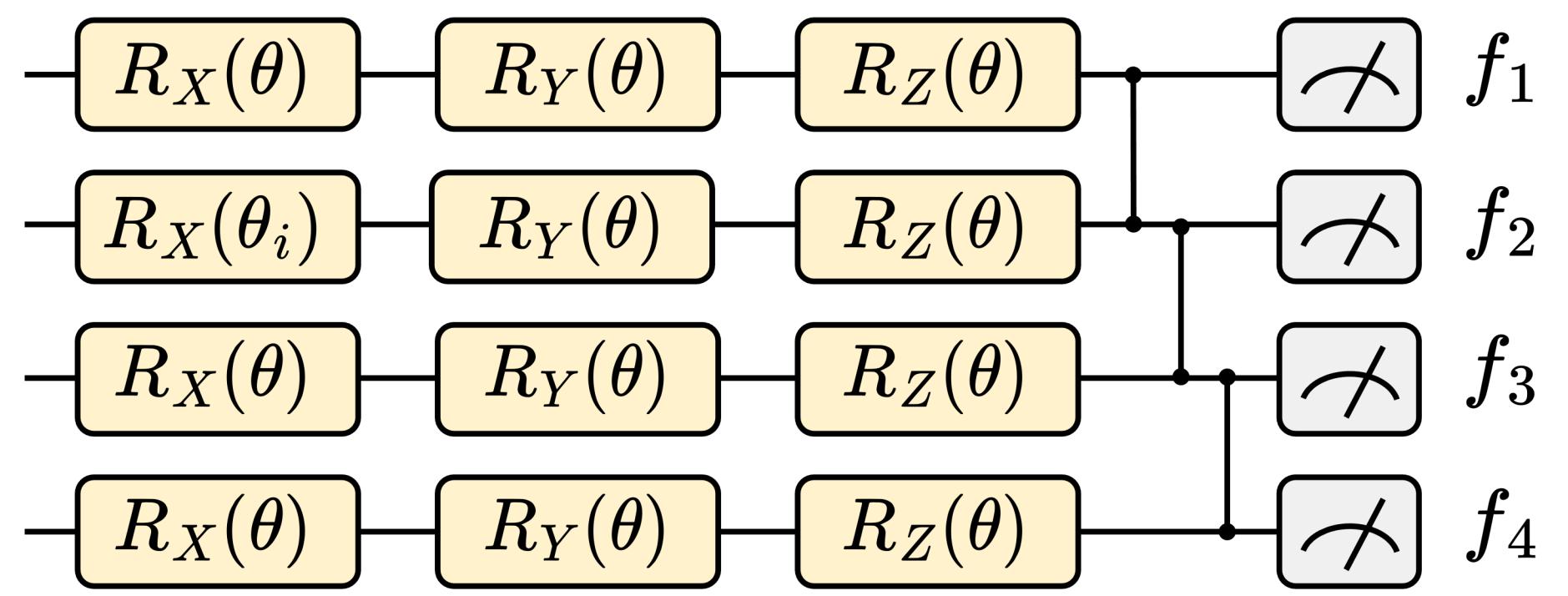
# Probabilistic Gradient Pruning

- Small magnitude gradients have **large** relative errors



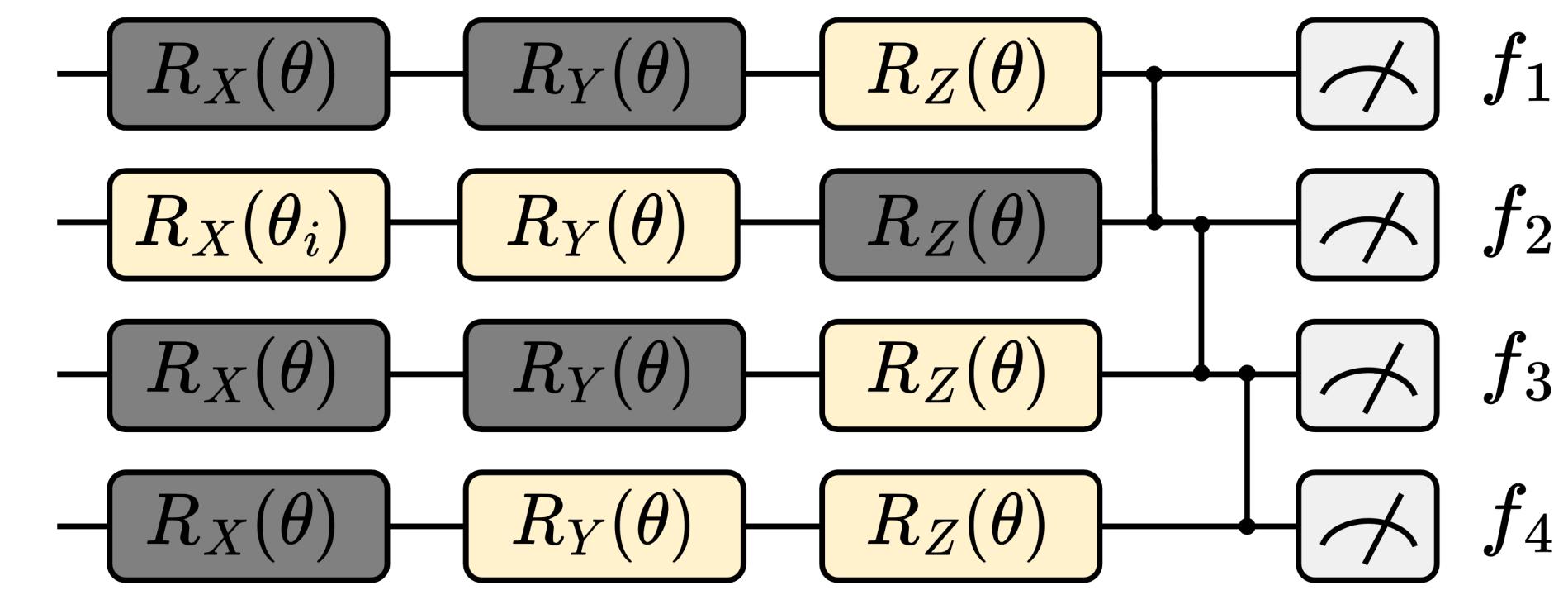
# Probabilistic Gradient Pruning

**Before pruning**



Only half the gradients  
calculated and updated

**After pruning**

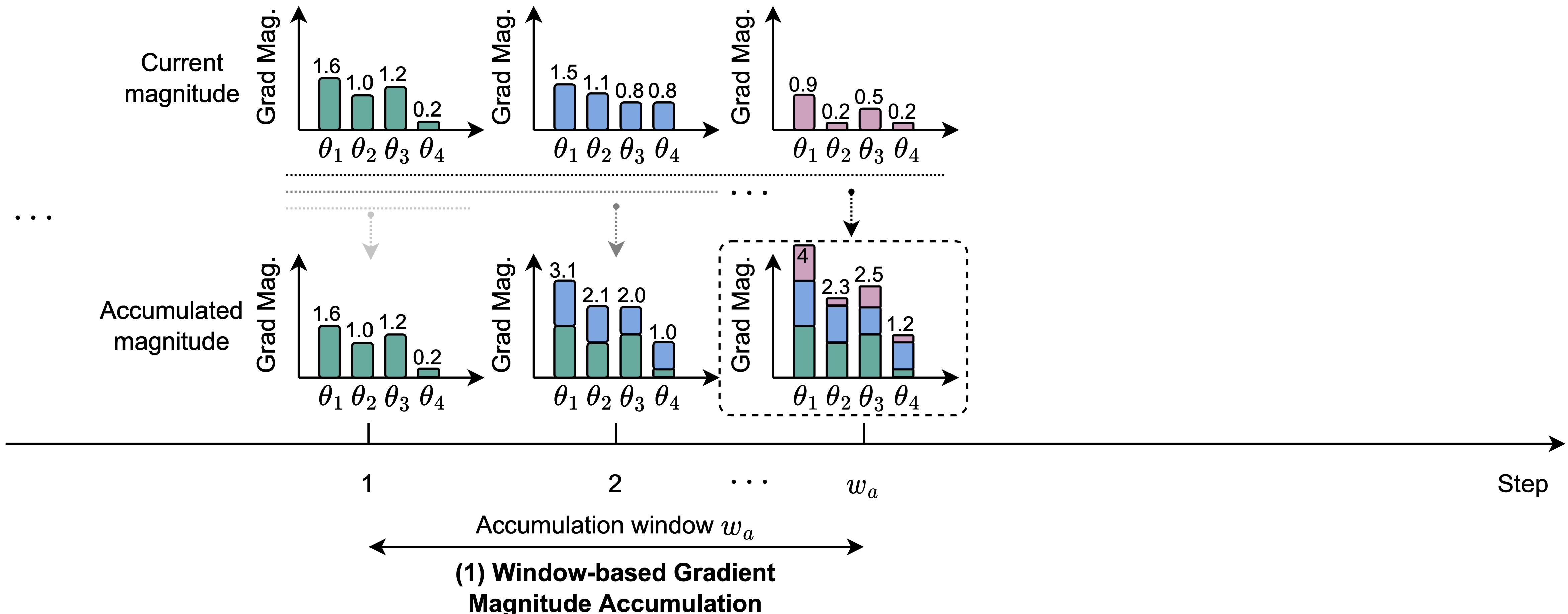


Update      Freeze

Update      Freeze

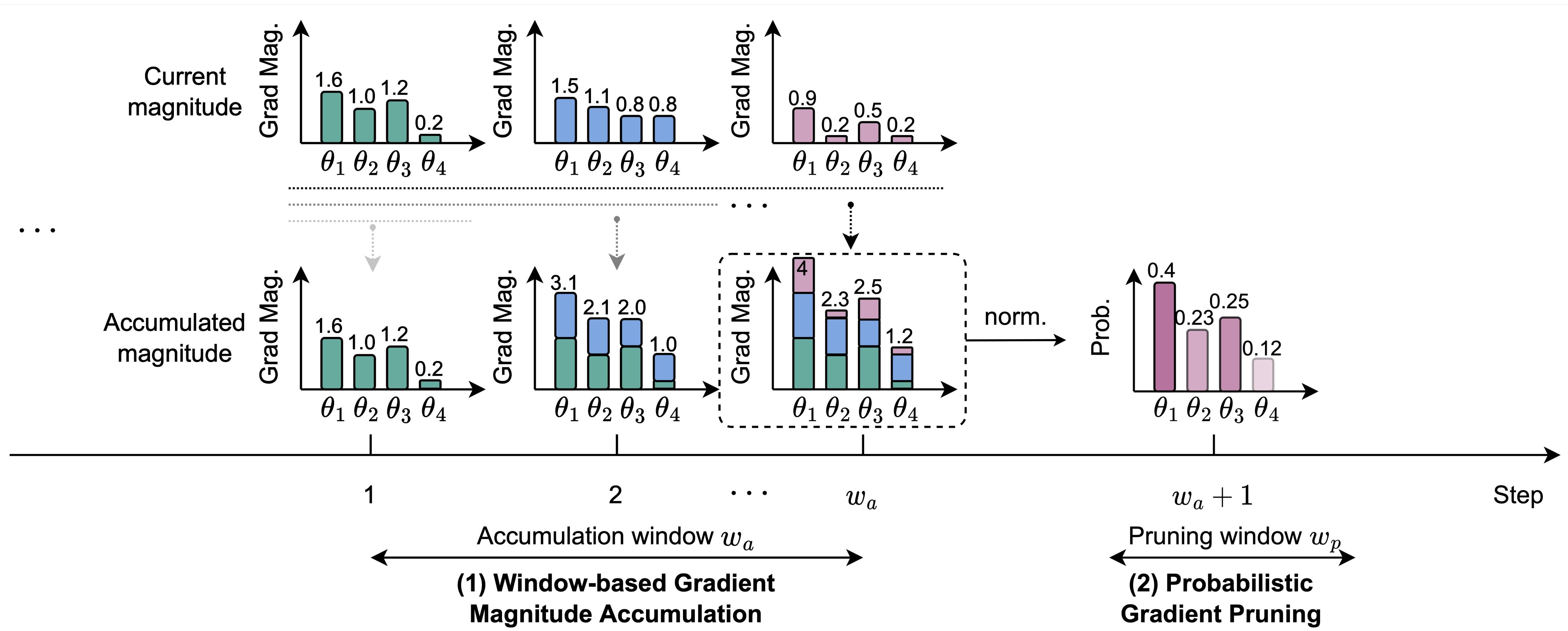
# Accumulation Window

- Keep a record of accumulated gradient magnitude



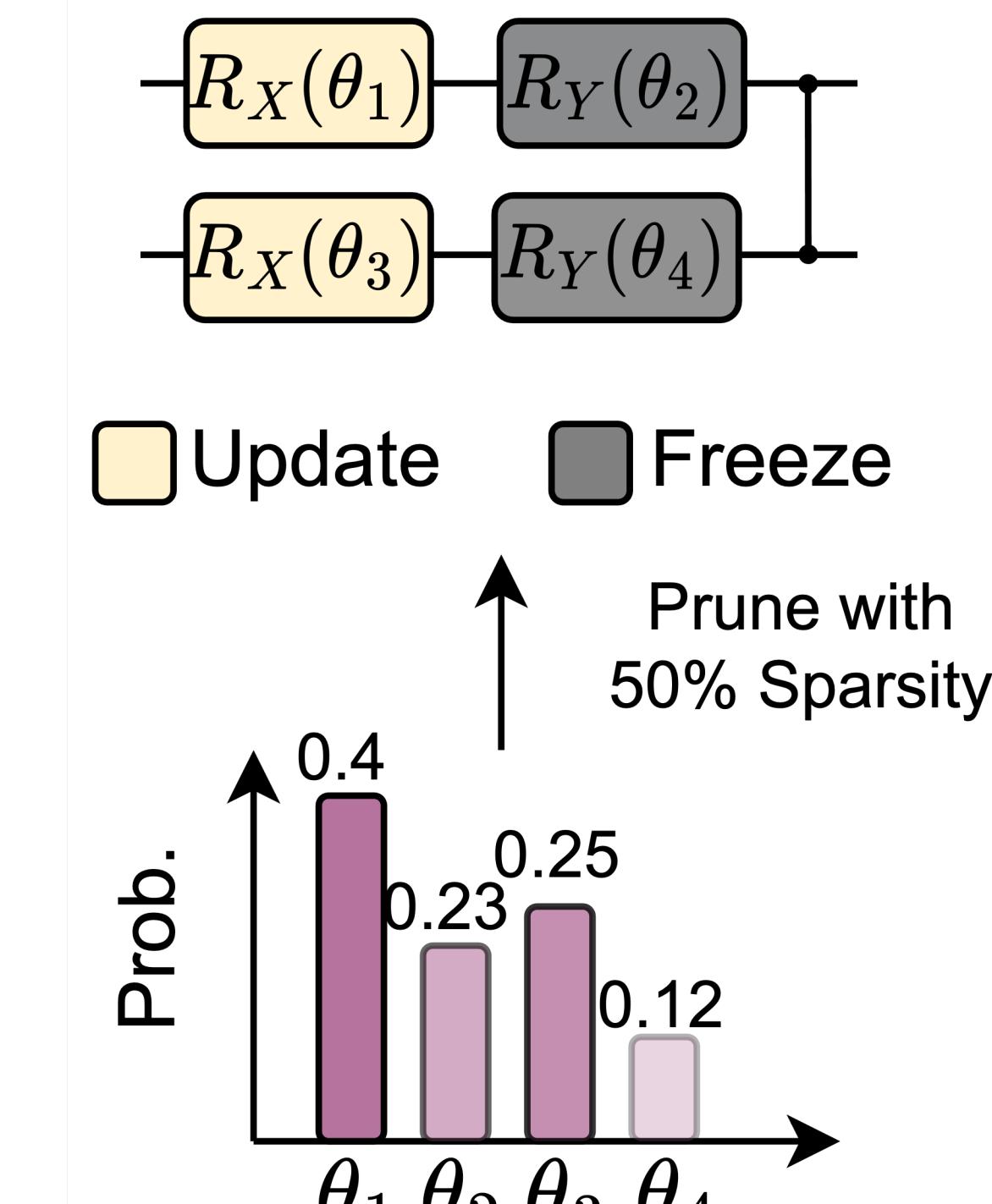
# Accumulation Window

- Normalize the accumulated gradient magnitude to a probability distribution



# Accumulation Window

- Prune the calculation of some gradients according to the probability distribution



# Evaluation

- Benchmarks
  - Quantum Machine Learning task: MNIST 4-class, 2-class, Fashion MNIST 4-class, 2-class, Vowel 4-class
  - Variational Quantum Eigensolver task: H<sub>2</sub> molecule
- Quantum Devices
  - IBMQ
  - #Qubits: 5 to 7
  - Quantum Volume: 8 to 32
- Circuit architecture
  - RZZ+RY, RXYZ+CZ, RZX+RXX

# Evaluation

- Benchmarks
  - Quantum Machine Learning task: MNIST 4-class, 2-class, Fashion MNIST 4-class, 2-class, Vowel 4-class
  - Variational Quantum Eigensolver task: H<sub>2</sub> molecule
- Quantum Devices
  - IBMQ
  - #Qubits: 5 to 7
  - Quantum Volume: 8 to 32
- Circuit architecture
  - RZZ+RY, RXYZ+CZ, RZX+RXX

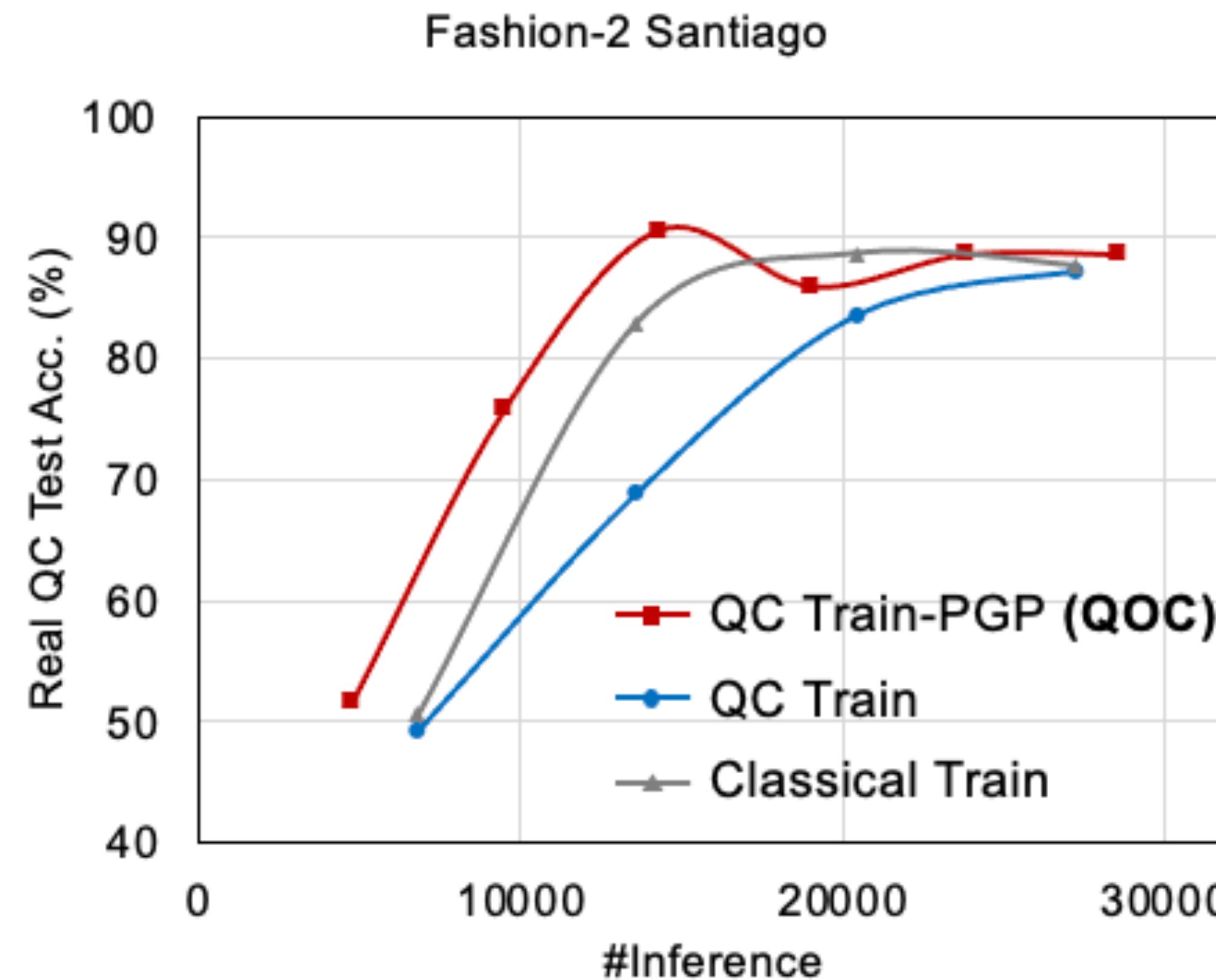
# QNN results

- QOC achieved similar results to classical simulation

Method	Acc.	MNIST-4	MNIST-2	Fashion-4	Fashion-2	Vowel-4
		Jarkata	Jarkata	Manila	Santiago	Lima
Classical-Train	Simu.	0.61	0.88	0.73	0.89	0.37
Classical-Train		0.59	0.79	0.54	0.89	0.31
QC-Train	QC	0.59	0.83	0.49	0.84	0.34
<b>QC-Train-PGP</b>		<b>0.64</b>	<b>0.86</b>	<b>0.57</b>	<b>0.91</b>	<b>0.36</b>

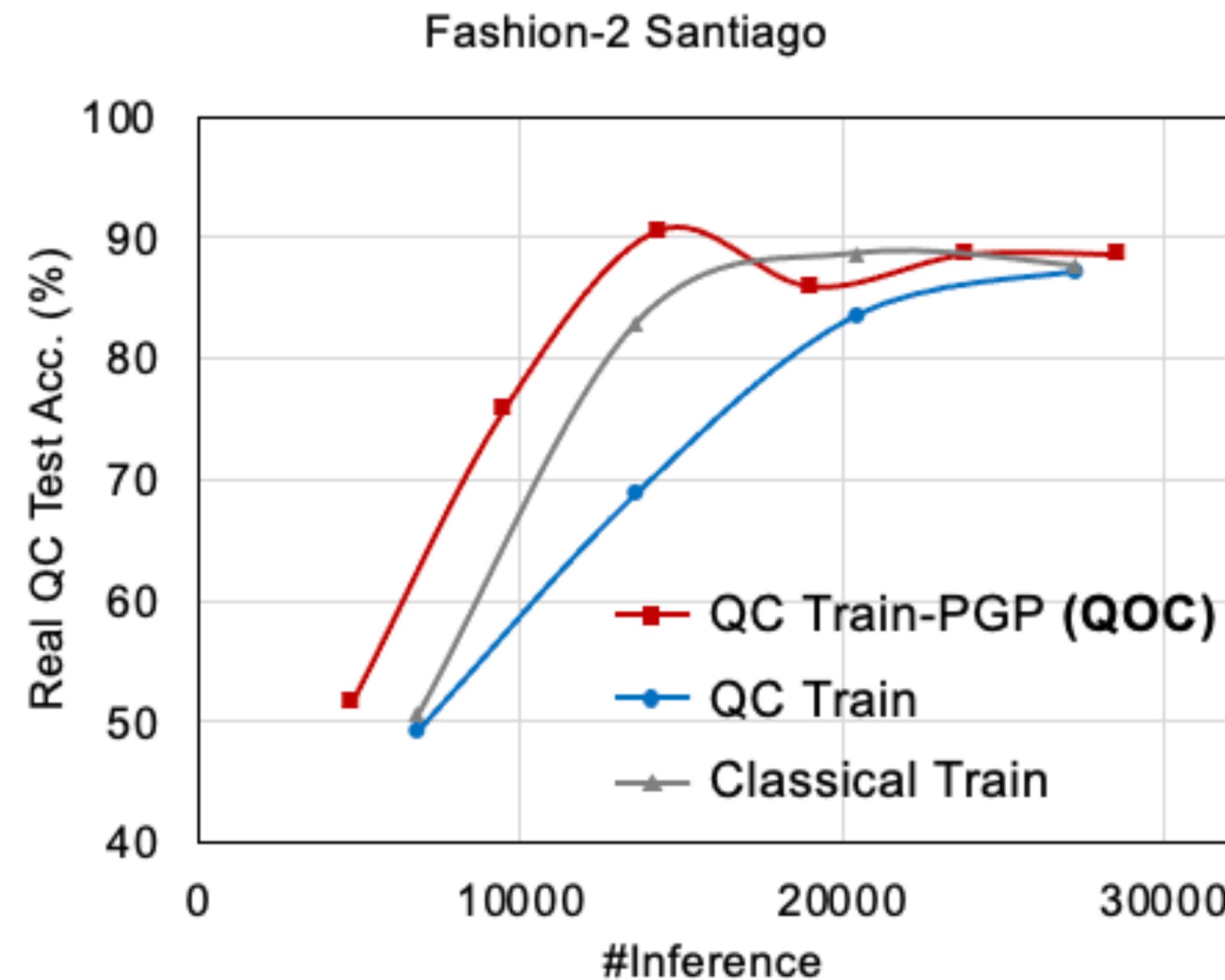
# QNN Training Curves

- Classical Train: Train on classical simulator and test on real QC
- QC Train: Train and test the model on real QC
- QC Train-PGP (**QOC**): Train and test on real QC with gradient pruning



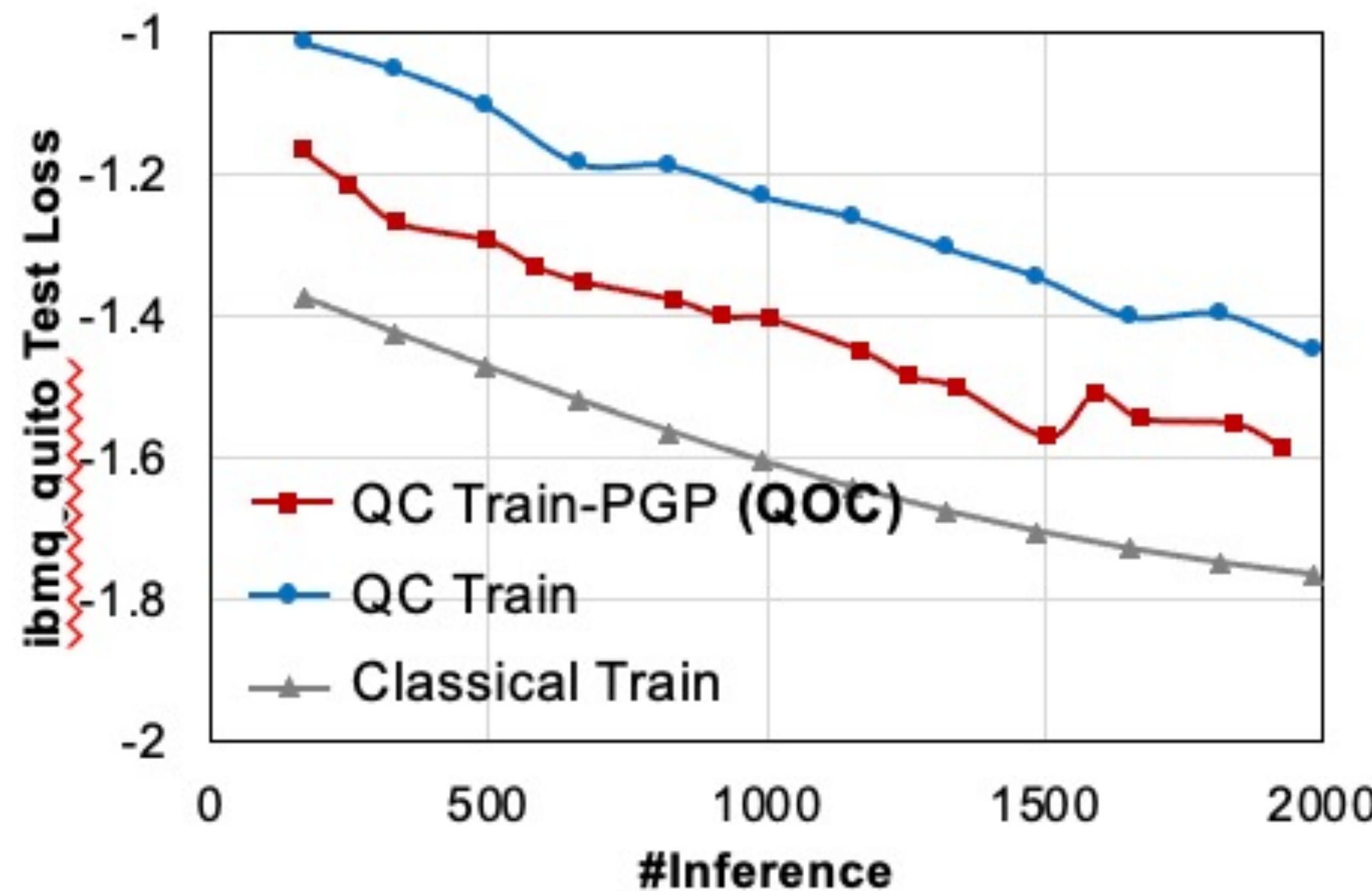
# QNN Training Curves

- Gradient pruning can brings **2%~4%** accuracy improvements
- Pruning **accelerates convergence** with **2x** training time reduction



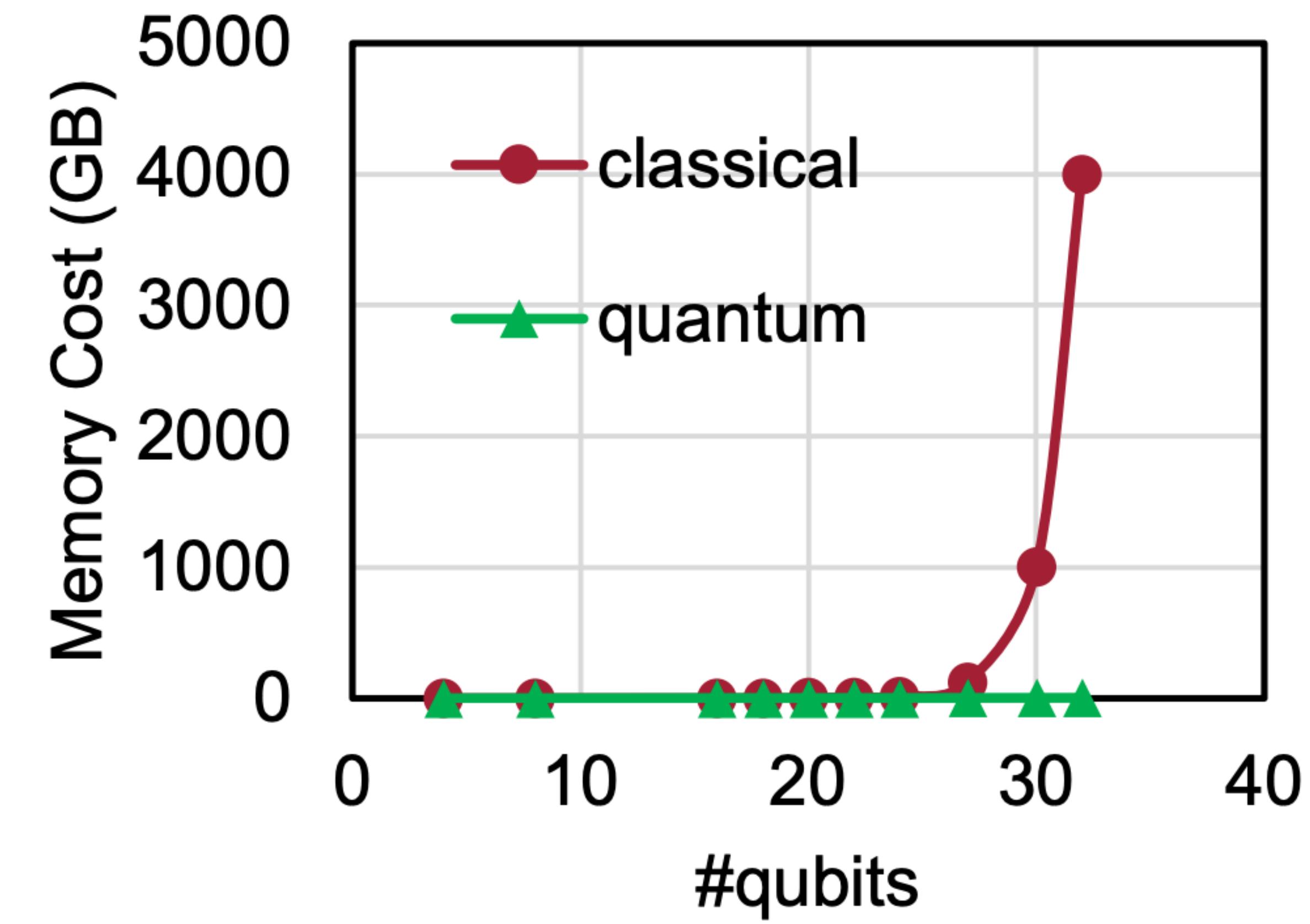
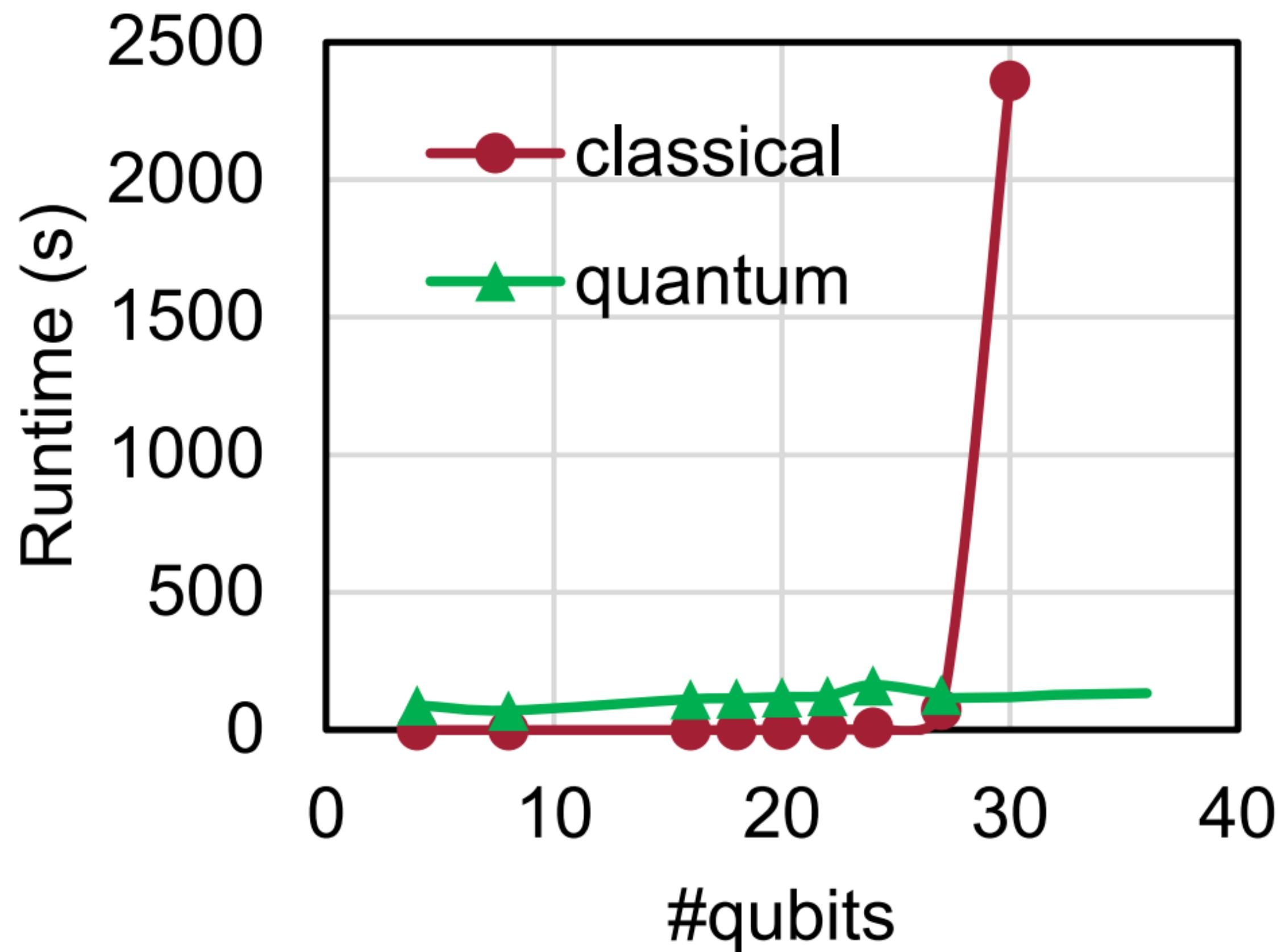
# VQE Training Curves

- Gradient pruning can **reduce the gap** between quantum and classical



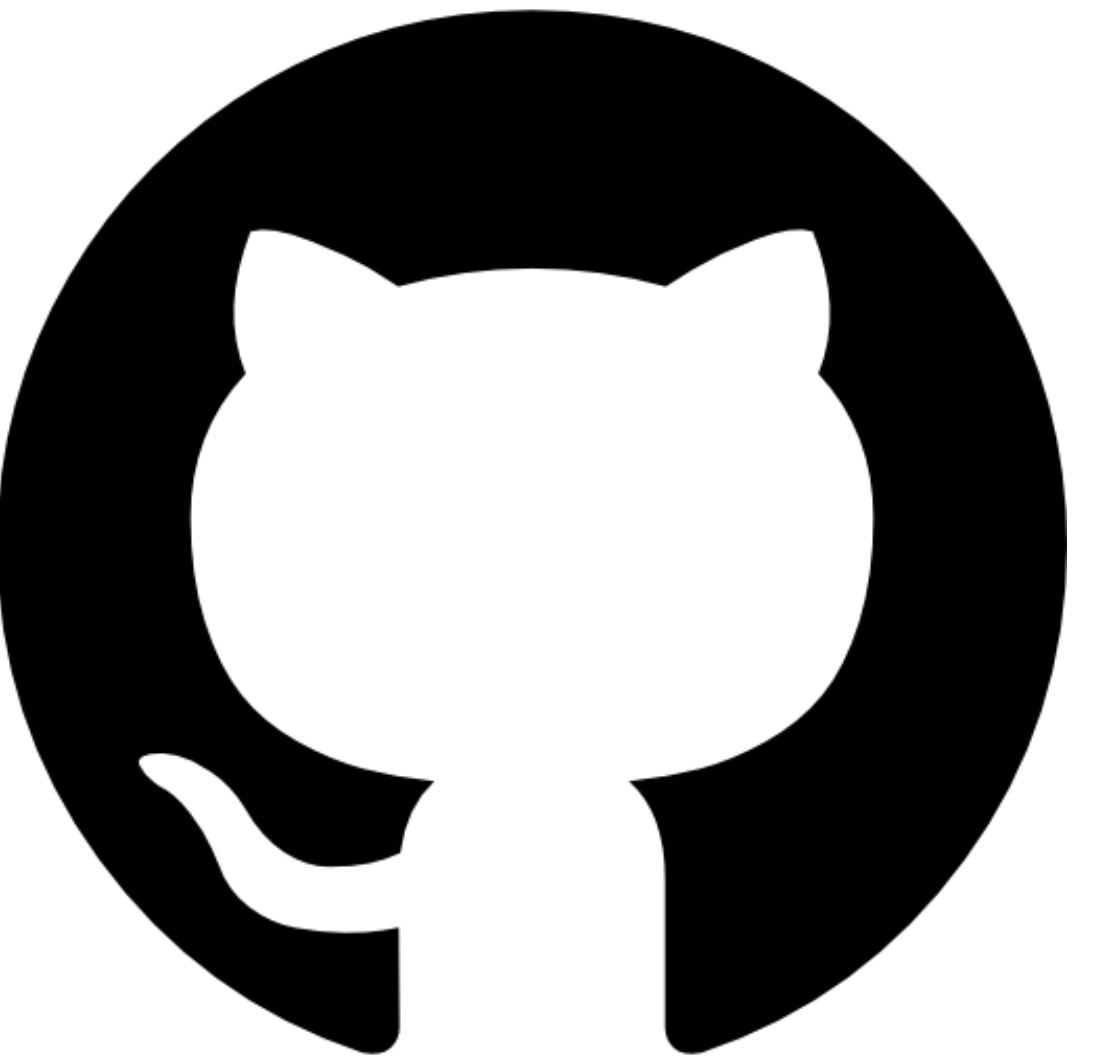
# Scalability Improvements

- On-chip Training is scalable



# Hands-On Section

## 2.3 Quantum On-chip Training



# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations 

1.3 TQ for State Prep 

1.4 TQ for VQE 

1.4 TQ for QNN 

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning 

2.2 QuantumNAT: Noise Injection and Quantization 

2.3 QOC: On-Chip Training 

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression 

## Section 3

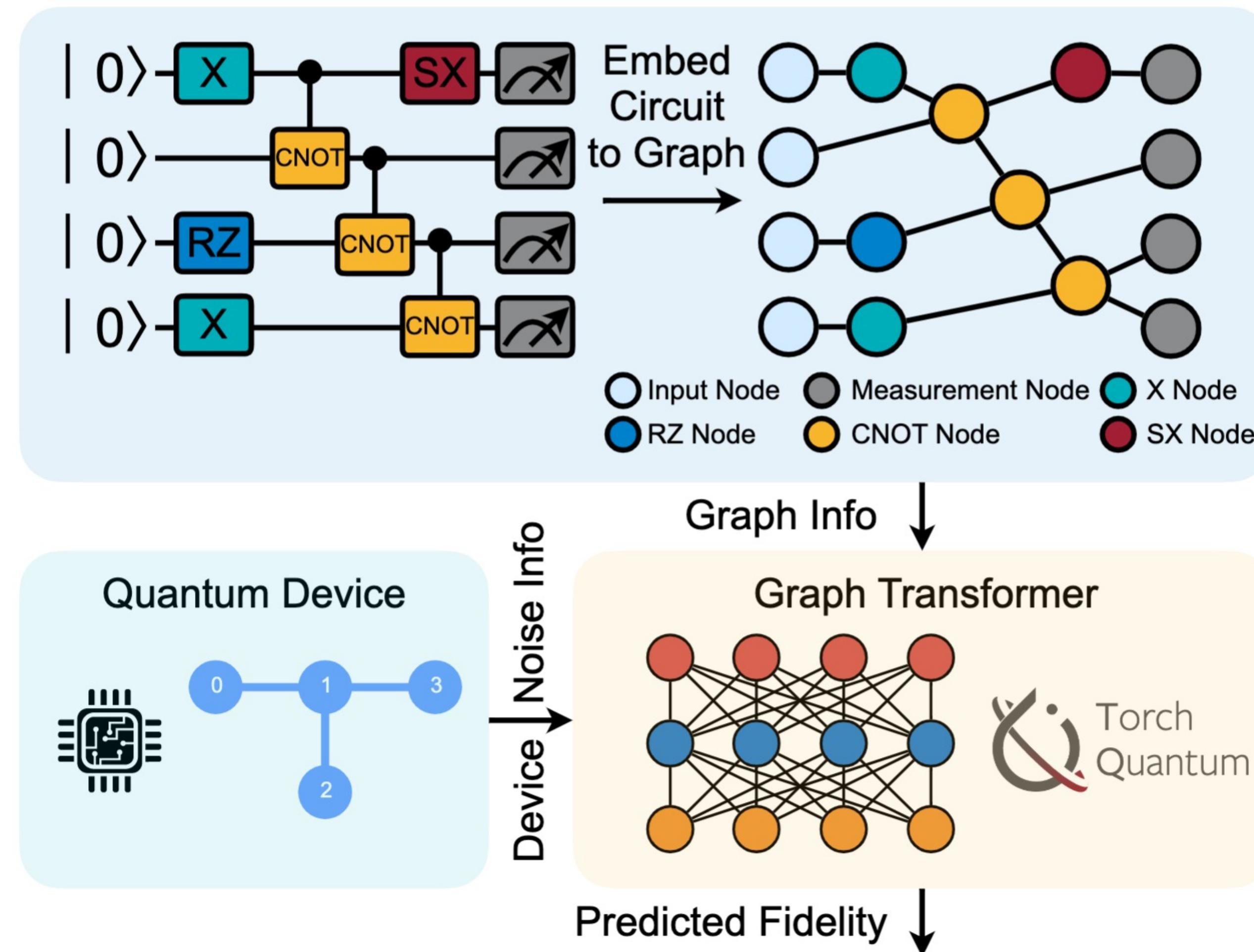
### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control 

3.2 Variational Pulse Learning 

# Transformer for Quantum Circuit Reliability Prediction

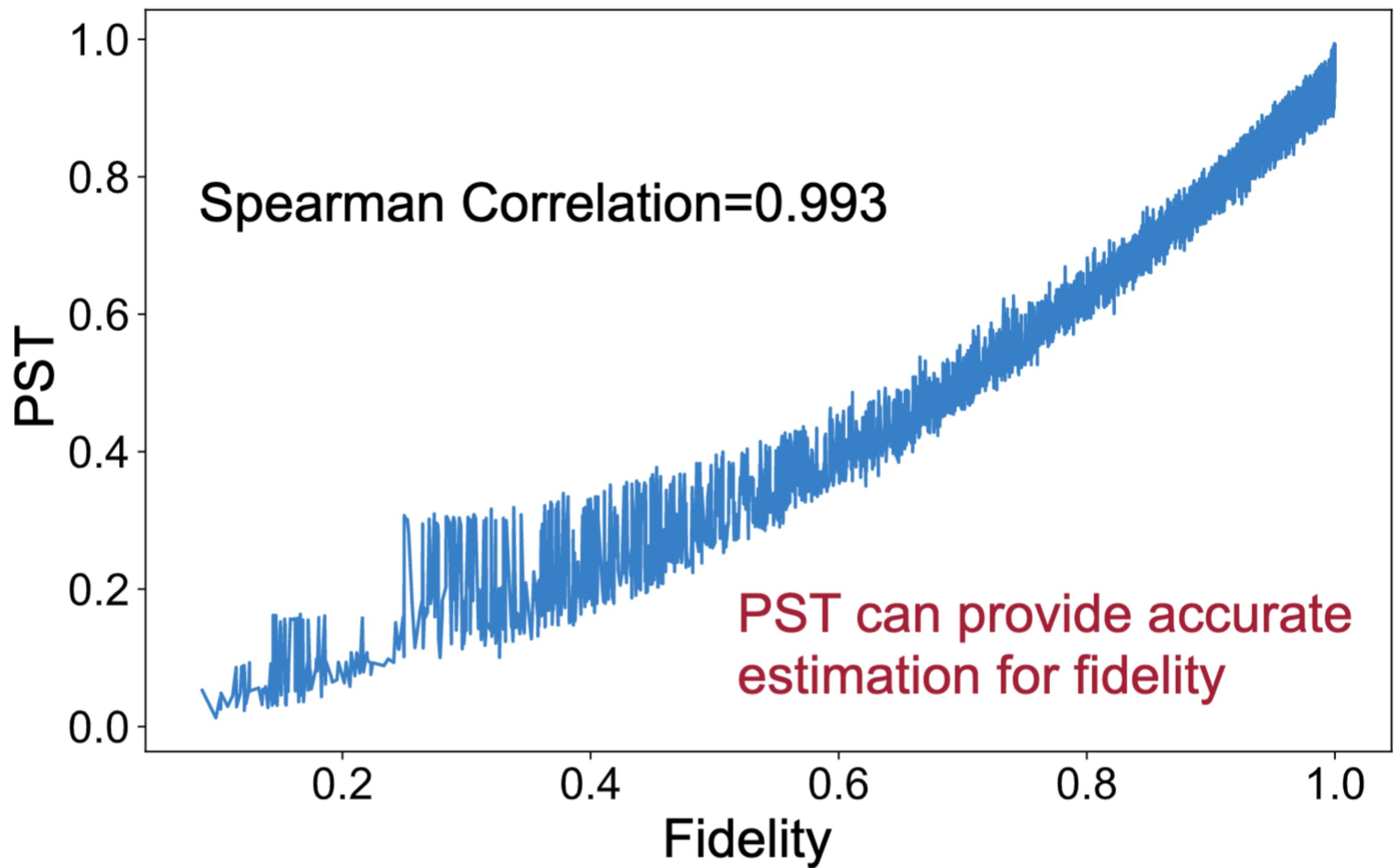
- Use the circuit graph information



# Transformer for Quantum Circuit Reliability Prediction

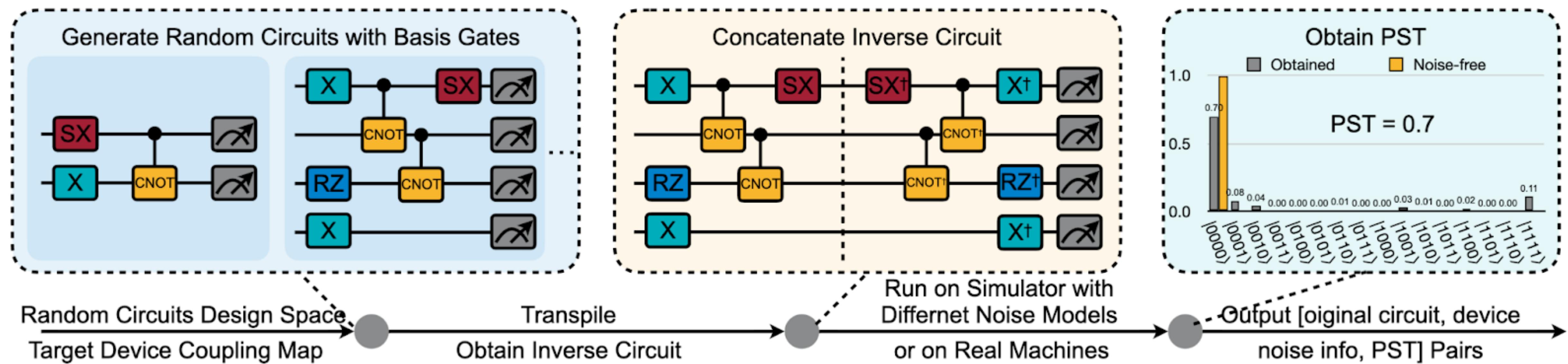
- Use PST as the metrics

$$PST = \frac{\#Trials \text{ with output same as initial state}}{\#Total trials}$$



# Dataset collection

- Collect dataset on real machine / noisy simulator

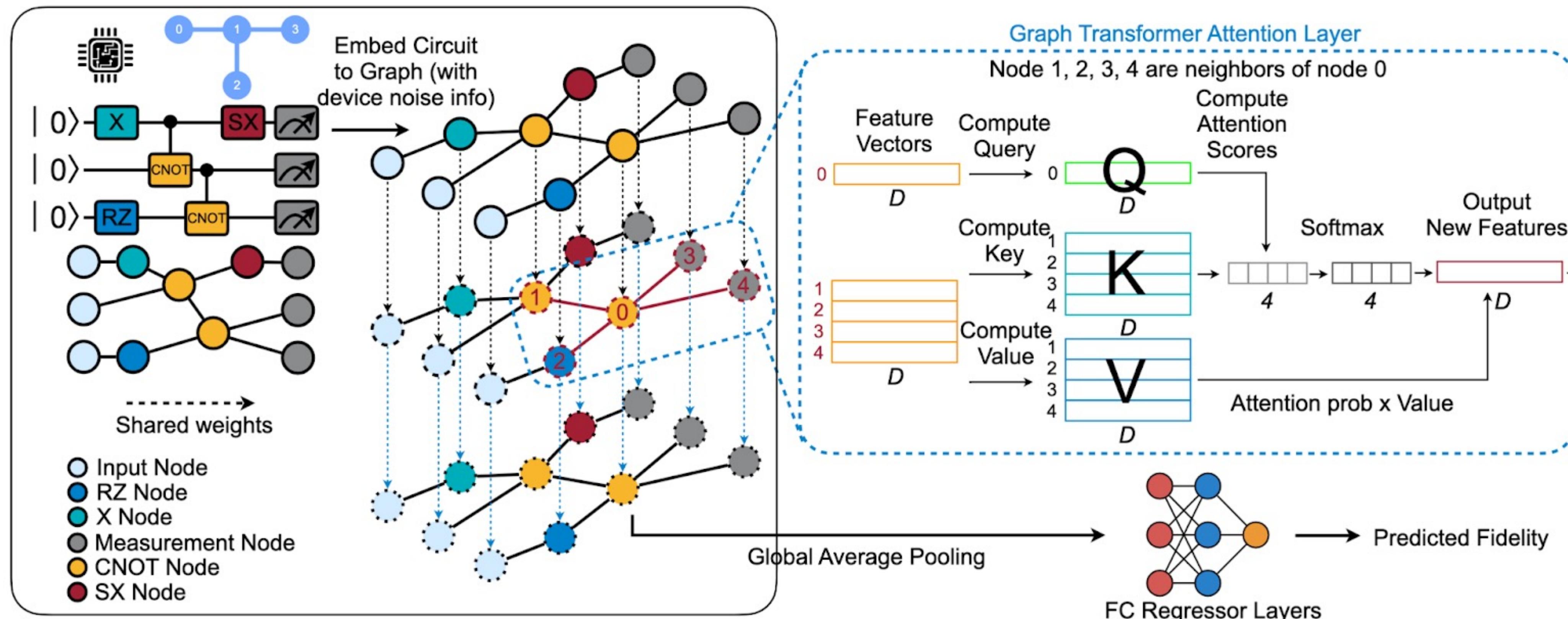


# Features on each node

0, 1, 0, 0, 0, 0,	0, 1, 0, 0, 0, 0, 0, 0, 0,	140.3, 200.2,	120.5, 230.6,	0.004,	0.03,	0.05,	11
One-Hot Node Type	One-Hot Gate Qubit	First Qubit T1, T2	Second Qubit T1, T2	Gate Error Rate	Readout Error 0 - 1	Readout Error 1 - 0	Gate Index

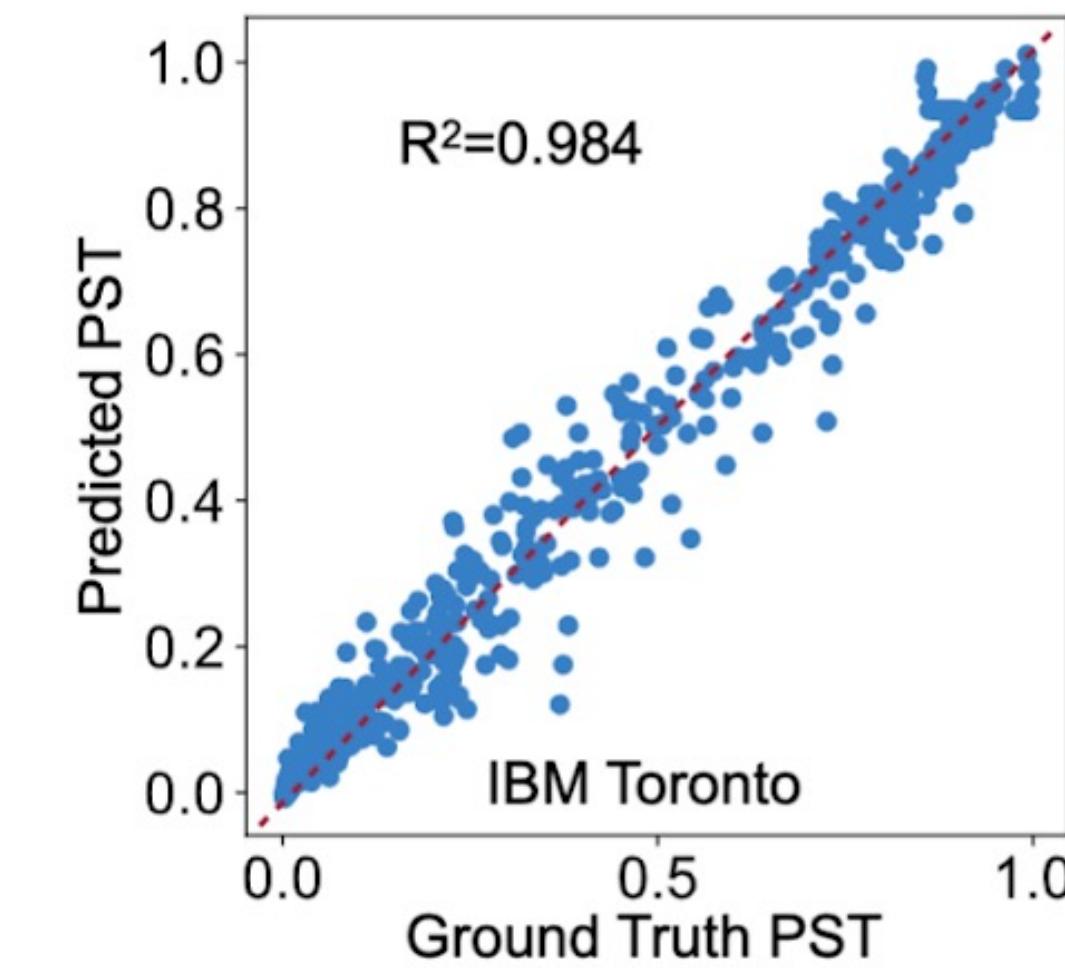
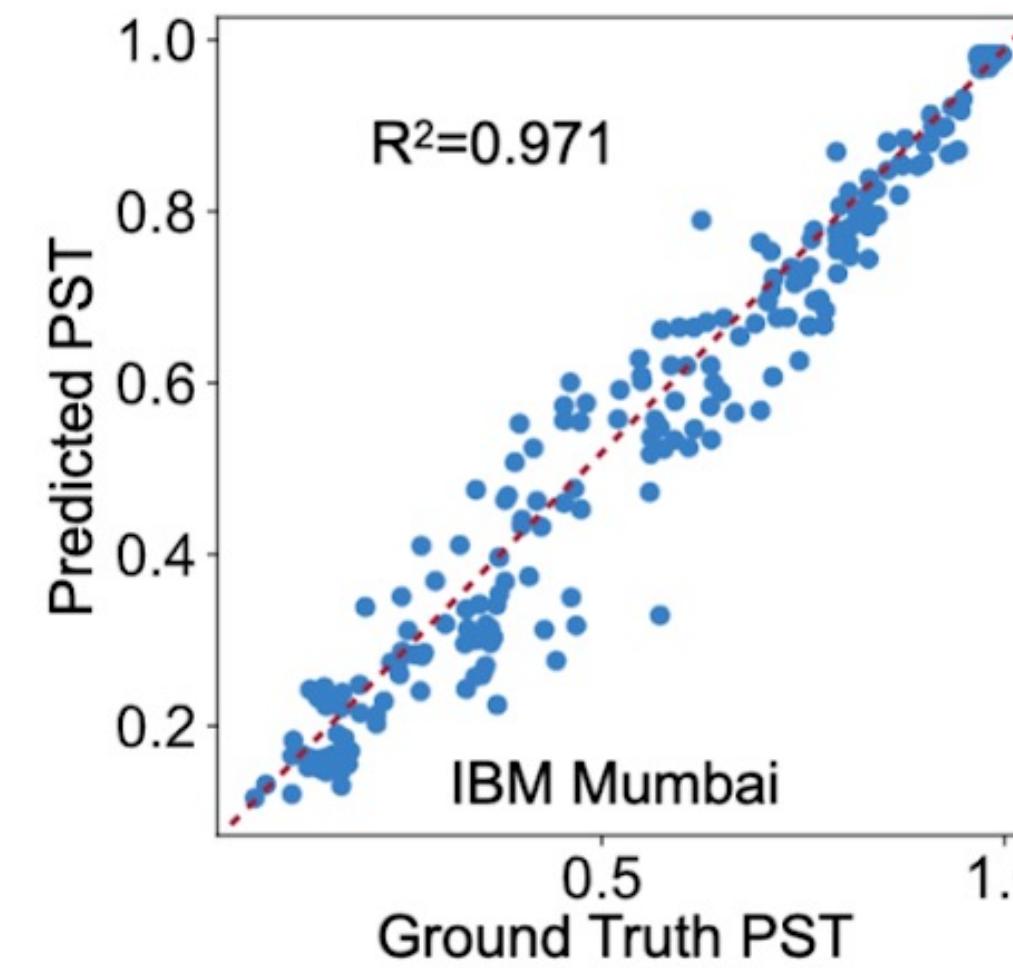
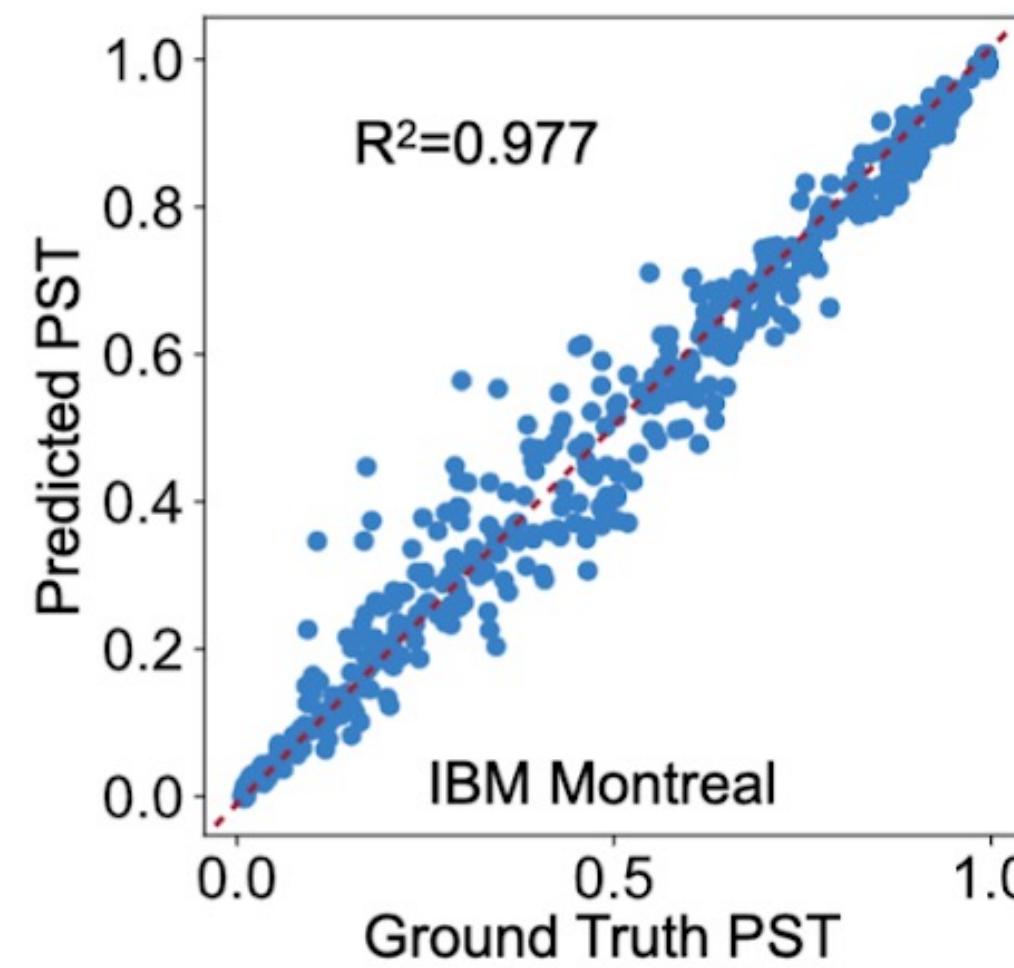
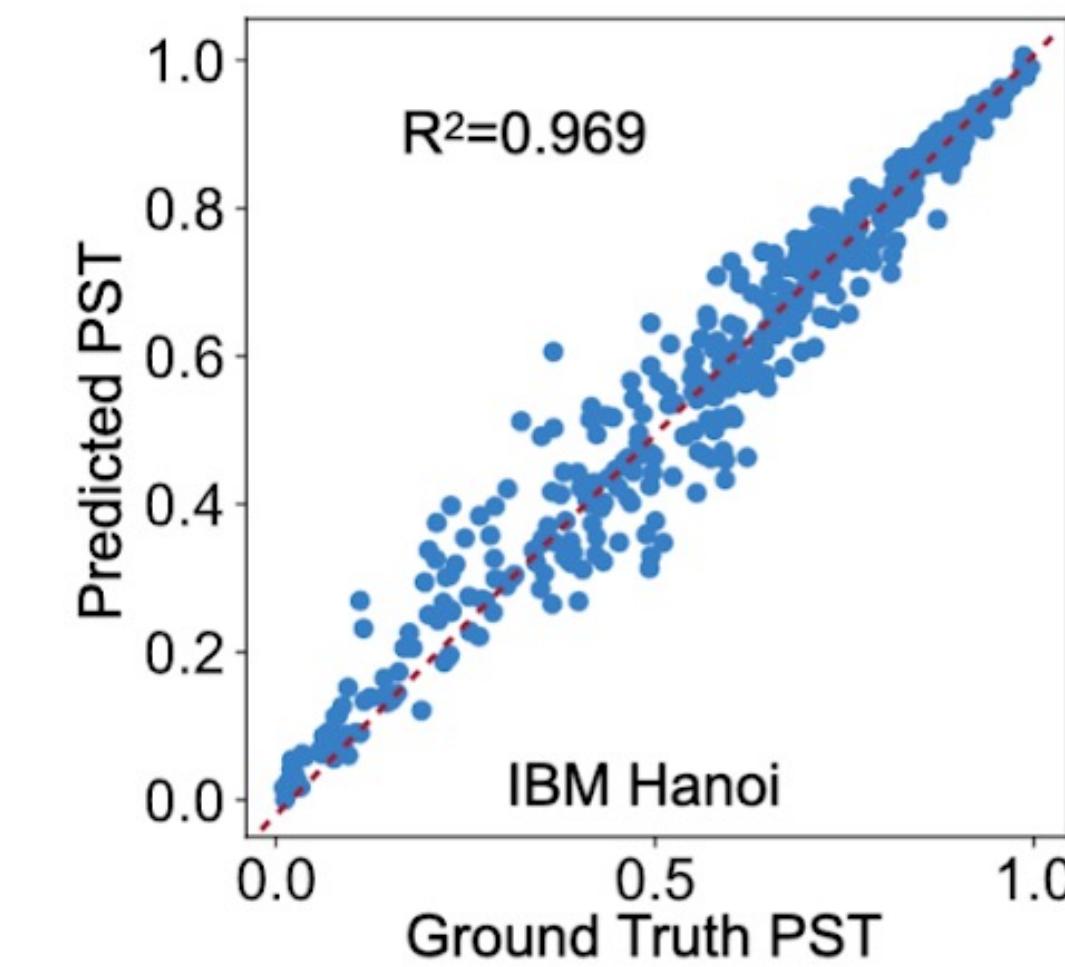
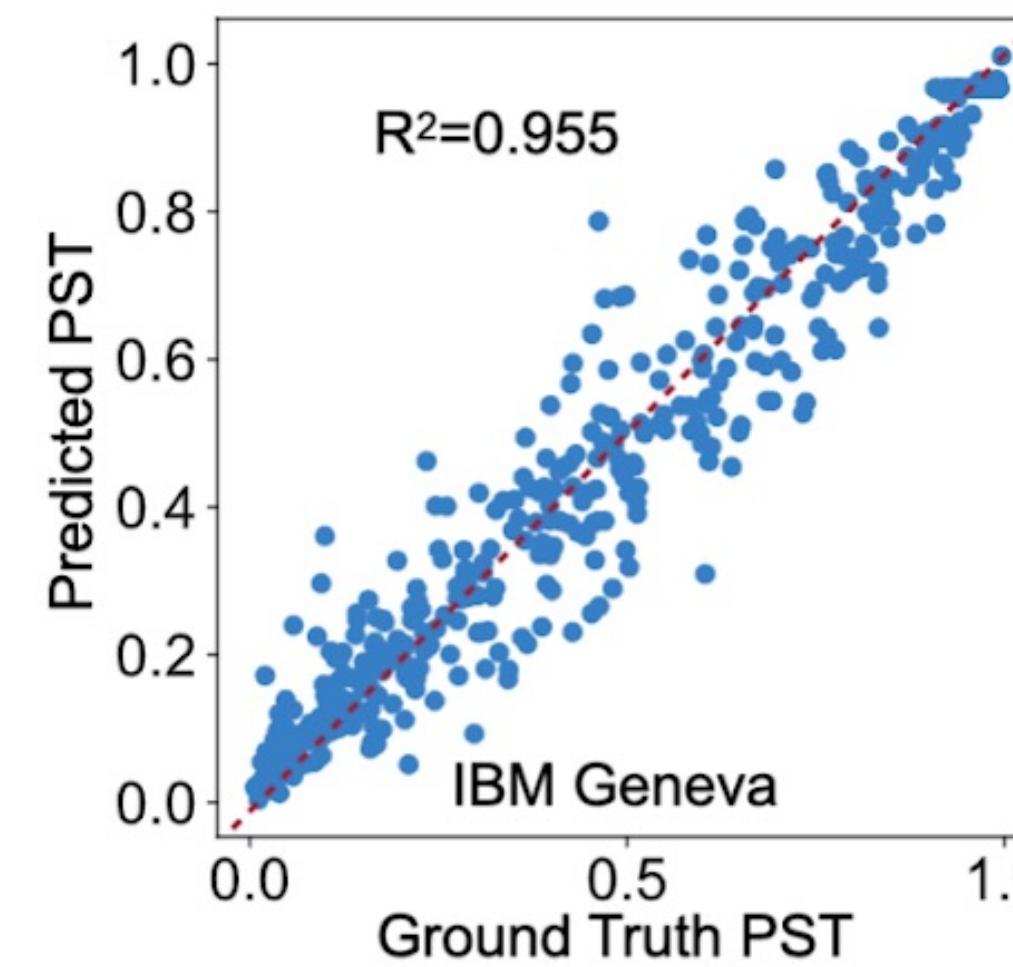
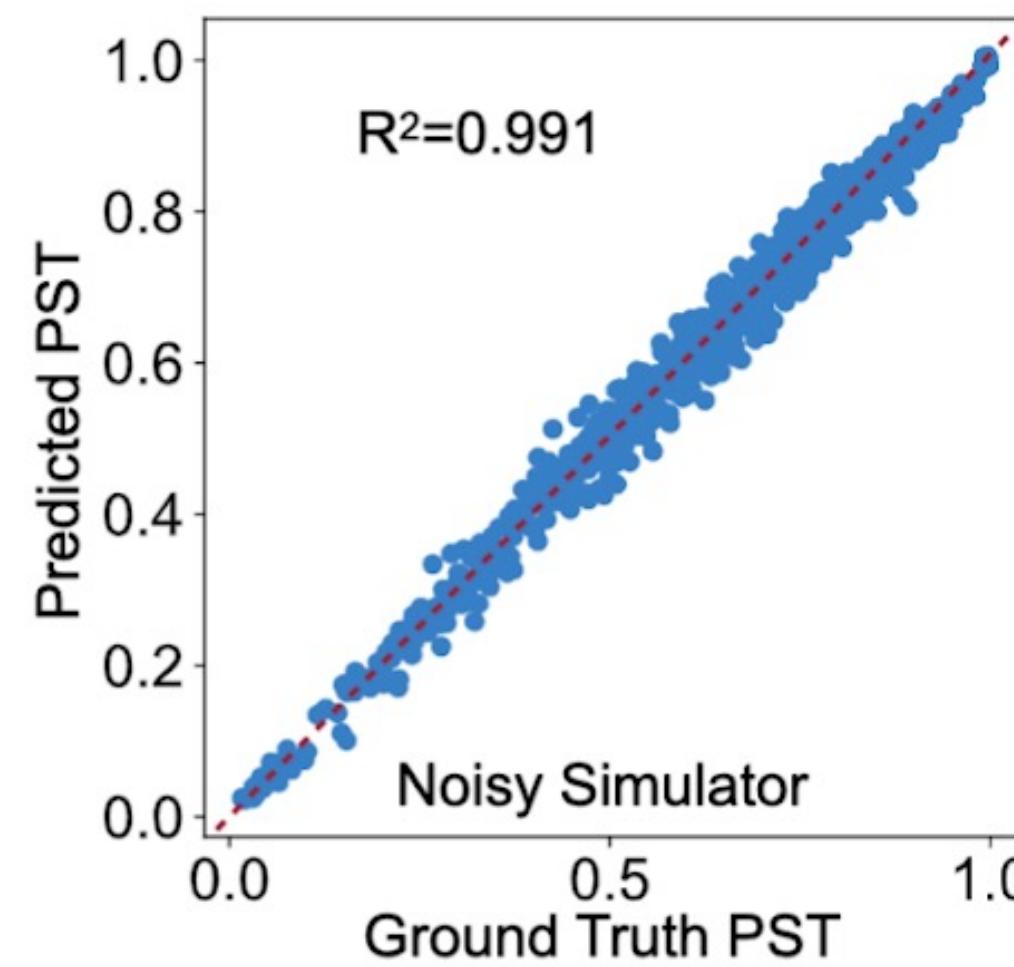
# Graph Transformer

- Graph Transformer



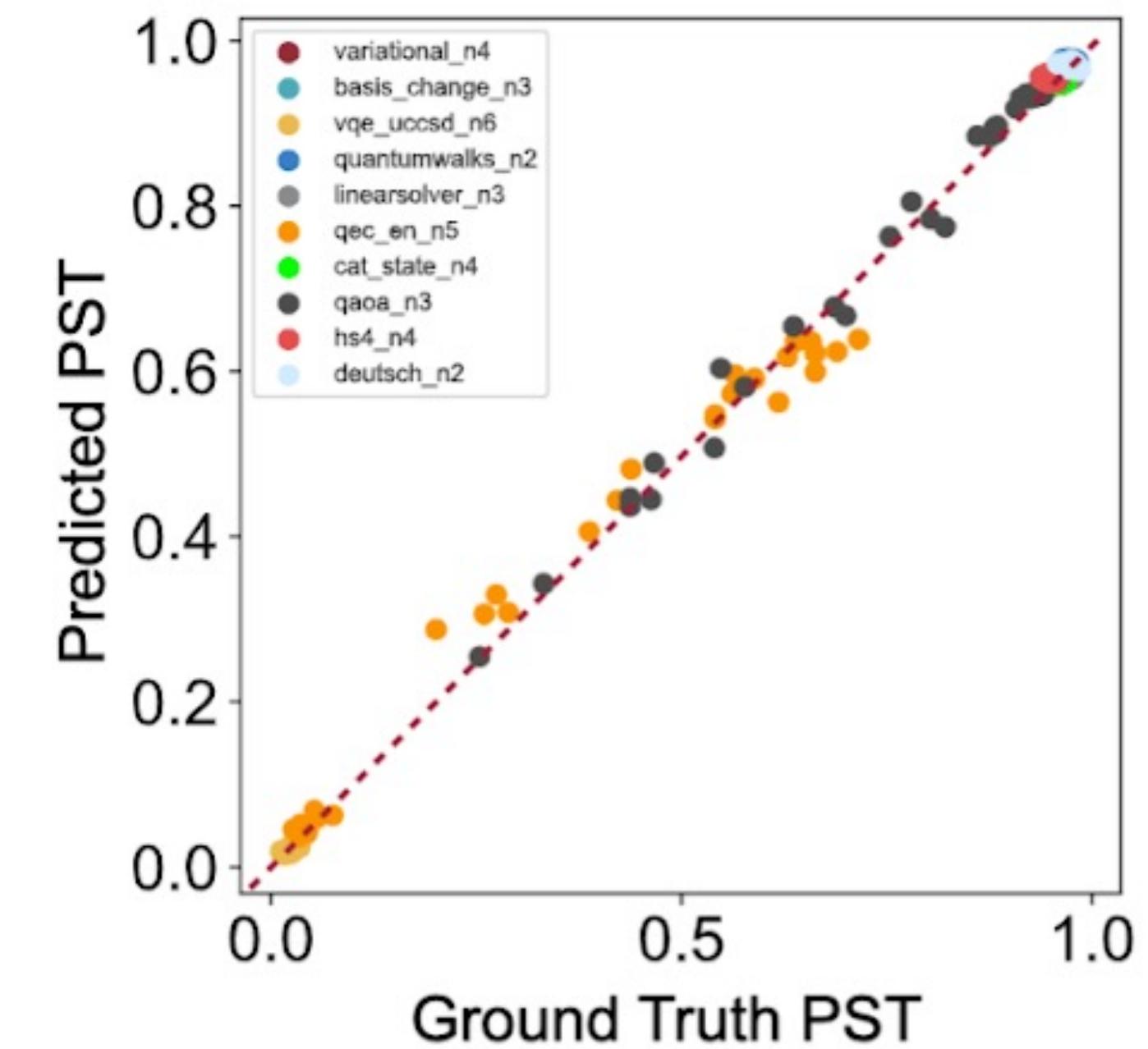
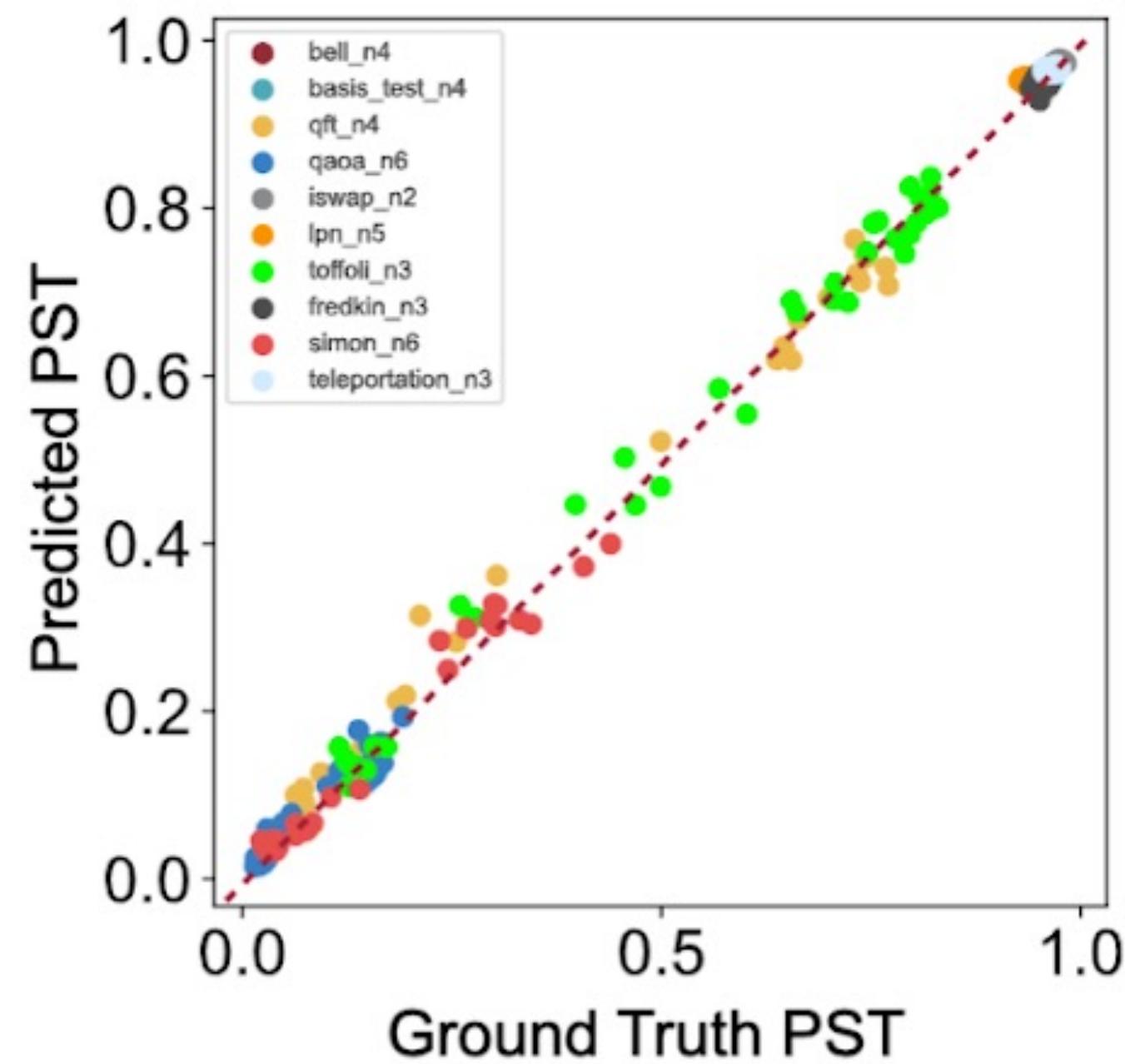
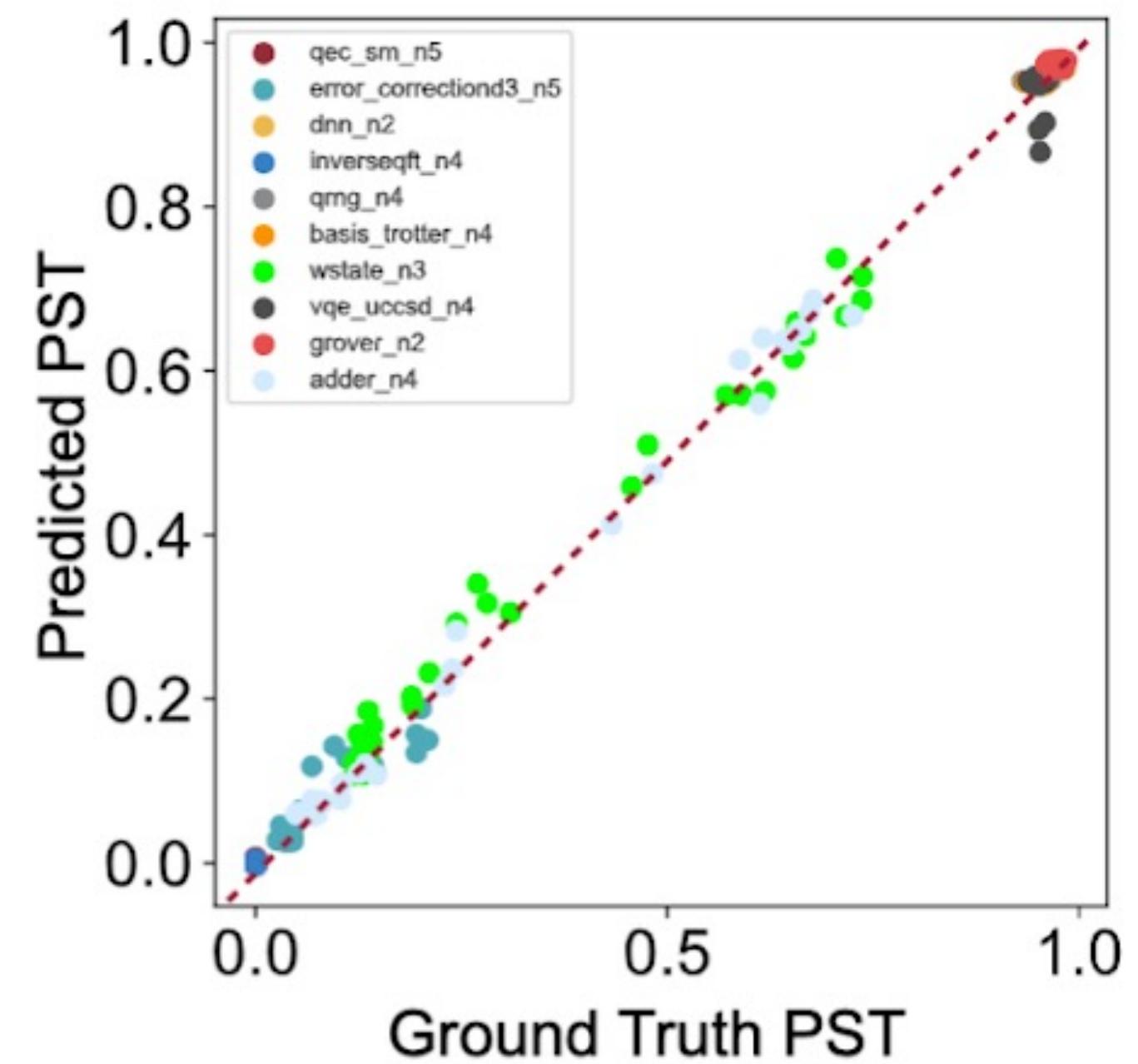
# Evaluation

- On random generated circuit



# Evaluation

- Circuits from quantum algorithms



# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations

1.3 TQ for State Prep

1.4 TQ for VQE

1.4 TQ for QNN

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning

2.2 QuantumNAT: Noise Injection and Quantization

2.3 QOC: On-Chip Training

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression

## Section 3

### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control

3.2 Variational Pulse Learning

# How to Compress a Quantum Neural Network?

## Quantum Neural Network Compression

---

<https://arxiv.org/pdf/2207.01578.pdf>

**Accepted by IEEE/ACM International Conference on Computer-Aided Design 2022**

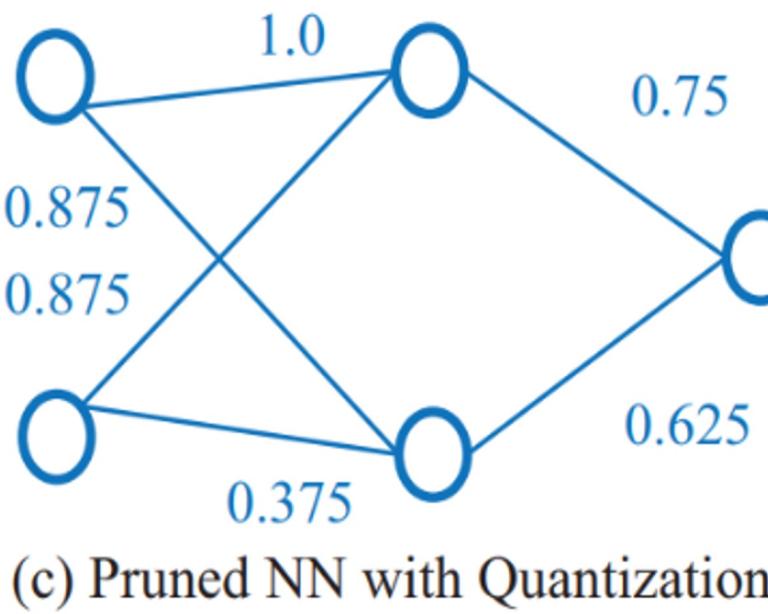
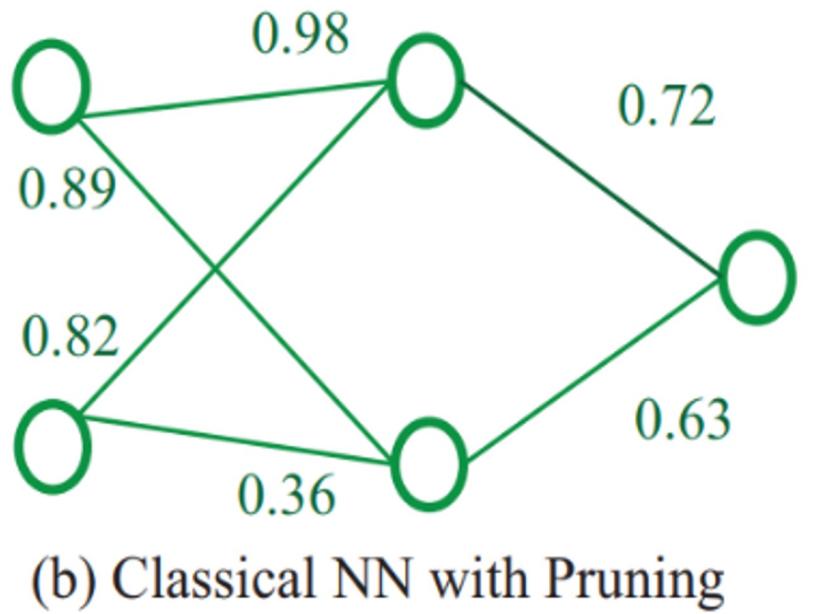
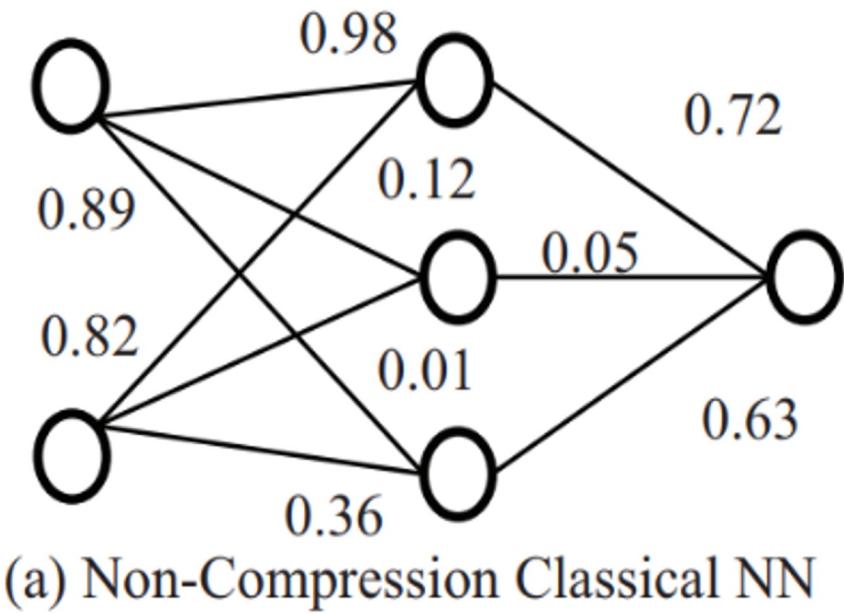
Zhepeng Wang (Presenter), Zhirui Hu, Dr. Weiwen Jiang

Department of Electrical and Computer Engineering

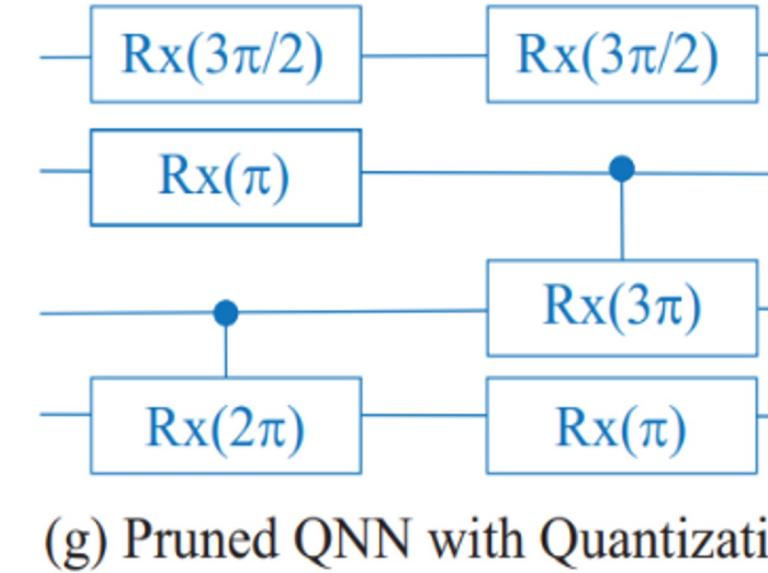
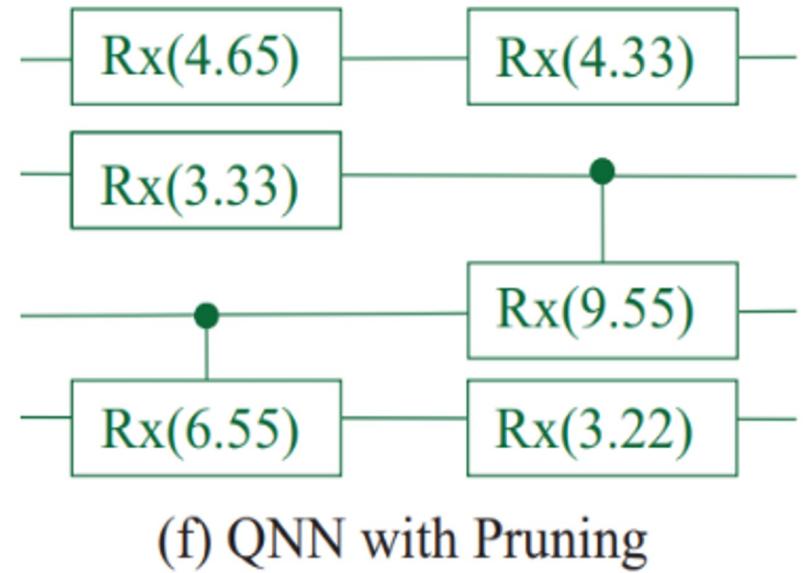
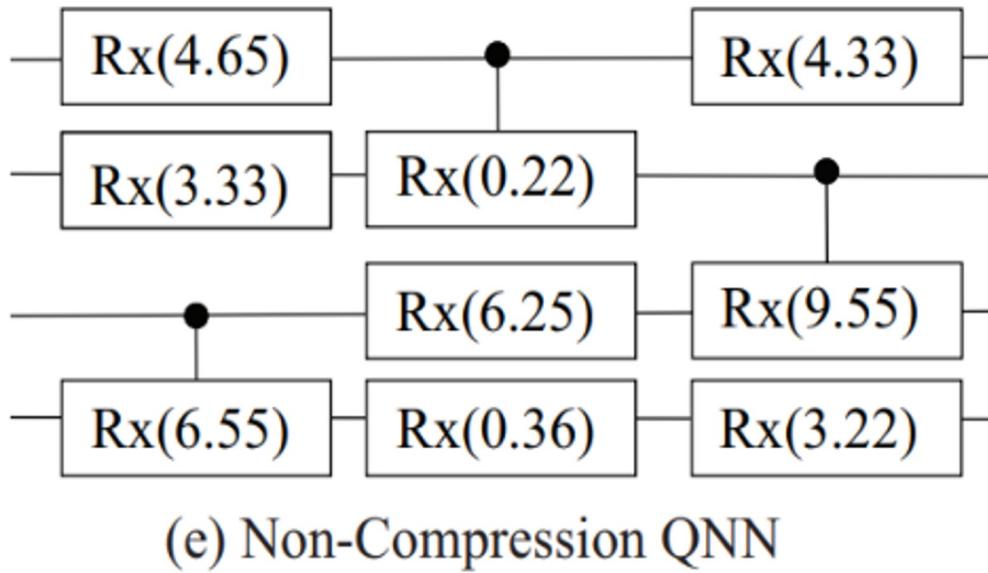
Jqub @ George Mason University

# Motivation and Background

- Pruning and Quantization in Classical ML



- Pruning and Quantization in Quantum ML



- **Pruning:** Not only 0 can be pruned, but also  $2\pi$ ,  $4\pi$ , etc.
- **Quantization:** Different quantization level may have different cost

# Motivation and Background

- Quantum Neural Network Compression Should be Compilation Aware

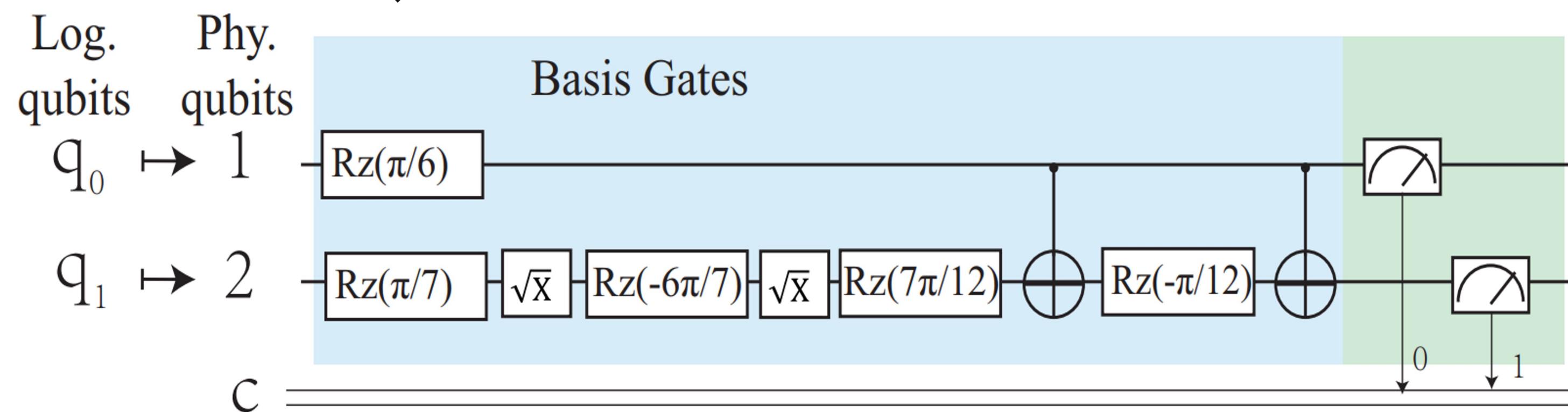
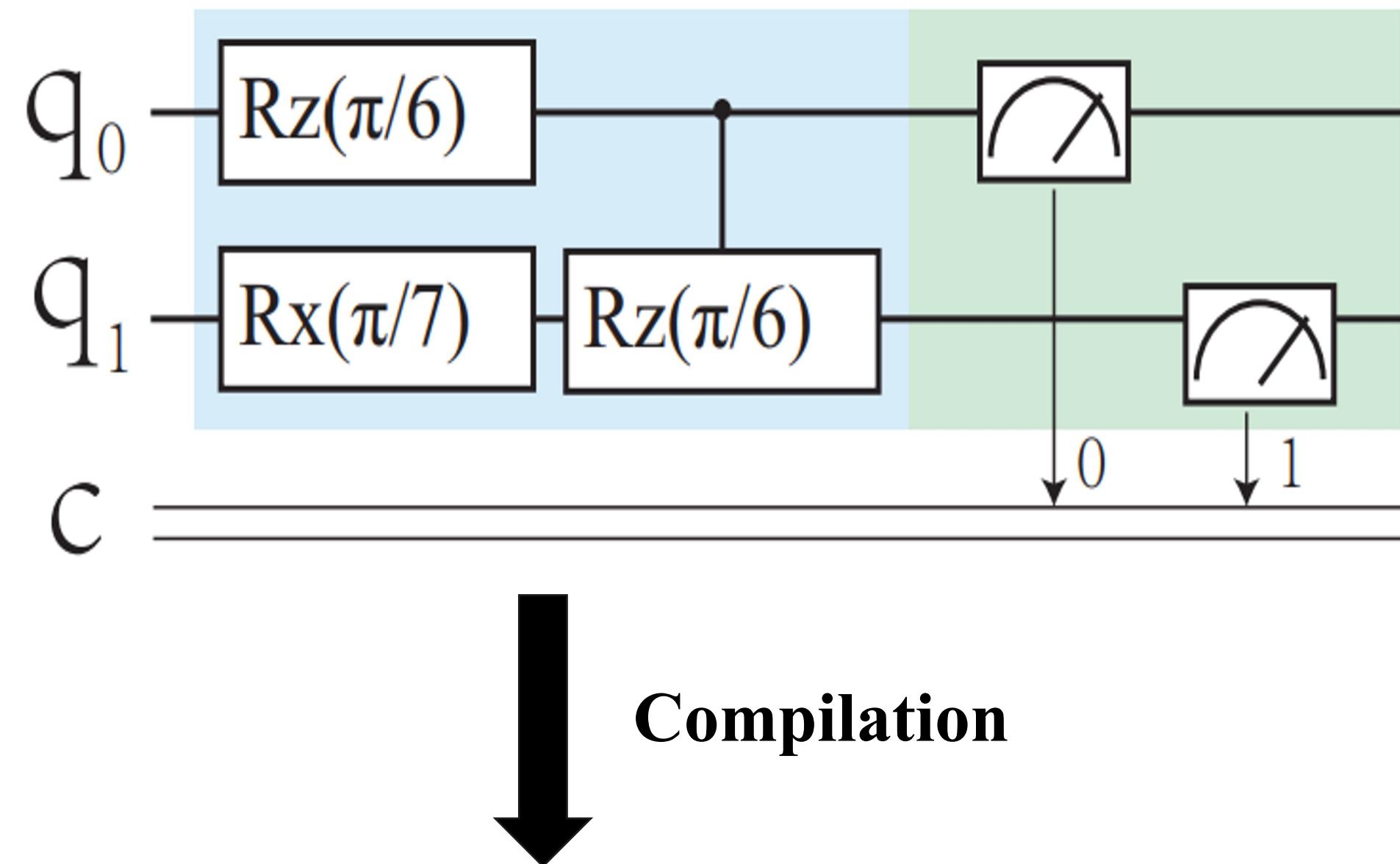


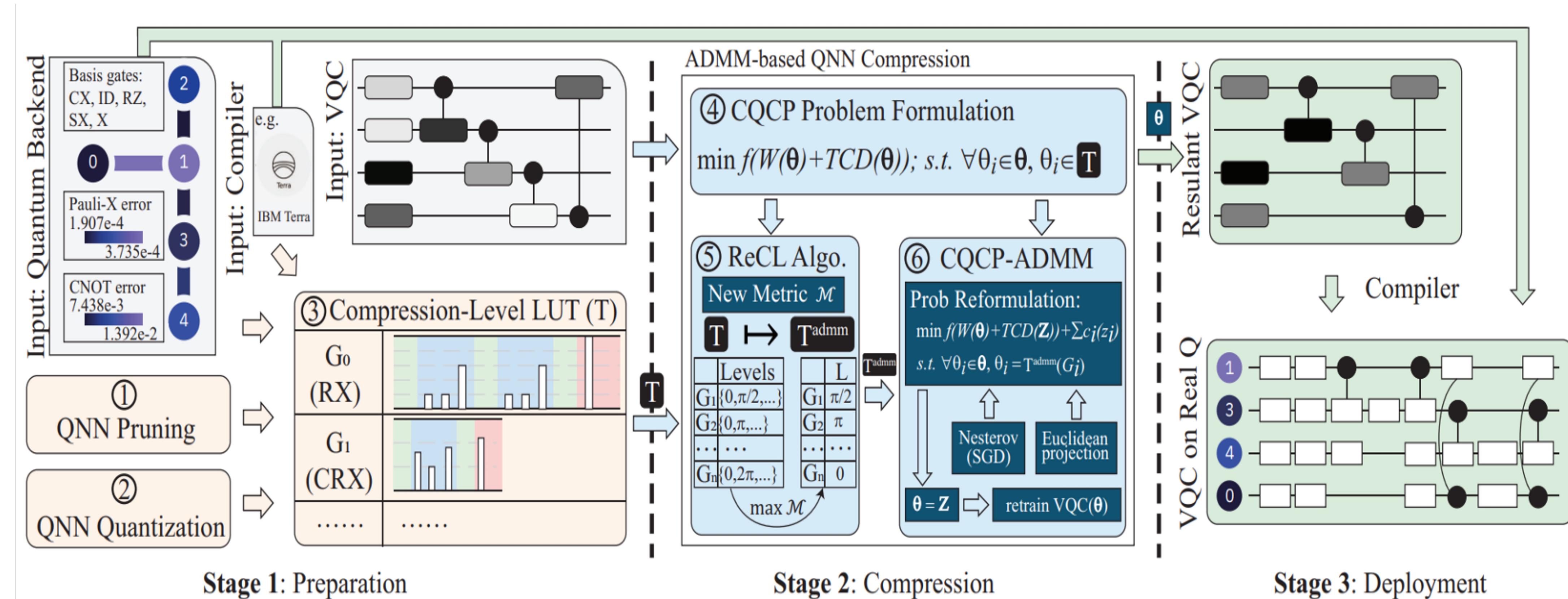
Table 1: circuit depth of compiled quantum gates on IBM quantum processors; parameters are in the range of  $[0, 4\pi]$

Gate	0	$\pi$	$2\pi$	$3\pi$	$4\pi$	$\pi/2$	$3\pi/2$	$5\pi/2$	$7\pi/2$	others
RX	0	1	0	1	0	1	3	1	3	5
RY	0	2	0	2	0	3	3	3	3	4
CRX	0	8	5	9	0	11	11	11	11	11
CRY	0	8	6	8	0	10	10	10	10	10

# CompVQC

- General Overview

Three stages: 1. Preparation; 2. Compression; 3. Deployment

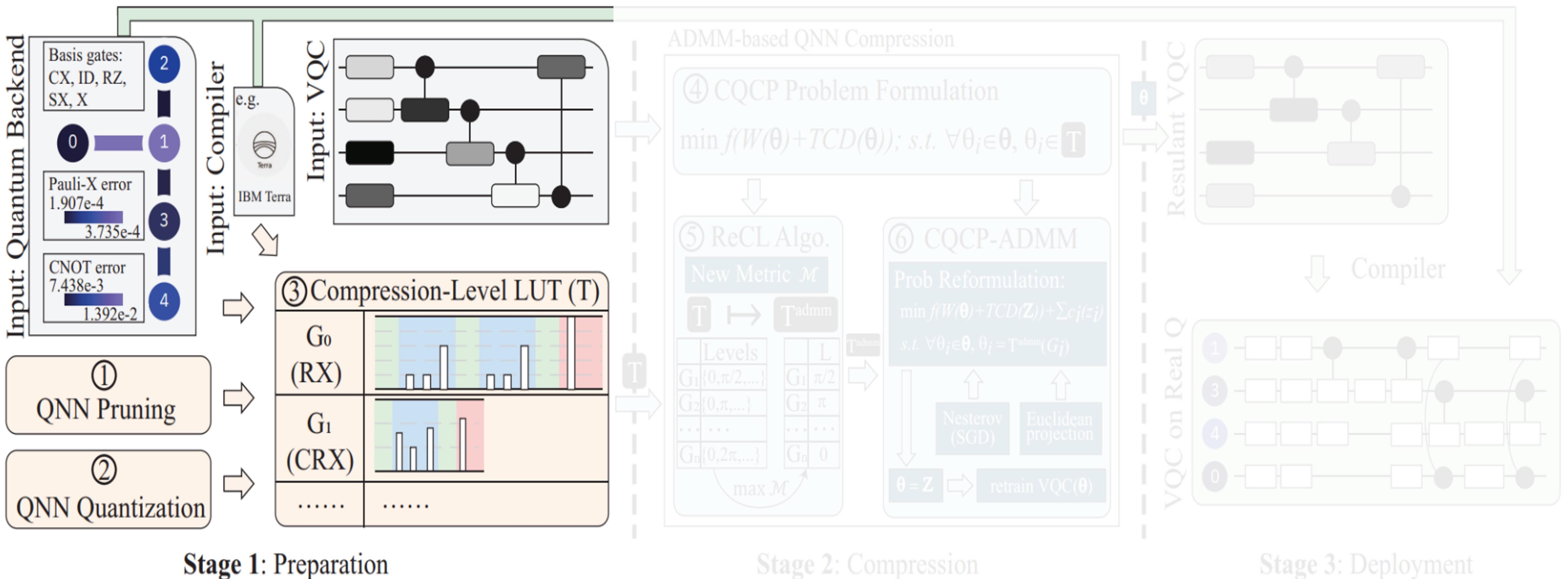


# CompVQC

- LUT Construction and Training a Quantum Model
- Reconstruct LUT for ADMM
- Compression based on ADMM
- Deployment

# CompVQC

- LUT Construction and Training a Quantum Model

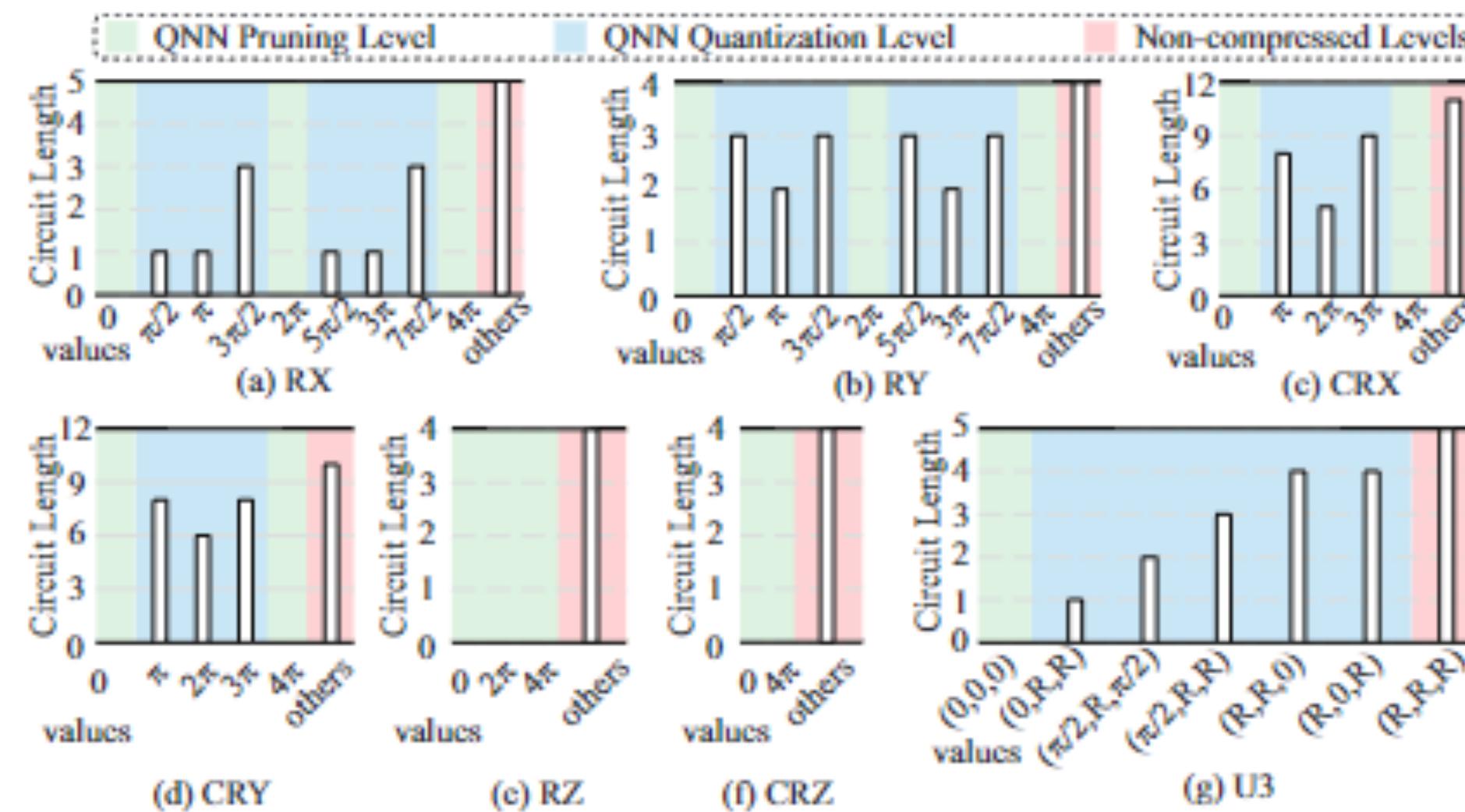


# CompVQC

- LUT Construction and Training a Quantum Model

- **Compression-Level Lookup Table (LUT)**

A combination of pruning/quantization level called as “compression level”.



**Table 1: circuit depth of compiled quantum gates on IBM quantum processors; parameters are in the range of  $[0, 4\pi]$**

Gate	0	$\pi$	$2\pi$	$3\pi$	$4\pi$	$\pi/2$	$3\pi/2$	$5\pi/2$	$7\pi/2$	others
RX	0	1	0	1	0	1	3	1	3	5
RY	0	2	0	2	0	3	3	3	3	4
CRX	0	8	5	9	0	11	11	11	11	11
CRY	0	8	6	8	0	10	10	10	10	10
RZ	0	1	0	1	0	1	3	1	3	5
CRZ	0	1	0	1	0	1	3	1	3	5
U3	0	1	0	1	0	1	3	1	3	5

- **VQC Pre-Training**

A VQC model is pre-trained for compression and the training process is implemented with **Torch Quantum**.

# Hands-On Tutorial (1) : LUT Construction

- **Input**

- Fixing points list
- Logical Gates List to be used
- Quantum Backend

- **Do**

- Get the compiler for the backend
- Get the compiled circuit length of each Logical Gate at each special fixing points

```
#Input
test_fixing_points = [math.pi*4, math.pi*2, math.pi, math.pi*3, math.pi/2,
                      math.pi/2*5, math.pi/2*7, math.pi/2*3, math.pi/6]
logical_gates = ['rx', 'ry', 'rz', 'crx', 'cry', 'crz']
backend = FakeValencia()

#api
df = LUT_construction(test_fixing_points, logical_gates, backend)
```

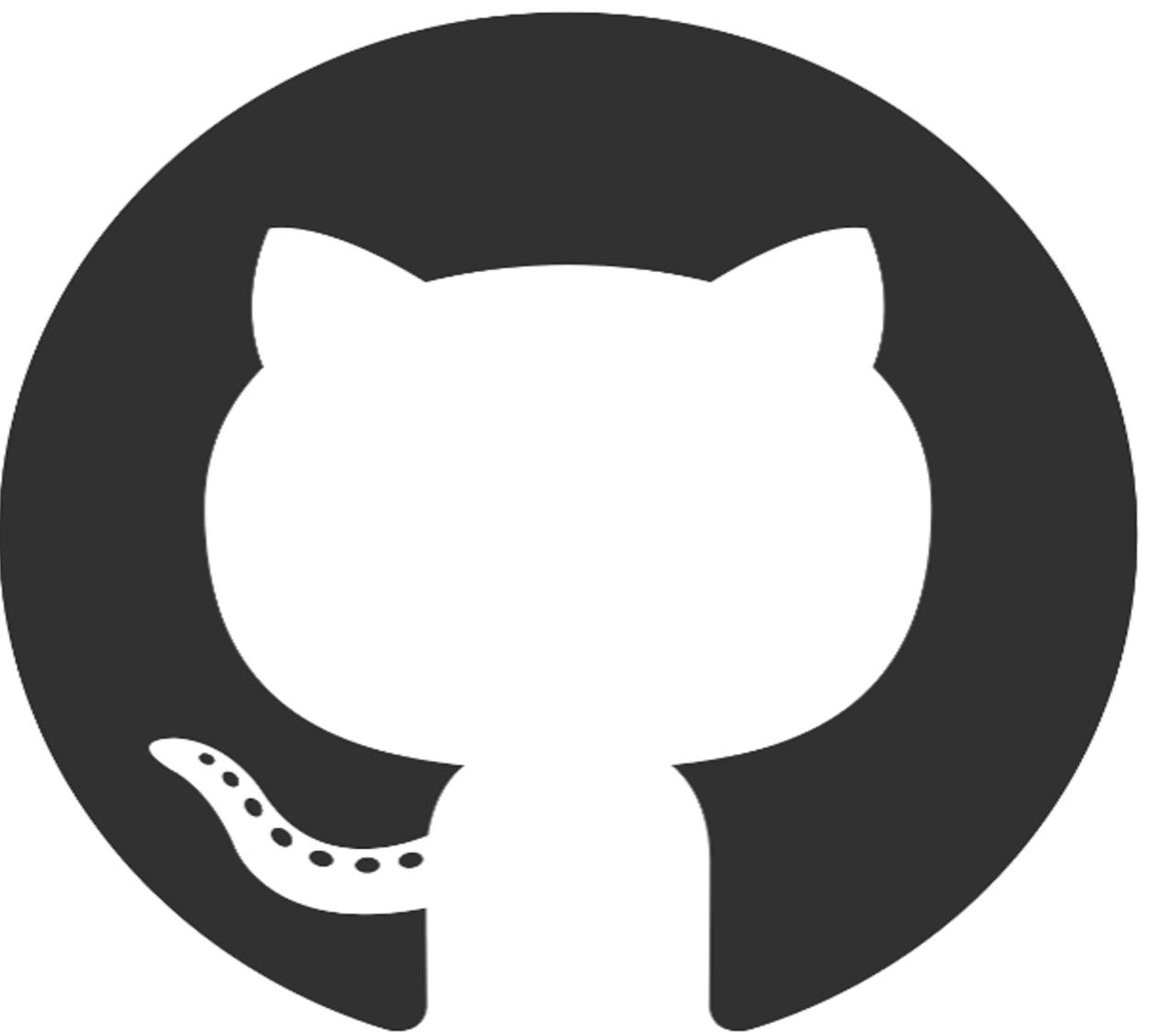
- **Output**

- Compression-Level Lookup Table (LUT)

fixing_points	rx	ry	rz	crx	cry	crz
12.57	0	0	0	0	0	0
6.28	0	0	0	5	6	4
3.14	1	2	1	8	8	4
9.42	1	2	1	9	8	4
1.57	1	3	1	11	10	4
7.85	1	3	1	11	10	4
11.00	3	3	1	11	10	4
4.71	3	3	1	11	10	4
0.52	5	4	1	11	10	4

# Hands-On Tutorial (1)

## *LUT Construction*



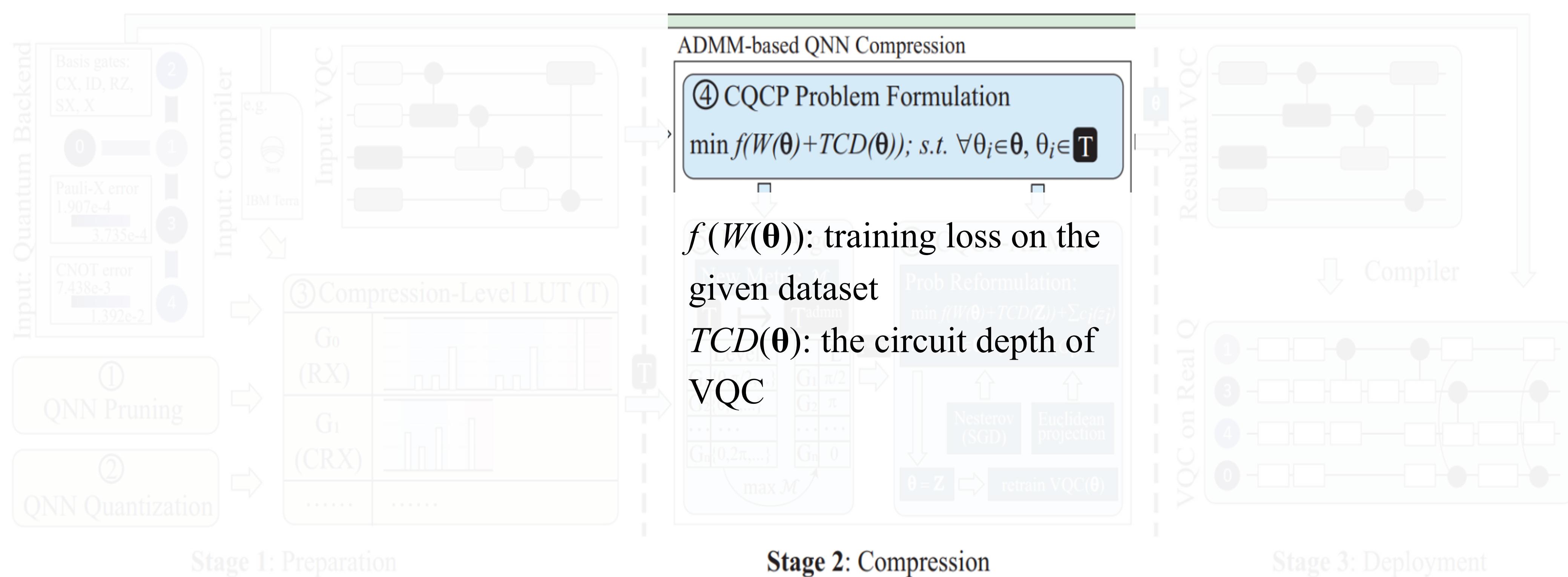
# CompVQC

- LUT Construction and Training a Quantum Model
- Reconstruct LUT for ADMM
- Compression based on ADMM
- Deployment

# CompVQC

- Problem Definition

Given VQC  $W(\theta)$ , LUT  $T$ , quantum compiler  $C$ , the problem is to determine trainable parameters  $\theta$ , such that:



# CompVQC

- Reconstruction LUT for ADMM

Process is conducted by traversing all quantum gates in VQC and **select the compression target with highest metric.**

A heuristic metric for the choice

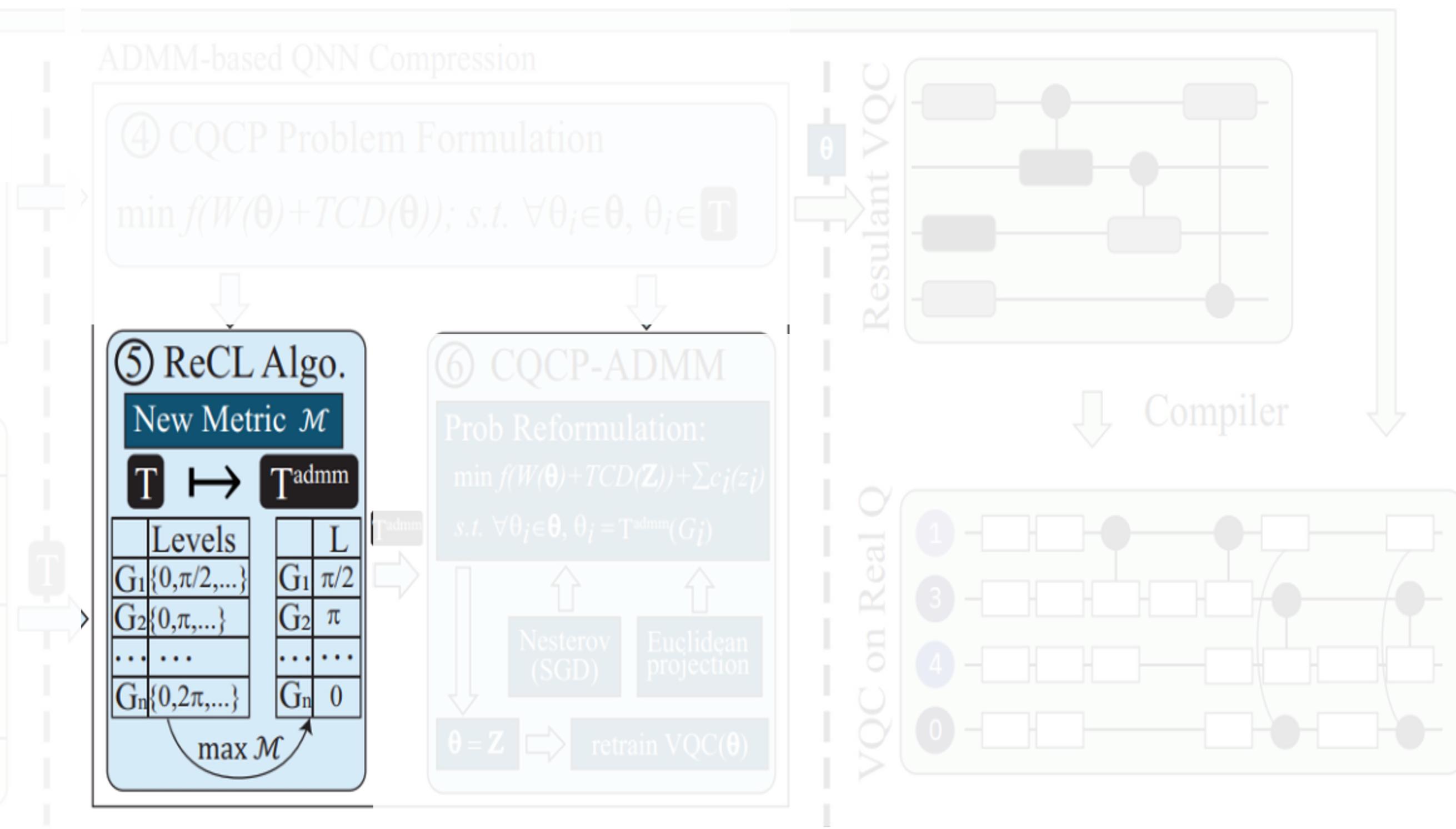
$$\mathcal{M}(\theta, G_i(\gamma_{i,k})) = acc(W(\theta^{i,k})) \cdot \tau(\theta^{i,k}, \theta)$$

$$\tau(\theta^{i,k}, \theta) = \frac{TCD(\theta)}{TCD(\theta^{i,k})}$$

$acc(W(\theta^{i,k}))$ : the accuracy of the VQC under new parameters

$TCD(\theta)$ : the inverse of the compression ratio by changing parameters from  $\theta$  to  $\theta^{i,k}$

Stage 1: Preparation



Stage 2: Compression

Stage 3: Deployment

# Hands-On Tutorial (2) : Reconstruct LUT for ADMM

- **Input**

- A trained model
- Original LUT
- The metrics function of accuracy and length

```
#input
model = torch.load('model.pth')
lut = pd.read_csv('lut.csv')
def metrics_func(acc, depth):
    return acc+1.0/depth
backend = FakeValencia()
```

- **For each parameter, Do**

- Replace it with points at compression level in original LUT while fixing other parameters
- Calculate the metrics of each new model
- Select the point with the highest metric as the compression level for ADMM

```
#api
new_lut = LUT_reconstruction(model, lut, backend, metrics_func)
```

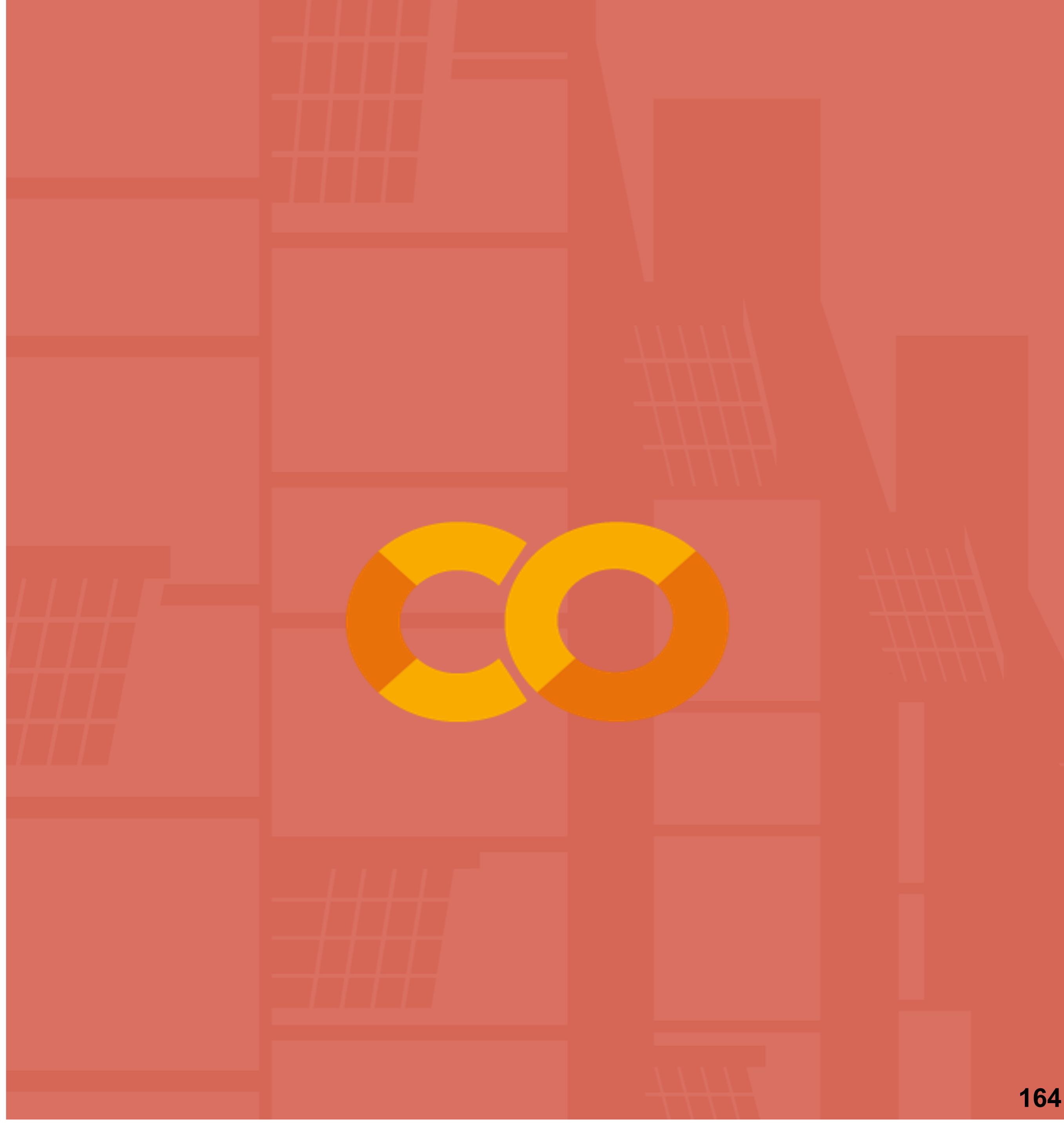
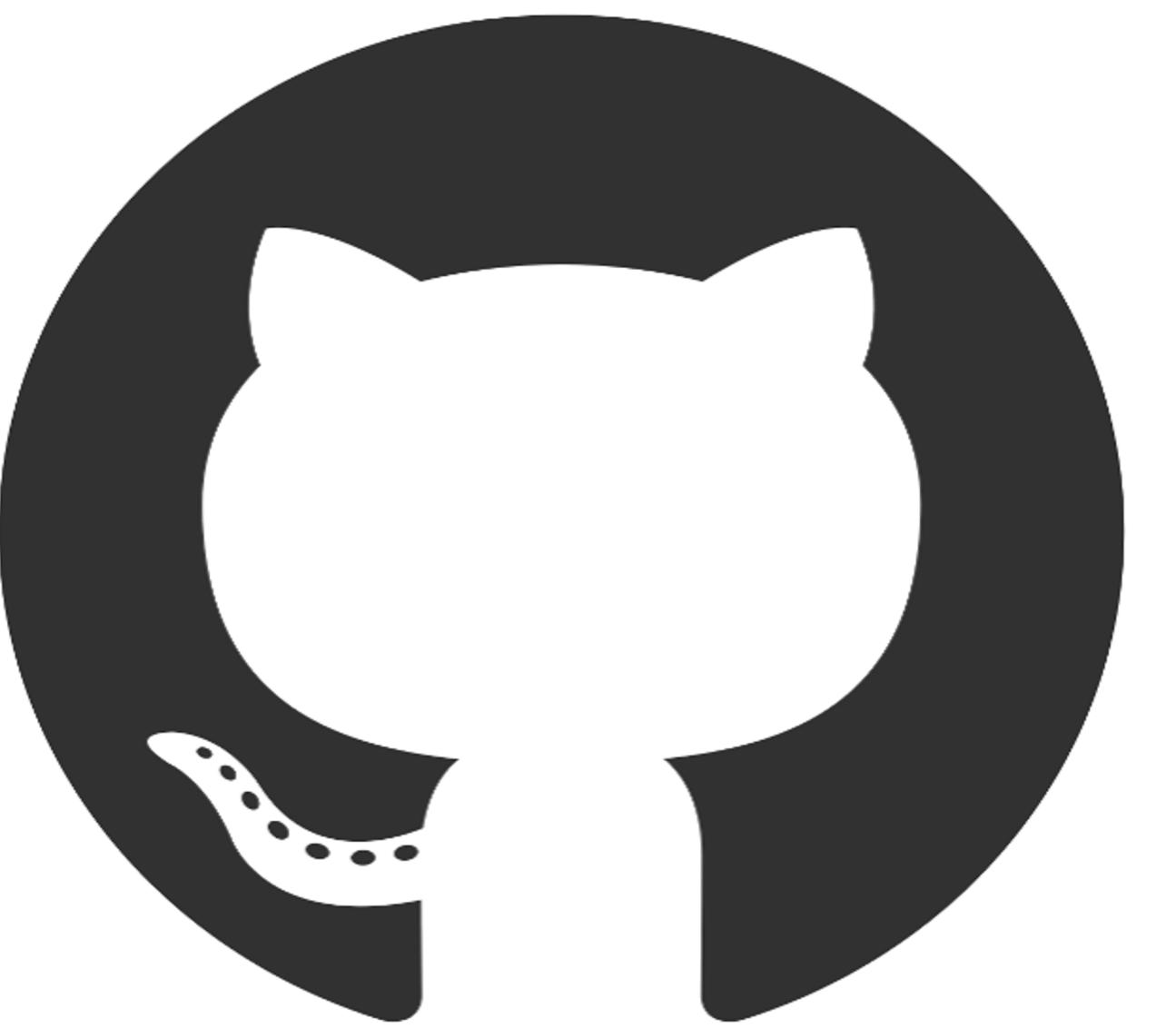
- **Output**

- A new LUT for ADMM

```
[ 1.57  6.28 11.     6.28  4.71  1.57  3.14  9.42  6.28 12.57 12.57  9.42
   7.85 12.57  9.42  1.57  11.     9.42]
```

# Hands-On Tutorial (2)

## *Reconstruct LUT for ADMM*



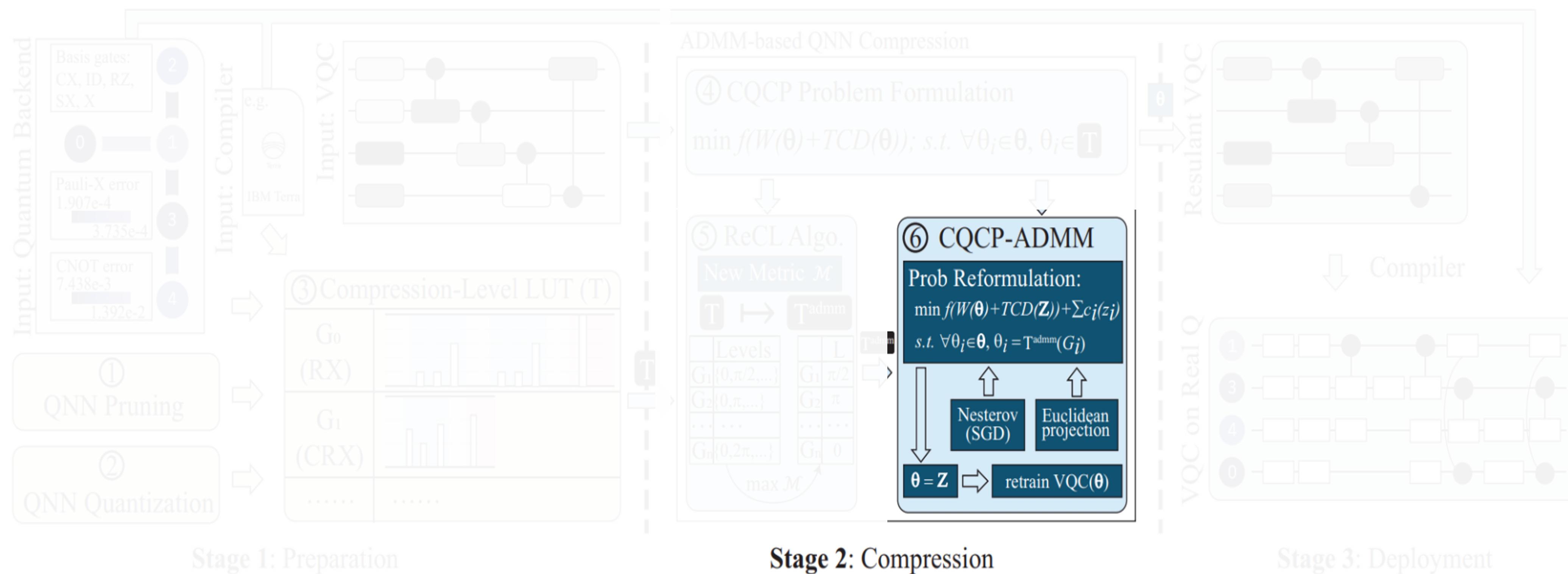
# CompVQC

- LUT Construction and Training a Quantum Model
- Reconstruct LUT for ADMM
- **Compression based on ADMM**
- Deployment

# CompVQC

- Compression based on ADMM

Each parameter can either be compressed to the target value in  $T^{\text{admm}}$  or not compressed.



# CompVQC

- Compression based on ADMM

Given reconstructed compression-level LUT  $T^{admm}$ , the CQCP is formulated as:

$$\begin{aligned} \min_{\{\theta_i\}} \quad & f(W(\theta)) + TCD(Z) + \sum_{\forall z_i \in Z} c_i(z_i), \\ \text{s.t.} \quad & \forall \theta_i \in \theta, \quad \theta_i = T^{admm}(G_i). \end{aligned}$$

$Z$ : a set of auxiliary variables for subproblem decomposition and  $z_i \in \mathbf{Z}$  is corresponding to  $\theta_i \in \theta$

$f(W(\theta)) + TCD(Z)$  : the objective function in the original CQCP problem(previously seen).

$$c_i(z_i) = \begin{cases} 0 & \text{if } \theta_i \in T^{s,r}(G_i), T^{s,r} = T^{admm} \odot mask^r \\ +\infty & \text{if otherwise.} \end{cases}$$

$c_i(z_i)$ : An indicator function to serve as a penalty term

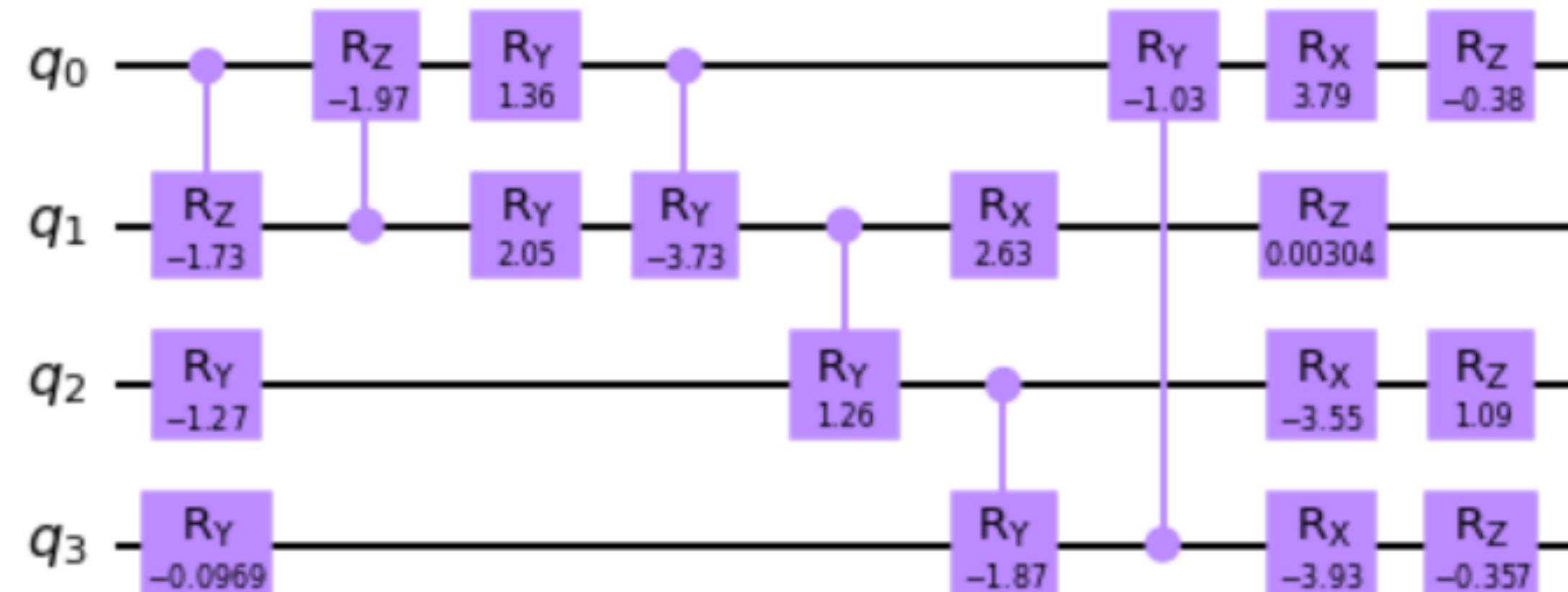
$mask^r$ : variable to indicate whether the parameters will be compressed at iteration  $r$ .

# Hands-On Tutorial (3) : Compression based on ADMM

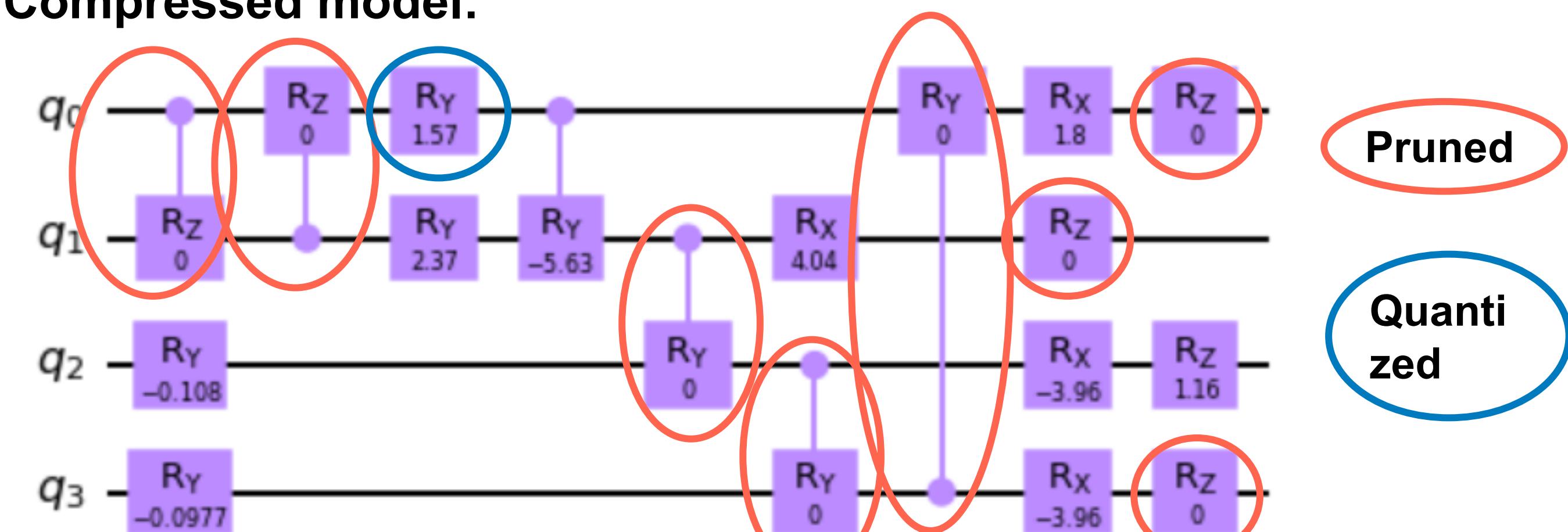
- **Input**
  - A trained model
  - A new LUT for ADMM
- **Do**
  - Compress a model with ADMM
  - Fine-tune the compressed model
- **Output**
  - A compressed model

	Circuit Length	Accuracy
Original model	51	95.6%
Compressed	21	96.60%

- **Original model:**

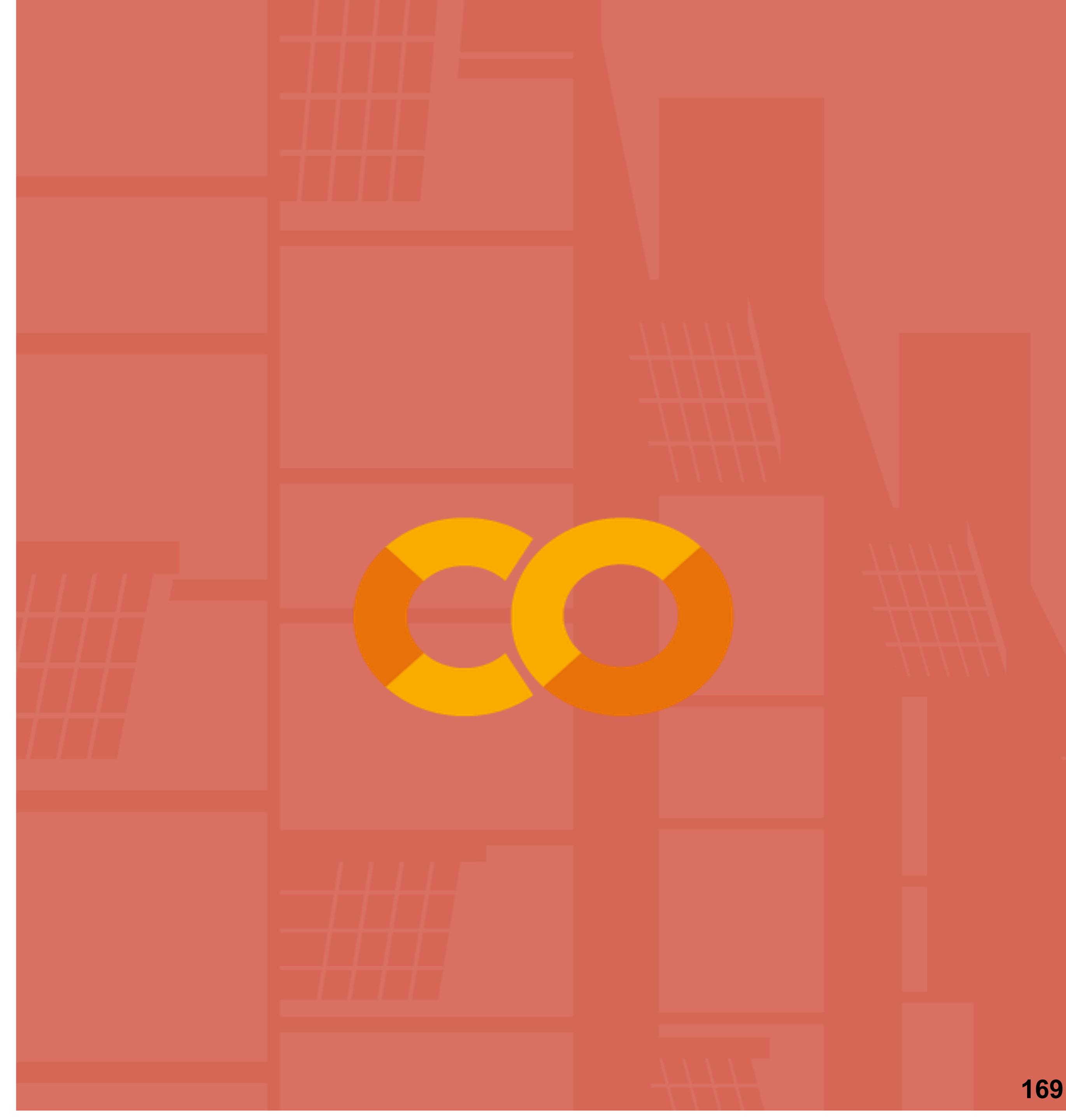


- **Compressed model:**



# Hands-On Tutorial (3)

## *Compression based on ADMM*

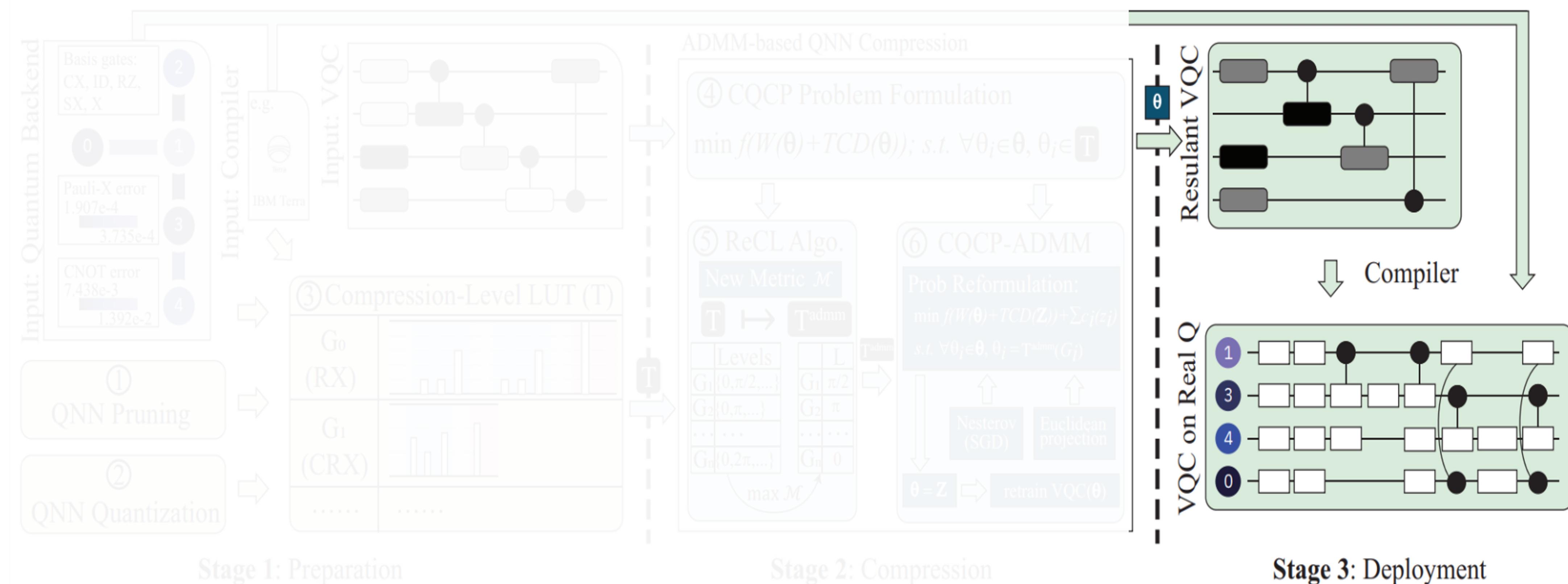


# CompVQC

- LUT Construction and Training a Quantum Model
- Reconstruct LUT for ADMM
- Compression based on ADMM
- Deployment

# CompVQC

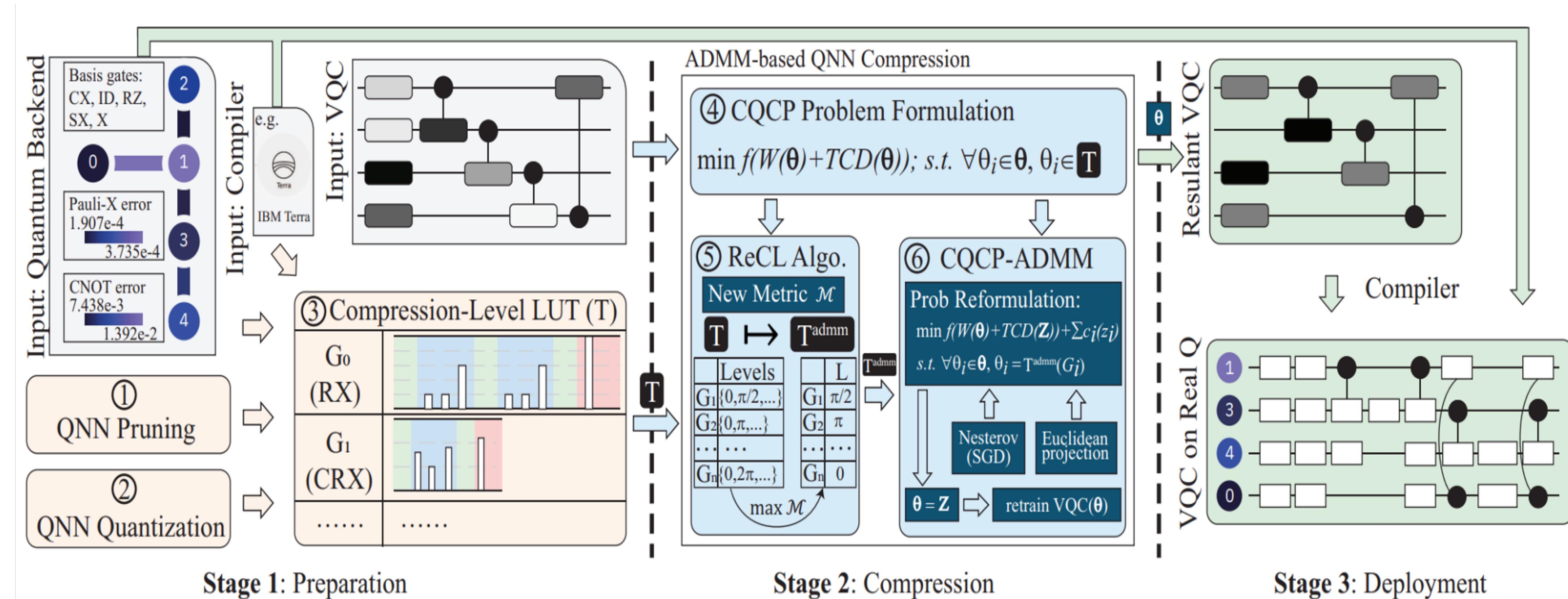
- Deployment



# CompVQC

- General Overview

Three stages: 1. Preparation; 2. Compression; 3. Deployment



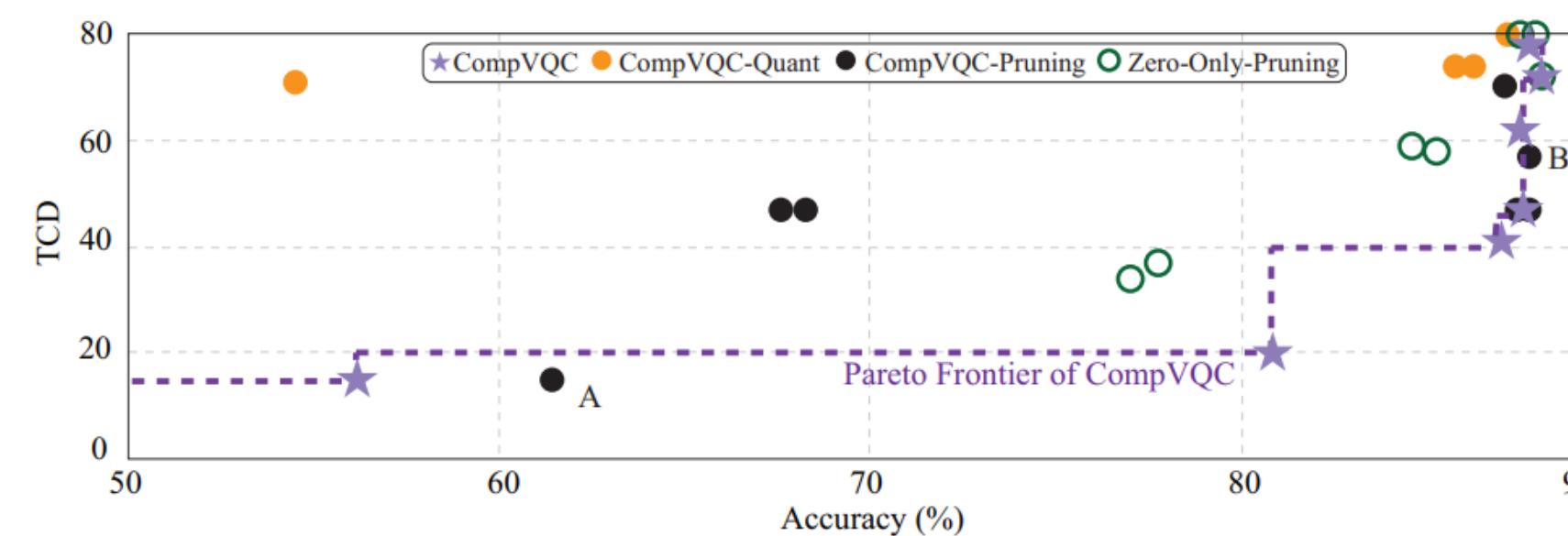
# Experimental Results

- Simulation Results on ML Dataset

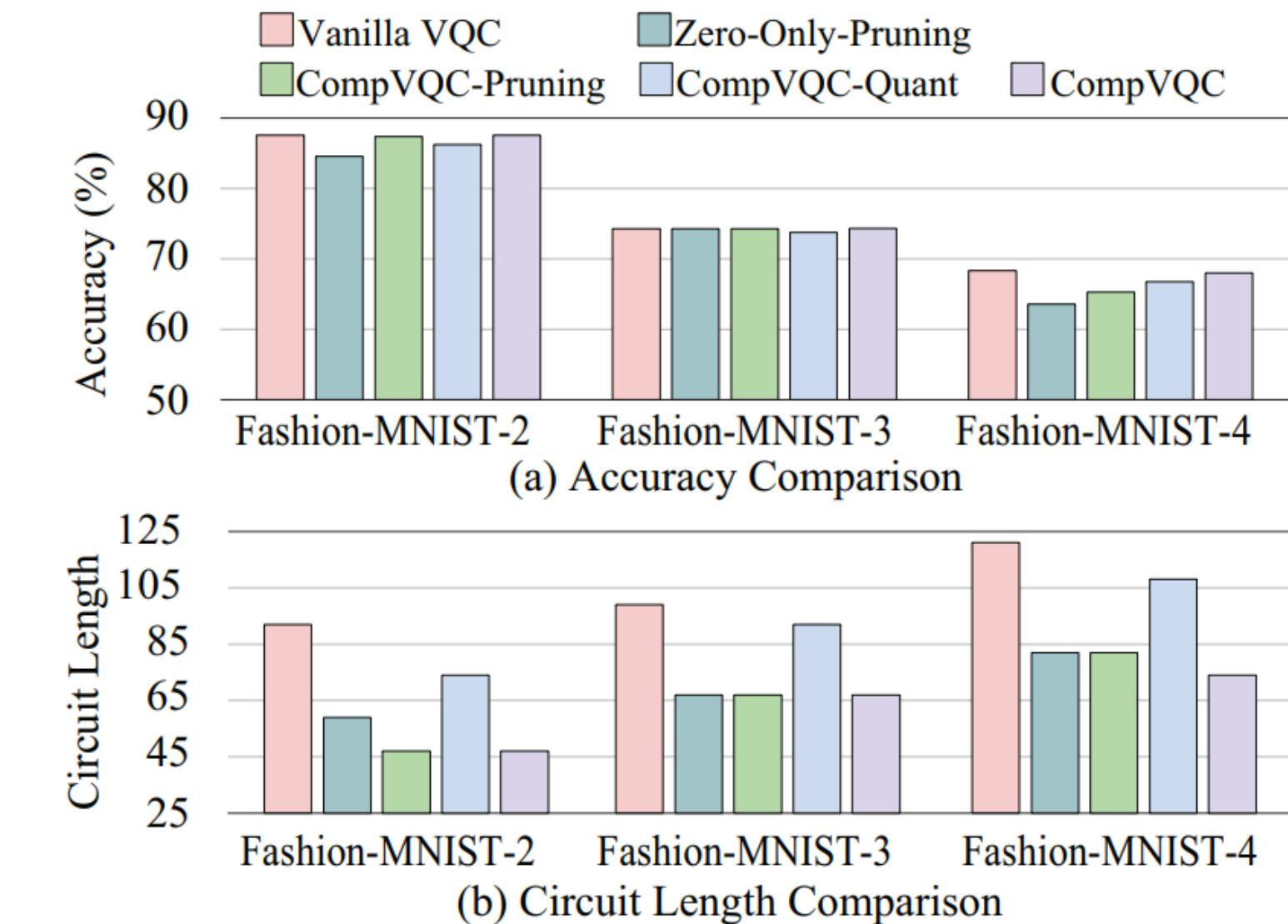
CompVQC can maintain high accuracy with **<1% accuracy loss**. And the reduction of circuit length is up to **2.5X**.

**Table 2: Comparison among different methods on the accuracy performance and the TCD of the VQC**

Compression Method	MNIST-2		Fashion-MNIST-2	
	Acc. (vs. Baseline)	TCD (Speedup)	Acc. (vs. Baseline)	TCD (Speedup)
Vanilla VQC	82.74%(0)	121(0)	87.58%(0)	92(0)
Zero-Only-Pruning	80.58%(-2.16%)	70(1.73×)	86.92%(-0.67%)	63(1.46×)
CompVQC-Pruning	81.83%(-0.91%)	74(1.64 ×)	87.41%(-0.17%)	47(1.96×)
CompVQC-Quant	80.99%(-1.75%)	108(1.10×)	86.25%(-1.33%)	74(1.24×)
CompVQC	<b>81.83%(-0.91%)</b>	<b>47(2.57×)</b>	<b>87.58%(-0.00%)</b>	<b>47(1.96×)</b>



**Figure 5: Main results: The Accuracy-Circuit Depth Tradeoff on Fashion-MNIST2**



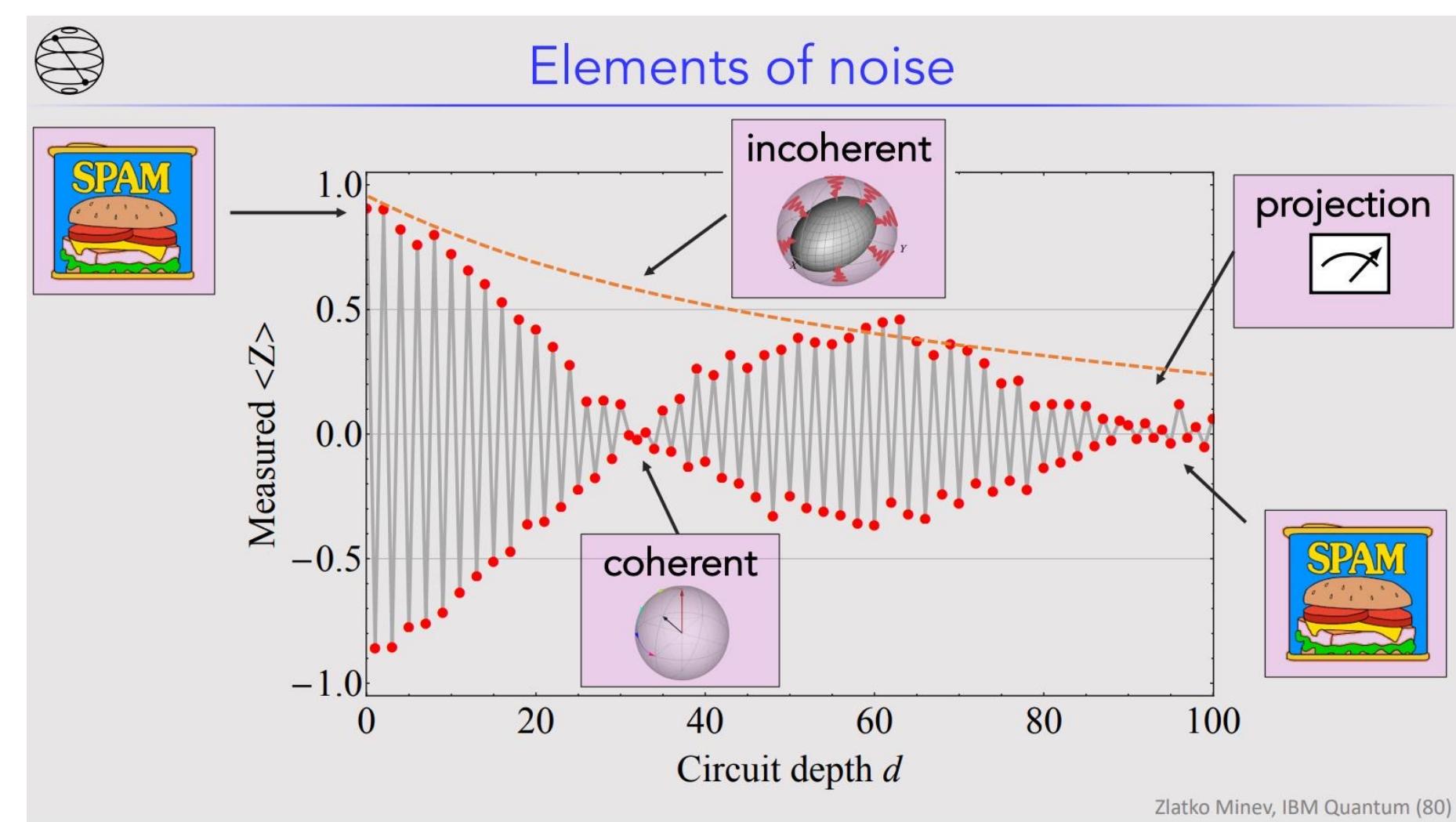
**Figure 6: Main Results: CompVQC Scalability on Fashion-MNIST with 2-4 class**

# Experimental Results

- Results on Multiple IBM Quantum Computers

CompVQC can reduce circuit length by 2x while the accuracy is also higher in a noisy environment

Datasets	Syn-Dataset-4		Syn-Dataset-16		
Compression Method	Acc. (vs. Baseline)	TCD (Speedup)	Acc. (vs. Baseline)	TCD (Speedup)	
Qiskit Aer	Vanilla VQC	94%(0)	23(0)	96%(0)	51(0)
	Comp-VQC	99%(5%)	11(2.09×)	98%(2%)	23(2.22×)
IBM Q	Vanilla VQC	79%(-15%)	23(1.00×)	86%(-10%)	51(1.00×)
	CompVQC	99%(5%)	11(2.09×)	98%(2%)	23(2.22×)
Acc.(vs. Baseline)	ibm_lagos	ibm_perth	ibm_jakarta		
Vanilla VQC(TCD=23)	79%(0)	86%(0)	92%(0)		
CompVQC(TCD=11)	99%(20%)	98%(12%)	100%(8%)		



Circuit compression can make the QNN model more robust to the noise

# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations 

1.3 TQ for State Prep 

1.4 TQ for VQE 

1.4 TQ for QNN 

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning 

2.2 QuantumNAT: Noise Injection and Quantization 

2.3 QOC: On-Chip Training 

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression 

## Section 3

### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control 

3.2 Variational Pulse Learning 

# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations 

1.3 TQ for State Prep 

1.4 TQ for VQE 

1.4 TQ for QNN 

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning 

2.2 QuantumNAT: Noise Injection and Quantization 

2.3 QOC: On-Chip Training 

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression 

## Section 3

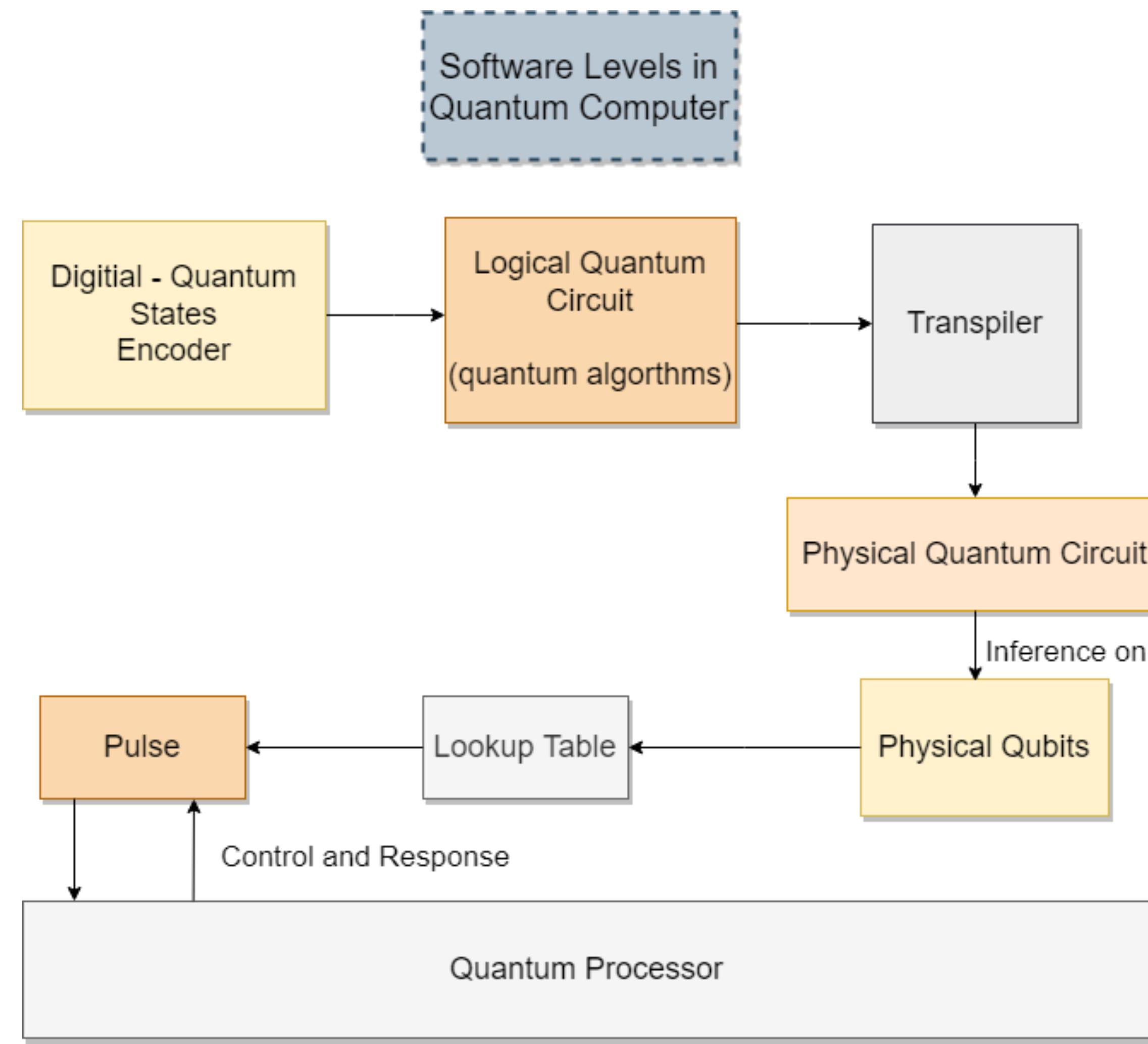
### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control 

3.2 Variational Pulse Learning 

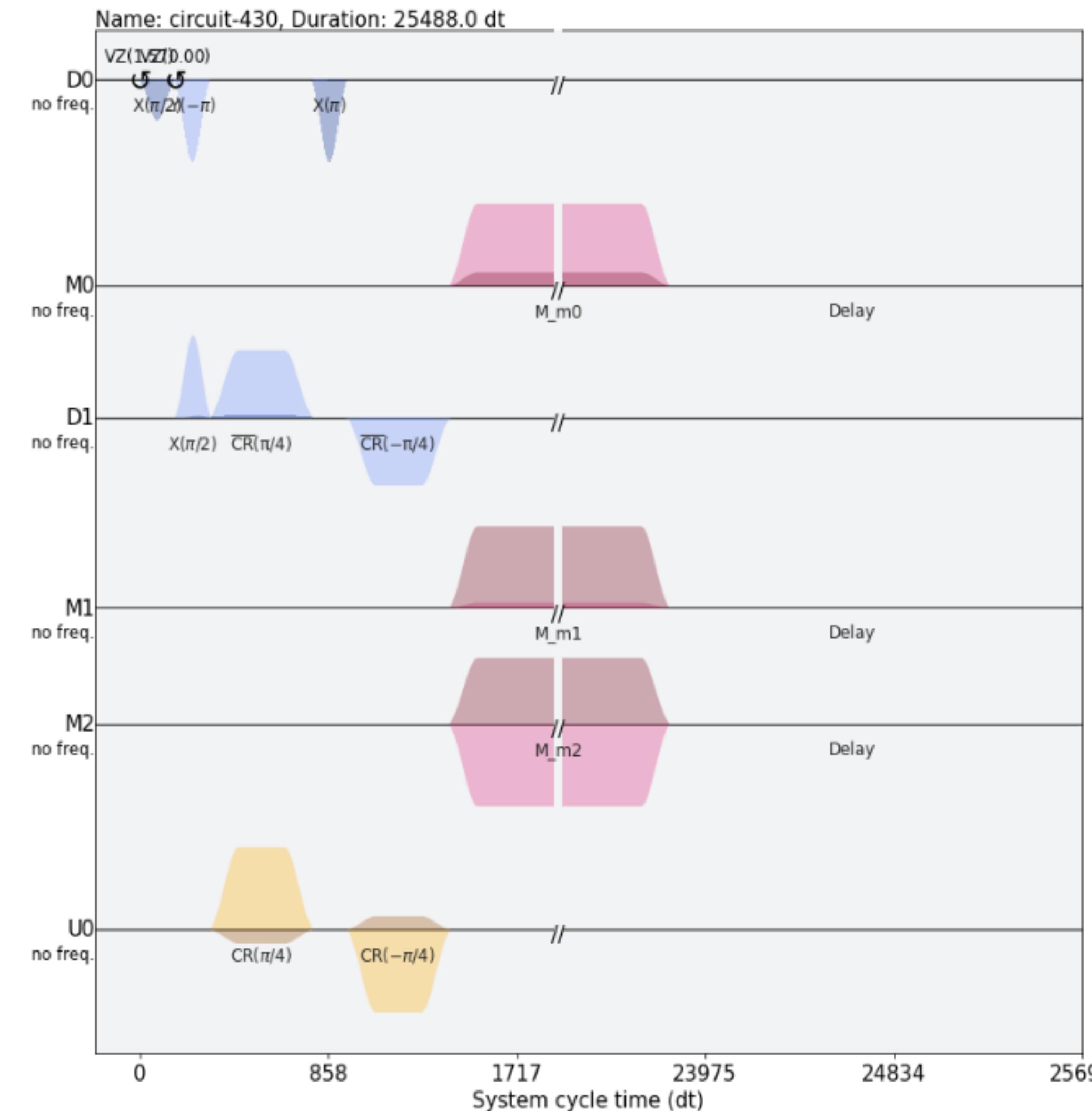
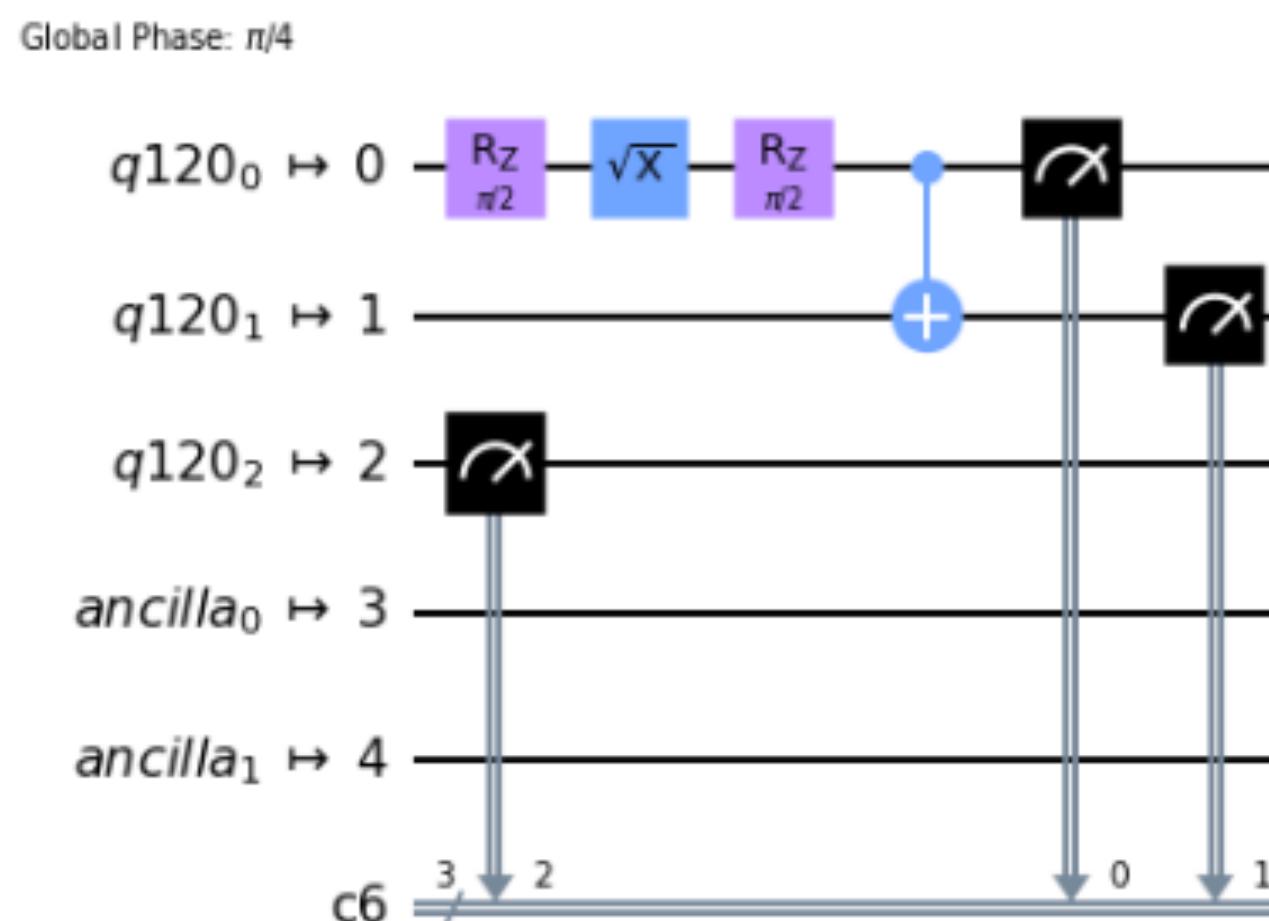
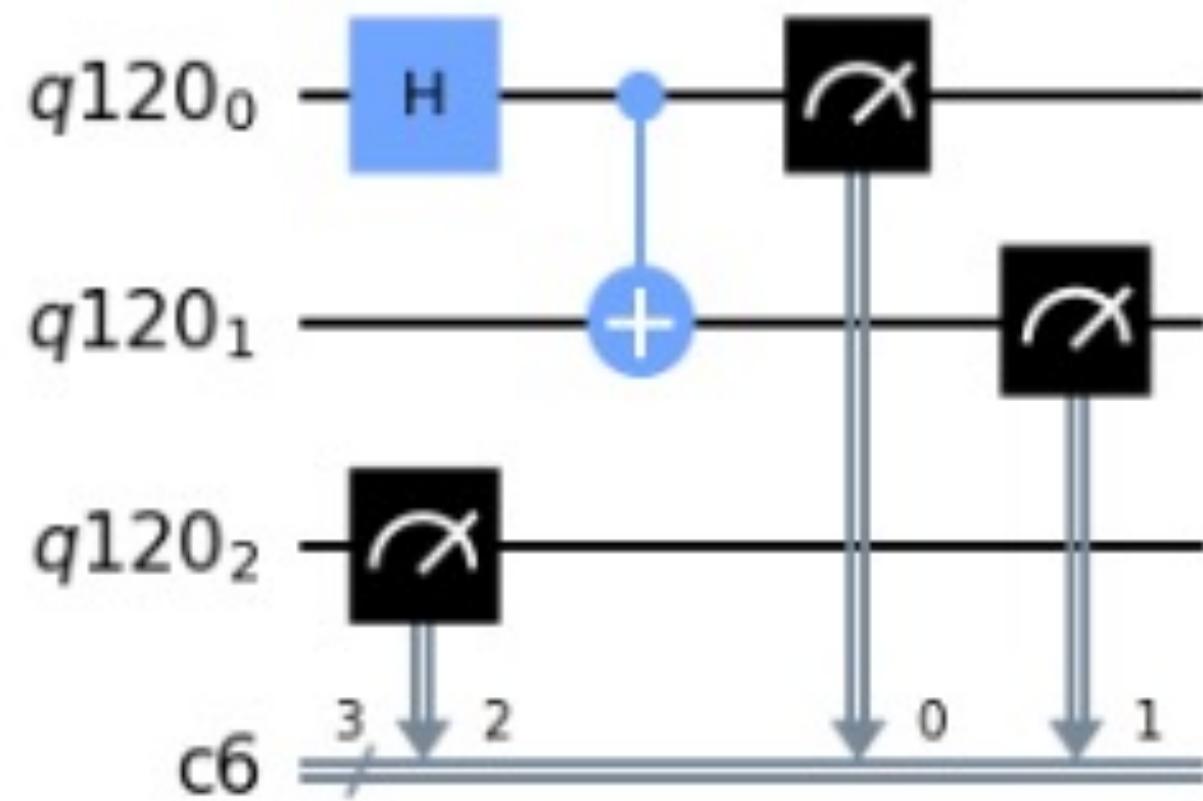
# Brief Overview of Workflow

- When we want to execute a quantum program on a quantum computer, we need to compile it first:



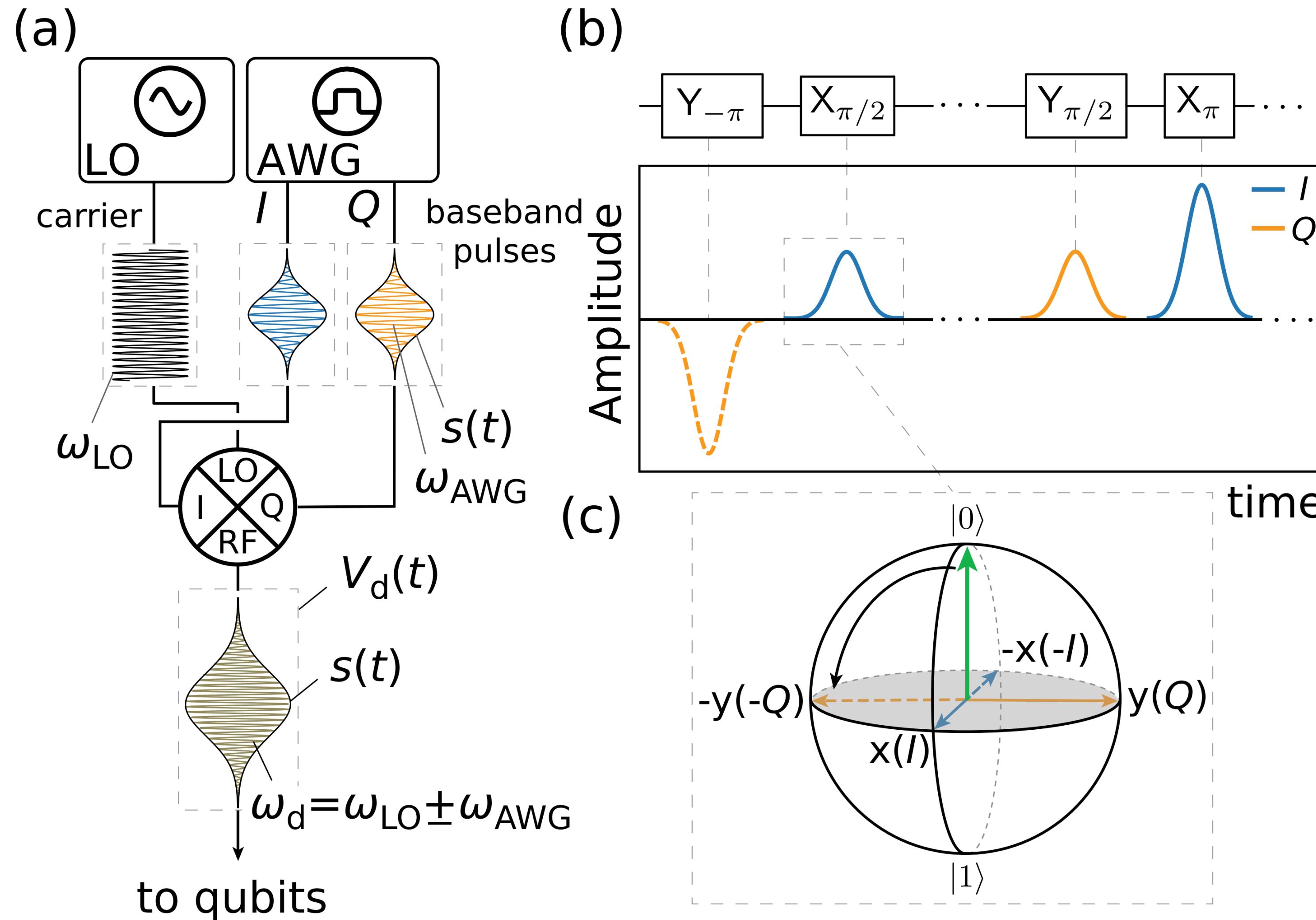
# Pulse-Based and Gate-Based

- Gate- and Pulse-level are two different abstraction layers.



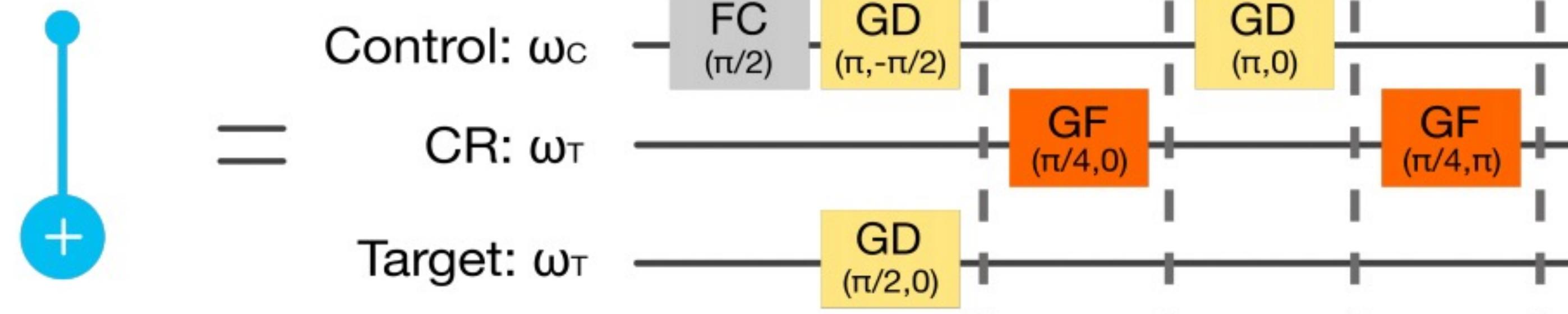
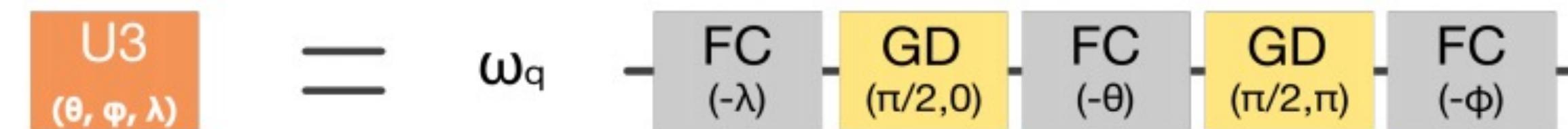
# Pulse-Based and Gate-Based

- The microwave is used to control the quantum bits.



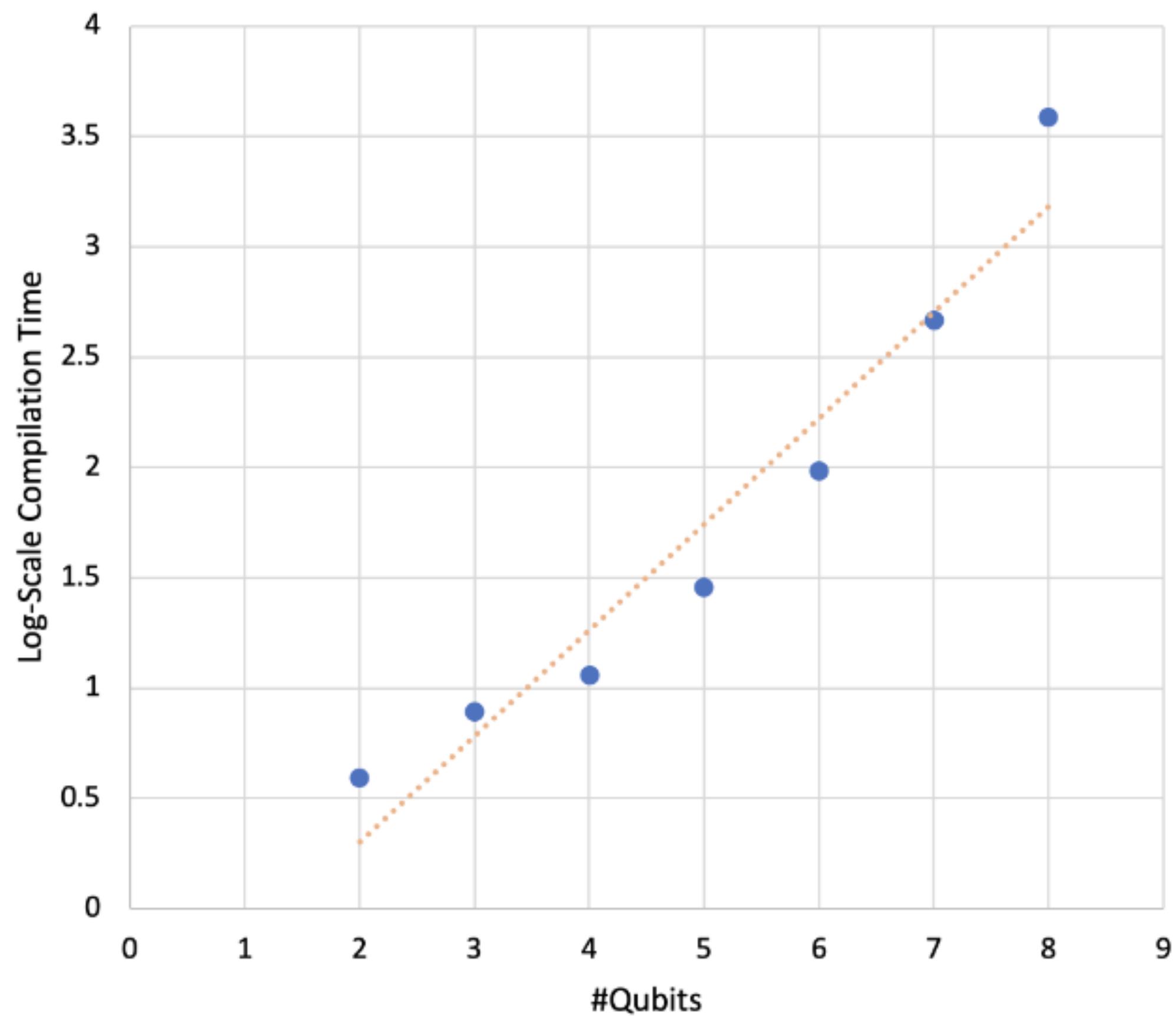
# Gate-based Compilation

- Each gate corresponds to a pulse. One-by-one concatenation of all pulses realize the function of many gates.



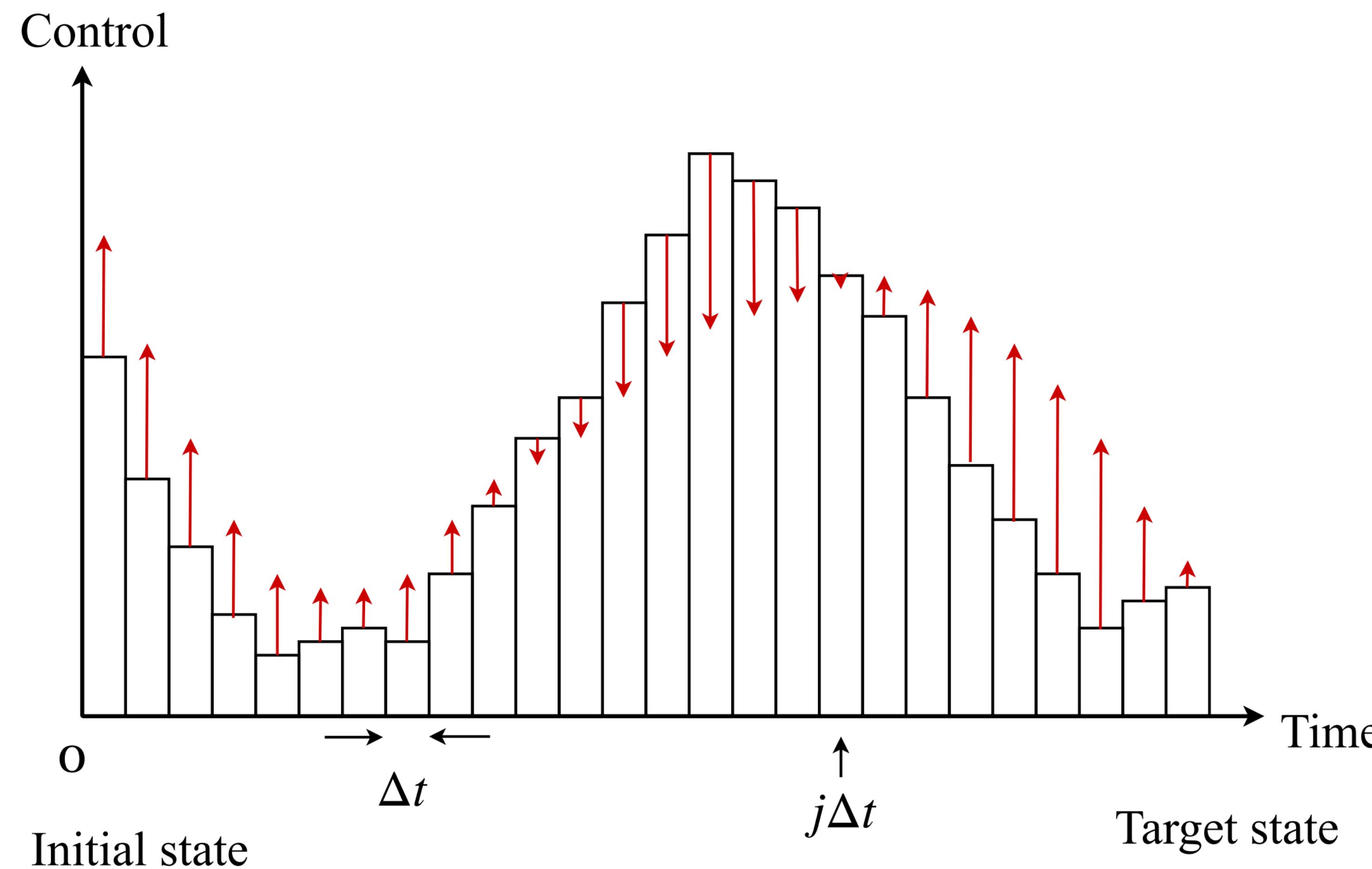
# Quantum Optimal Control(QOC)

- QOC is commonly used to generate pulses (from target unitary matrices to pulses)
- QOC is computationally expensive



# Quantum Optimal Control (QOC)

- As shown in the figure, the control pulse is divided into multiple time slots. The amplitude of each time slot will be adjusted through optimization.



# Accqoc: Group-based Pre-compilation

- Instead of dealing with single gate, we generate pulses from groups of gates.

---

## Algorithm 1 Bit Dividing

---

**Require:** qasm files, bit constraint(bc)

**Ensure:** large-groups

```
1: Initialize large-groups
2: for qasm in all qasm files do
3:   DAG = ToDAG(qasm)
4:   for node in DAG.topological-order: do
5:     if node can be grouped with both predecessor then
6:       Merge the groups the two predecessors are in
7:     else if node can be grouped with one predecessor
then
8:       Group the node with the predecessor
9:       Update large-groups
10:    else if node can be group with no predecessor then
11:      Put the node in a new group
12:      Update large-groups
13:    end if
14:   end for
15: end for
16: return large-groups
```

---

---

## Algorithm 2 Layer Dividing

---

**Require:** large-groups, layer constraint(lc)

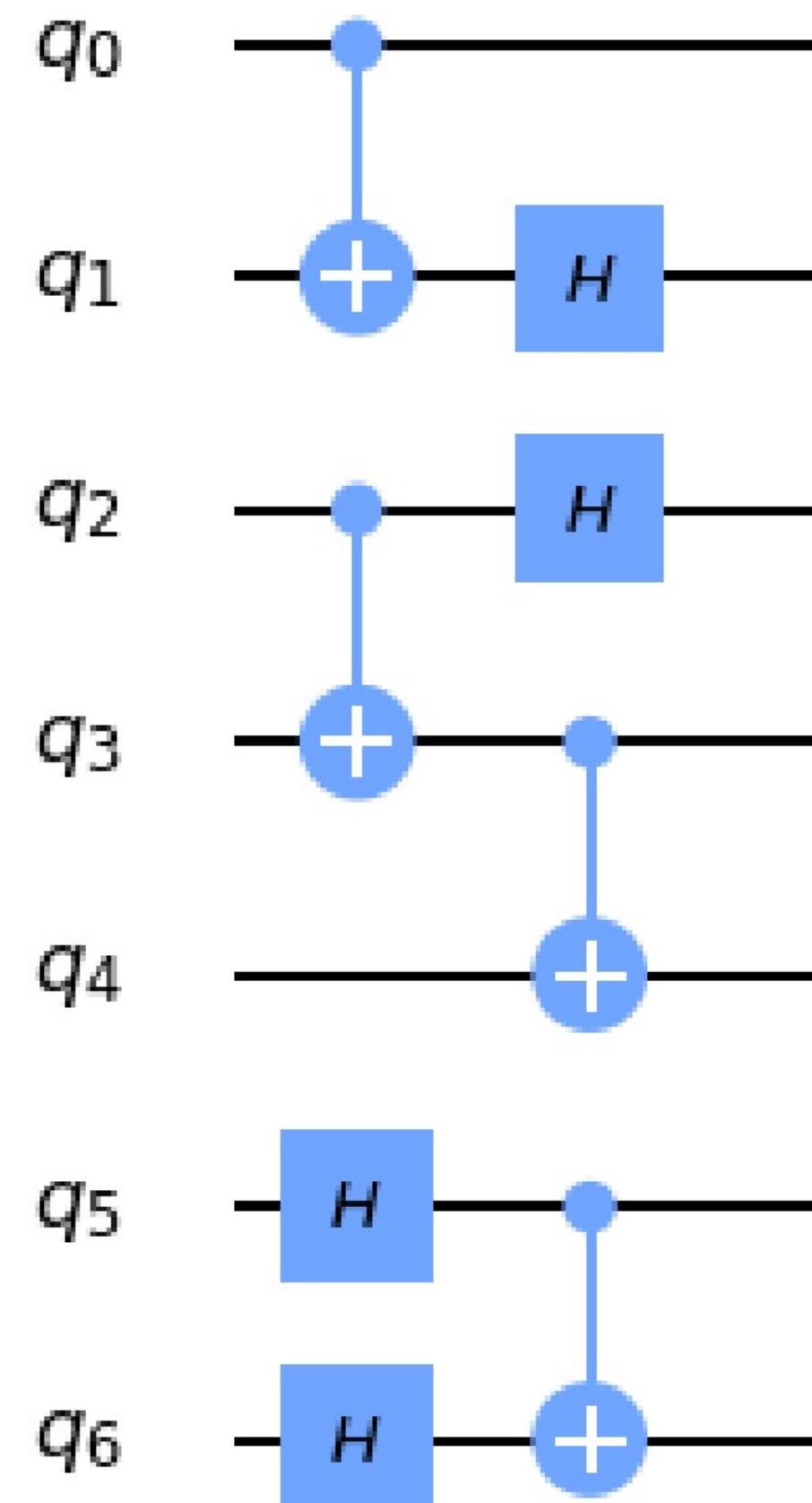
**Ensure:** group list

```
1: Initialize group-list
2: for node in DAG.topological-order: do
3:   Depth[node] = max(Depth[node's predecessor(s)]) + 1
4: end for
5: for subgroup in large-groups: do
6:   startDepth = depth of shallowest node
7:   layer = 0
8:   Initialize temp-group
9:   for node in subgroup: do
10:    diff = depth[node] - start
11:    if diff mod lc ≤ layer then
12:      Append node to temp-group
13:    else
14:      Append temp-group to group-list
15:      Clear temp-group
16:      Append node to temp-group
17:      layer += 1
18:    end if
19:   end for
20: end for
21: return group-list
```

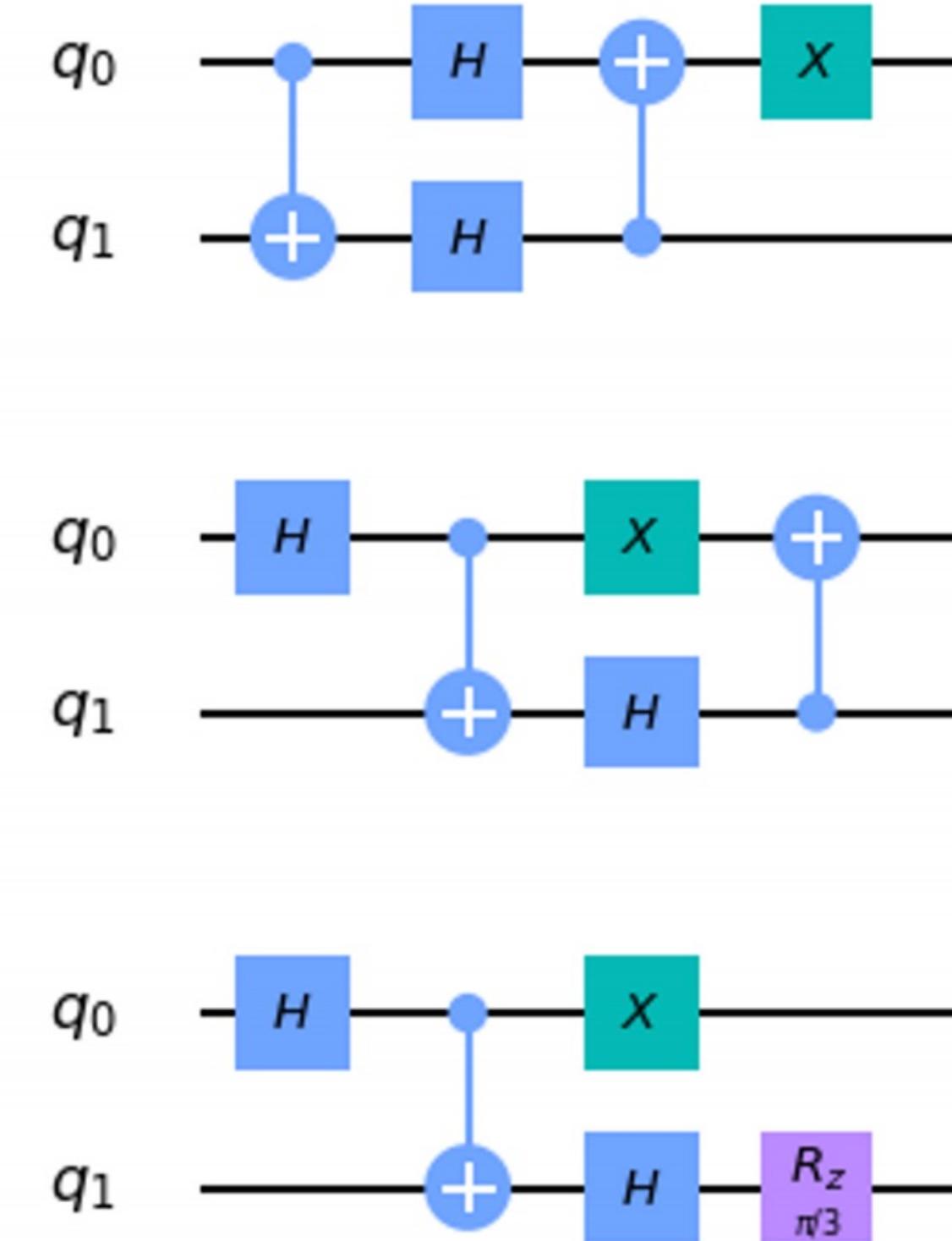
---

# Accqoc: Group-based Pre-compilation

- We limit the size of the groups that QOC takes. (Key Insight)



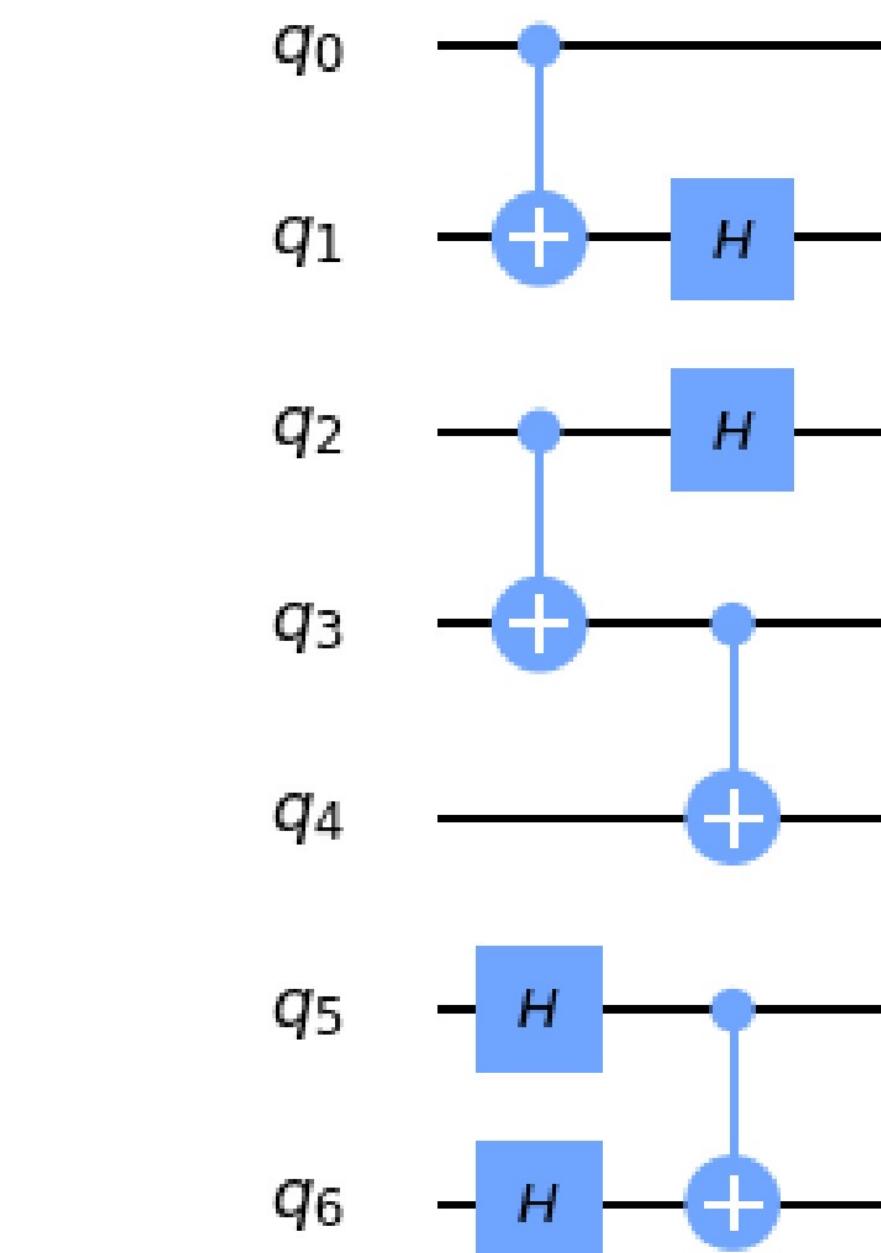
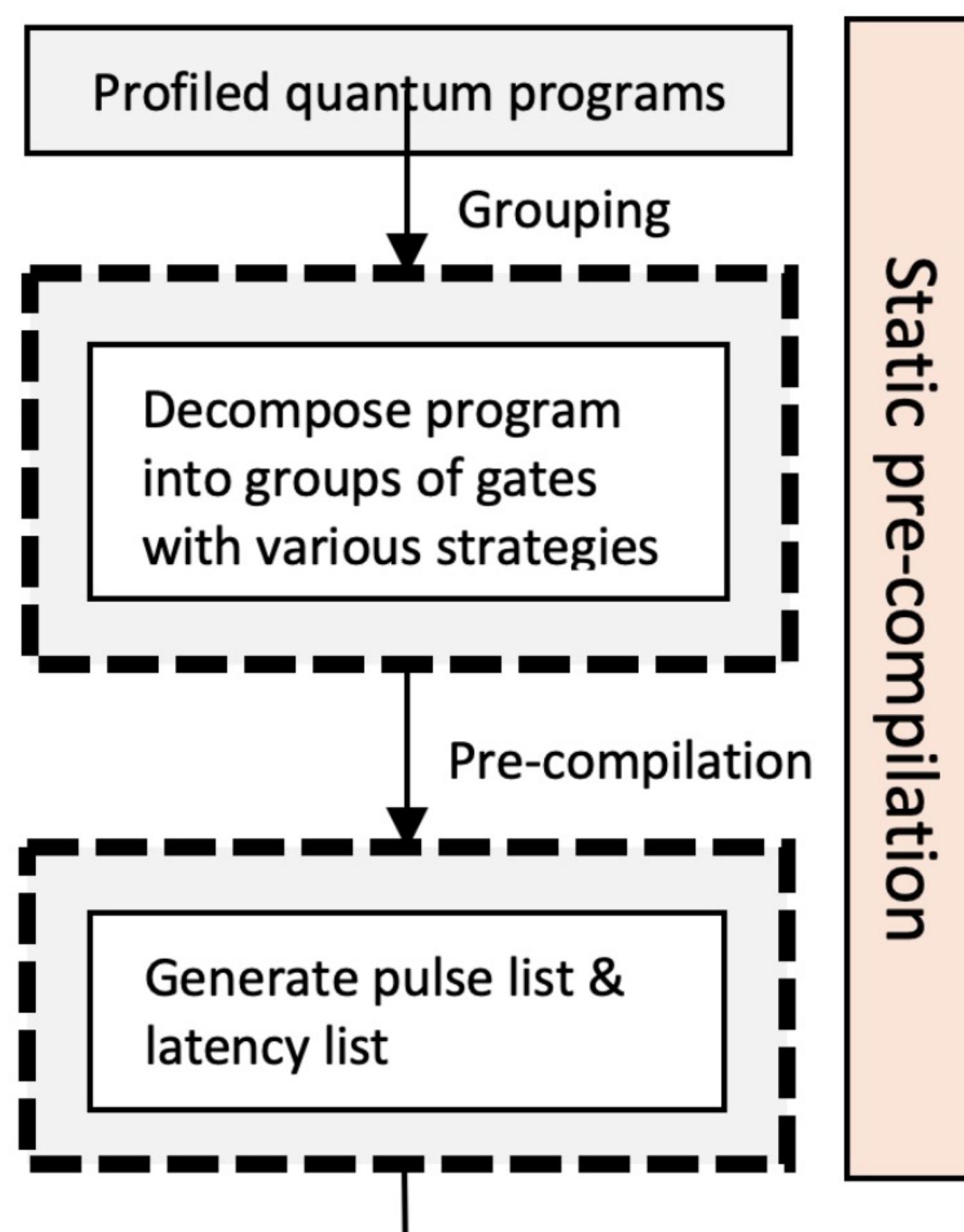
(c) Group with many qubits



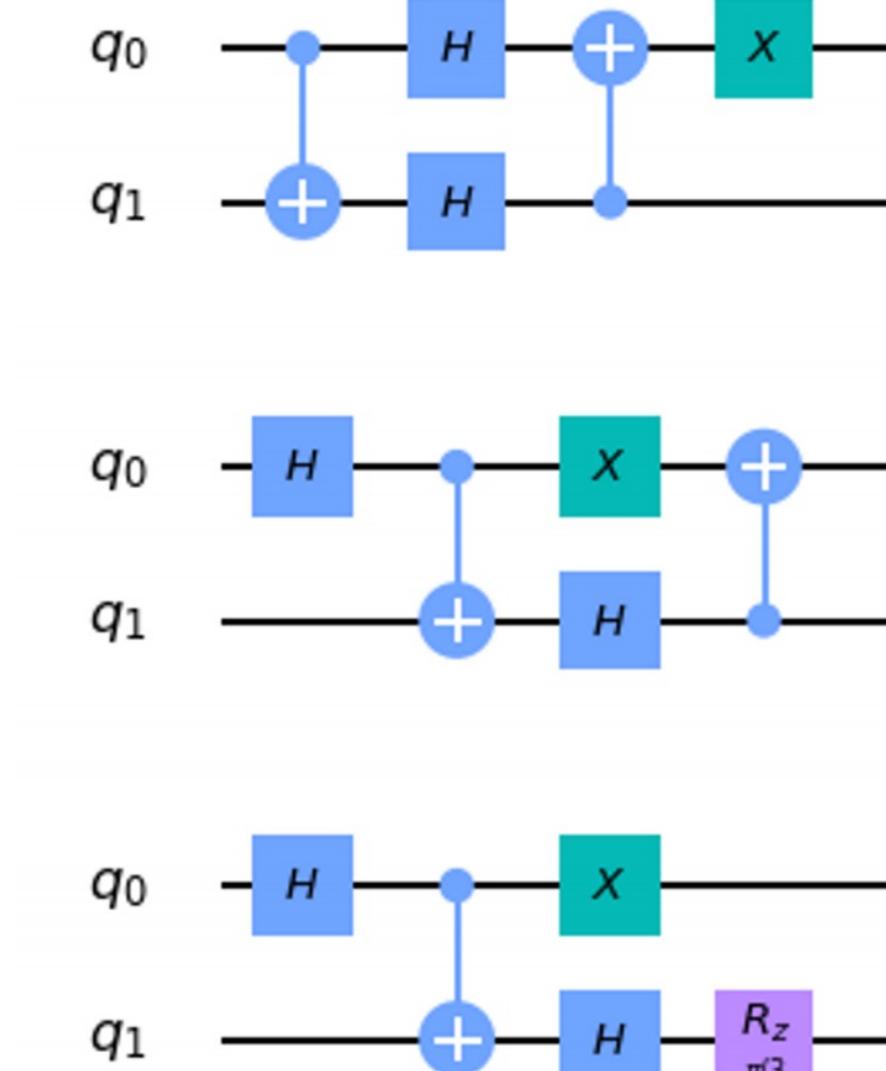
(d) Typical group in our paper

# Build the Library

- Pulses of these groups will be generated through QOC. And we get a library that stores group-pulse pairs.



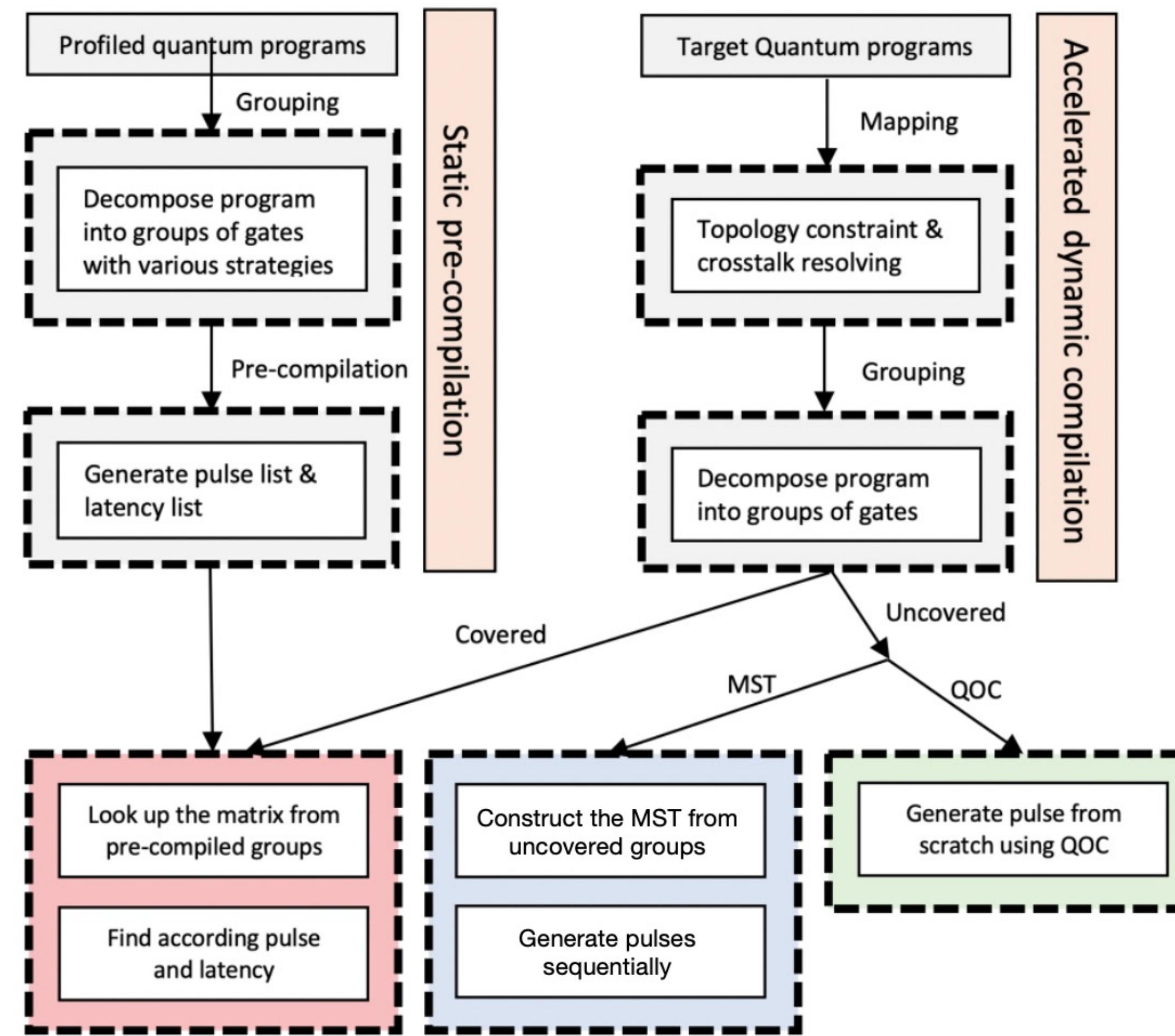
(c) Group with many qubits



(d) Typical group in our paper

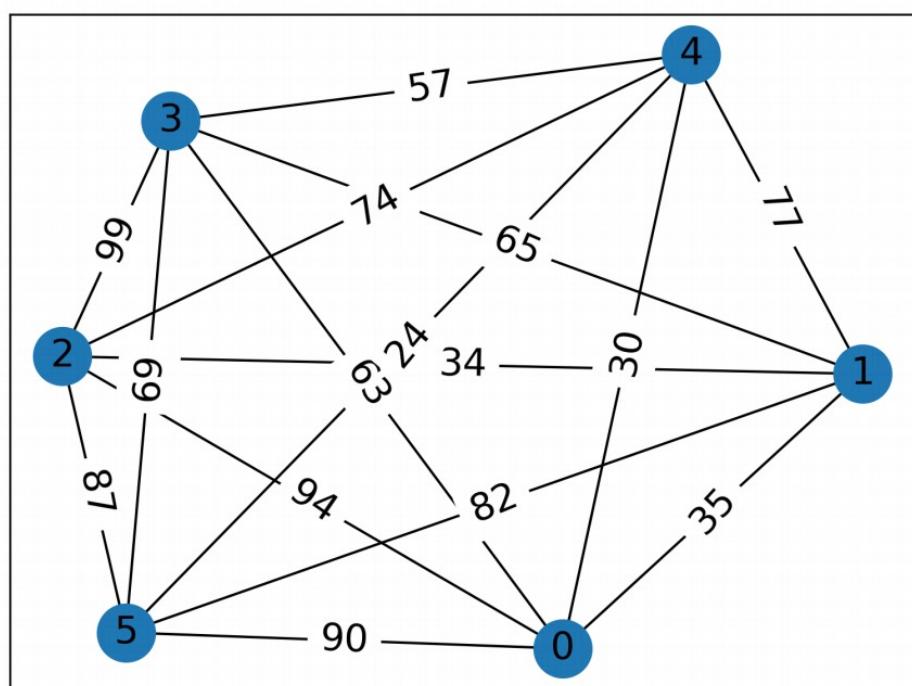
# Utilize the Library

- For groups that are covered by library, the pulse could be directly looked up.
- For uncovered groups, we could generate the pulses using pulses of very similar groups

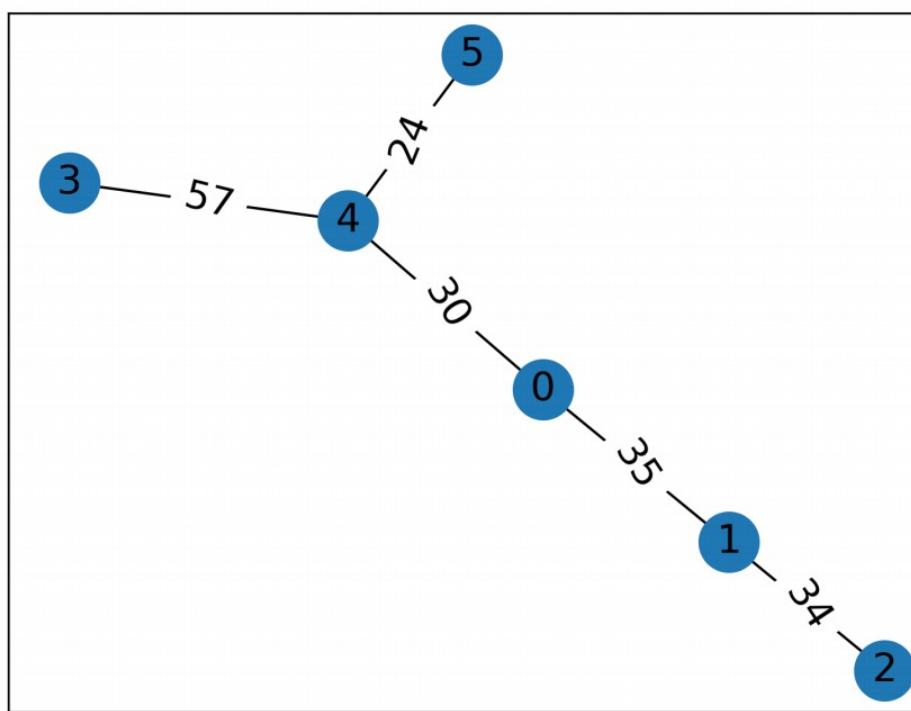


# Use MST to Accelerate QOC

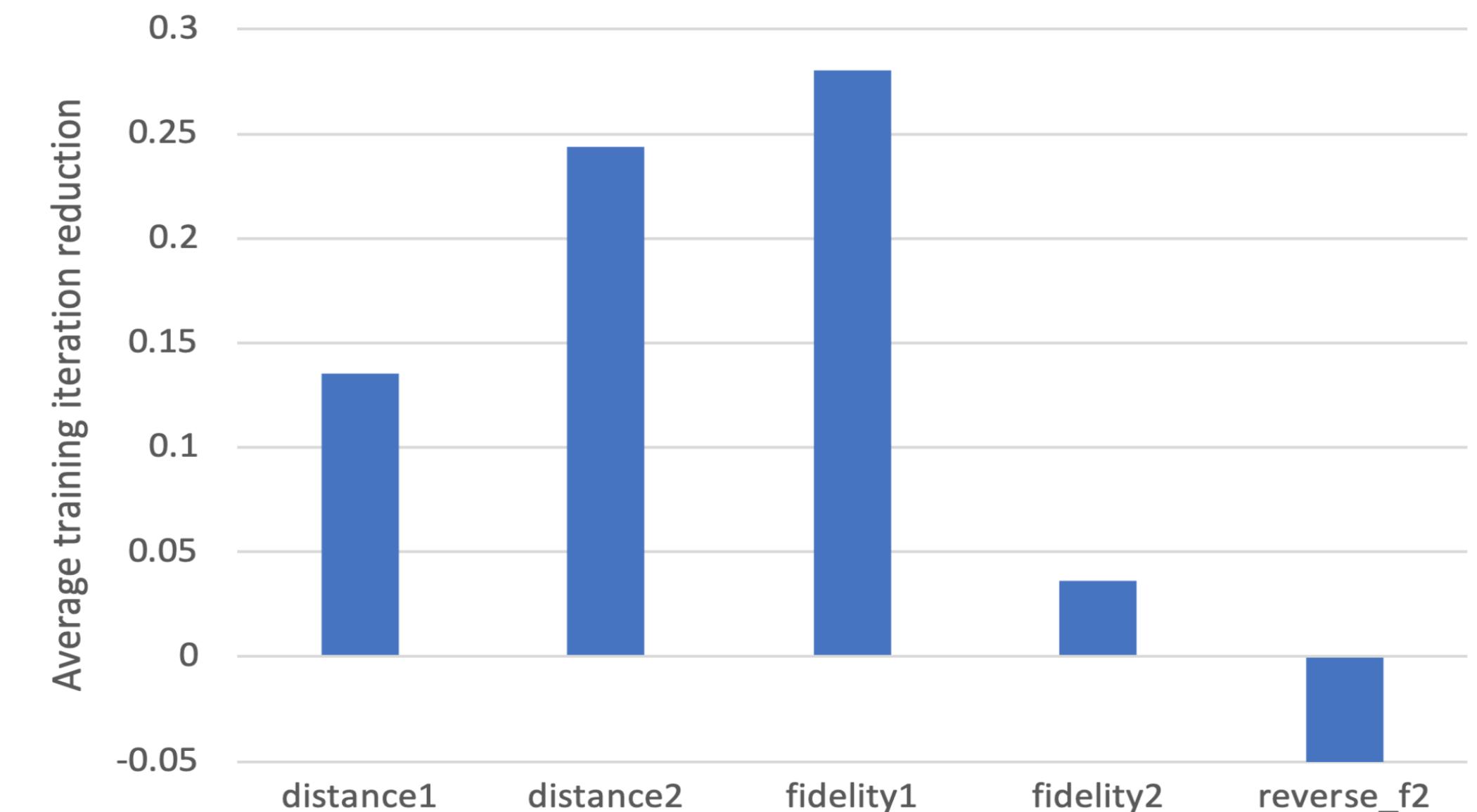
- We need to define similarity and find the corresponding Minimum spanning tree.



(a) A 6-node SG



(b) Minimum spanning tree

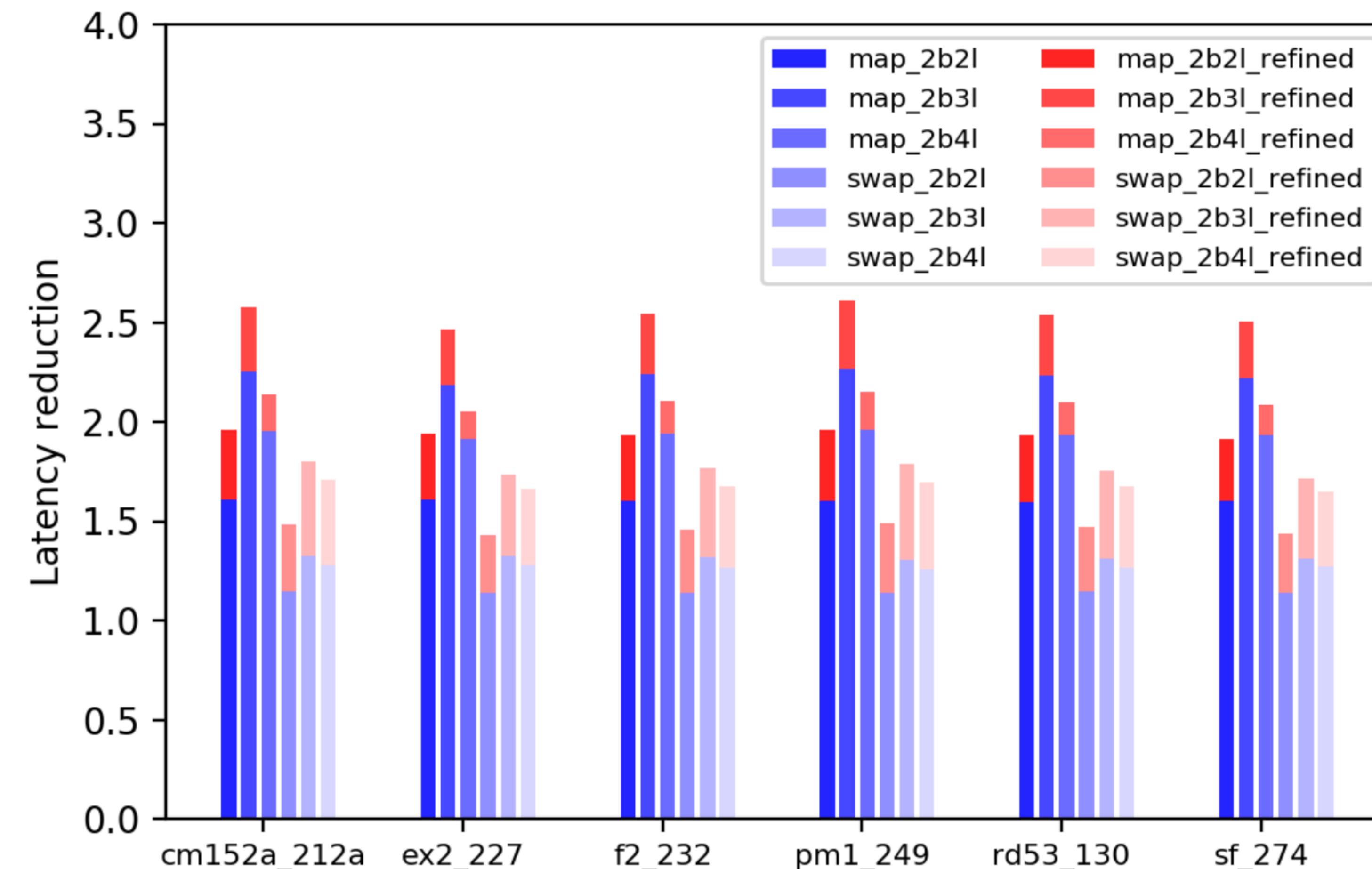


$$d_1(\mathbf{A}, \mathbf{B}) = \sum_{i=1}^n \sum_{j=1}^n |a_{ij} - b_{ij}| \quad d_2(\mathbf{A}, \mathbf{B}) = \sqrt{\sum_{i=1}^n \sum_{j=1}^n (a_{ij} - b_{ij})^2}$$

$$d_3(\mathbf{A}, \mathbf{B}) = \text{Tr}(A^*B) \quad d_4(\mathbf{A}, \mathbf{B}) = F(A, B) = \left( \text{tr} \sqrt{\sqrt{AB}\sqrt{A}} \right)^2$$

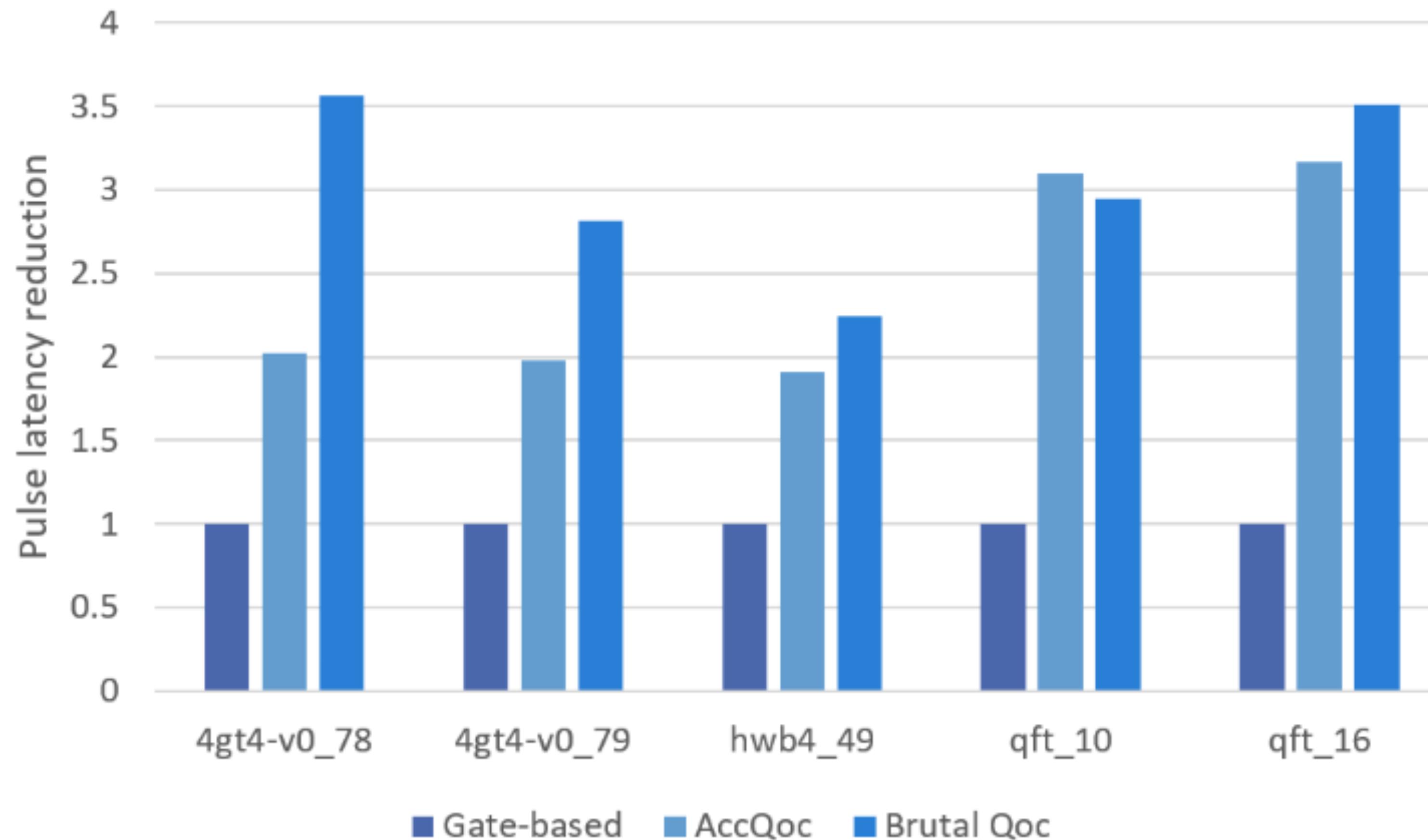
# Latency Reduction over Gate-based

- This figure shows the latency reduction compared with gate-based compilation. We show that the latency is reduced for various grouping strategy.



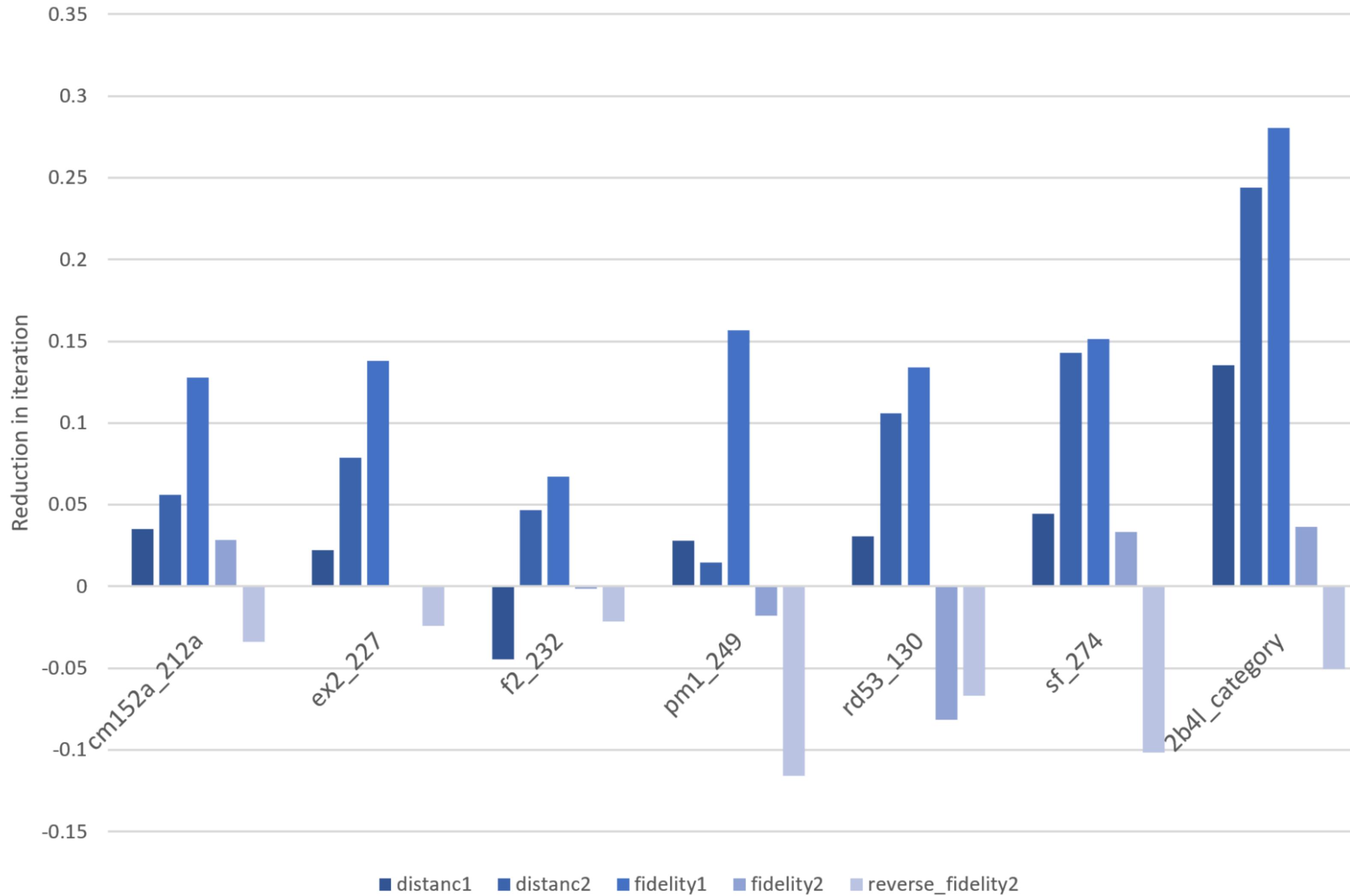
# Latency Reduction V.S. Brute-force

- This figure shows the latency reduction compared with brute-force compilation. Brute-force QOC means that we give QOC a very large group of gates to reach maximum latency reduction.



# Reduction of Training Iteration

- We see reduction of training iteration when we adopt the idea of similarity.



# Hands-On Section

## 3.1 Quantum Optimal Control



# TorchQuantum Tutorial Outline

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations 

1.3 TQ for State Prep 

1.4 TQ for VQE 

1.4 TQ for QNN 

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning 

2.2 QuantumNAT: Noise Injection and Quantization 

2.3 QOC: On-Chip Training 

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression 

## Section 3

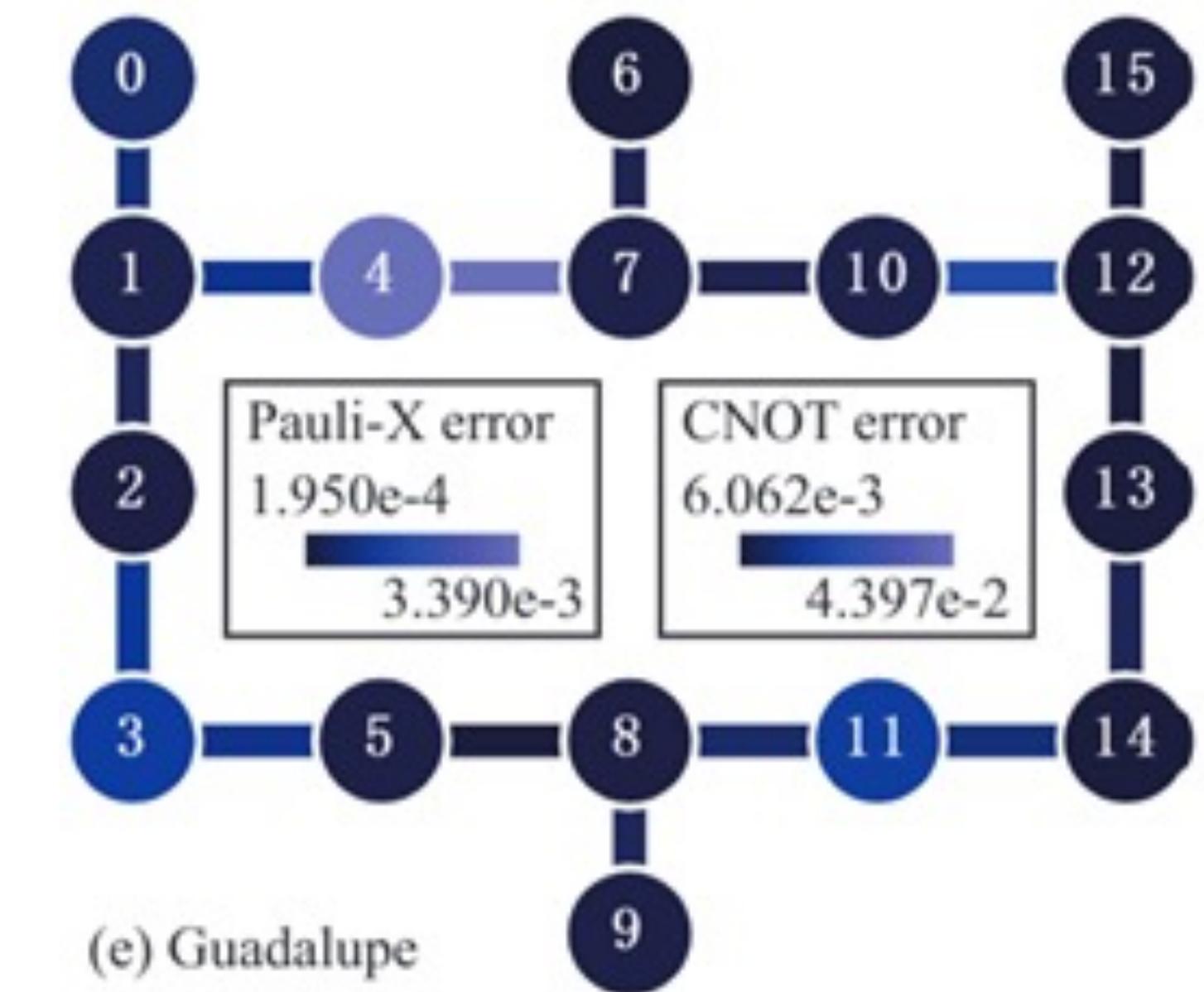
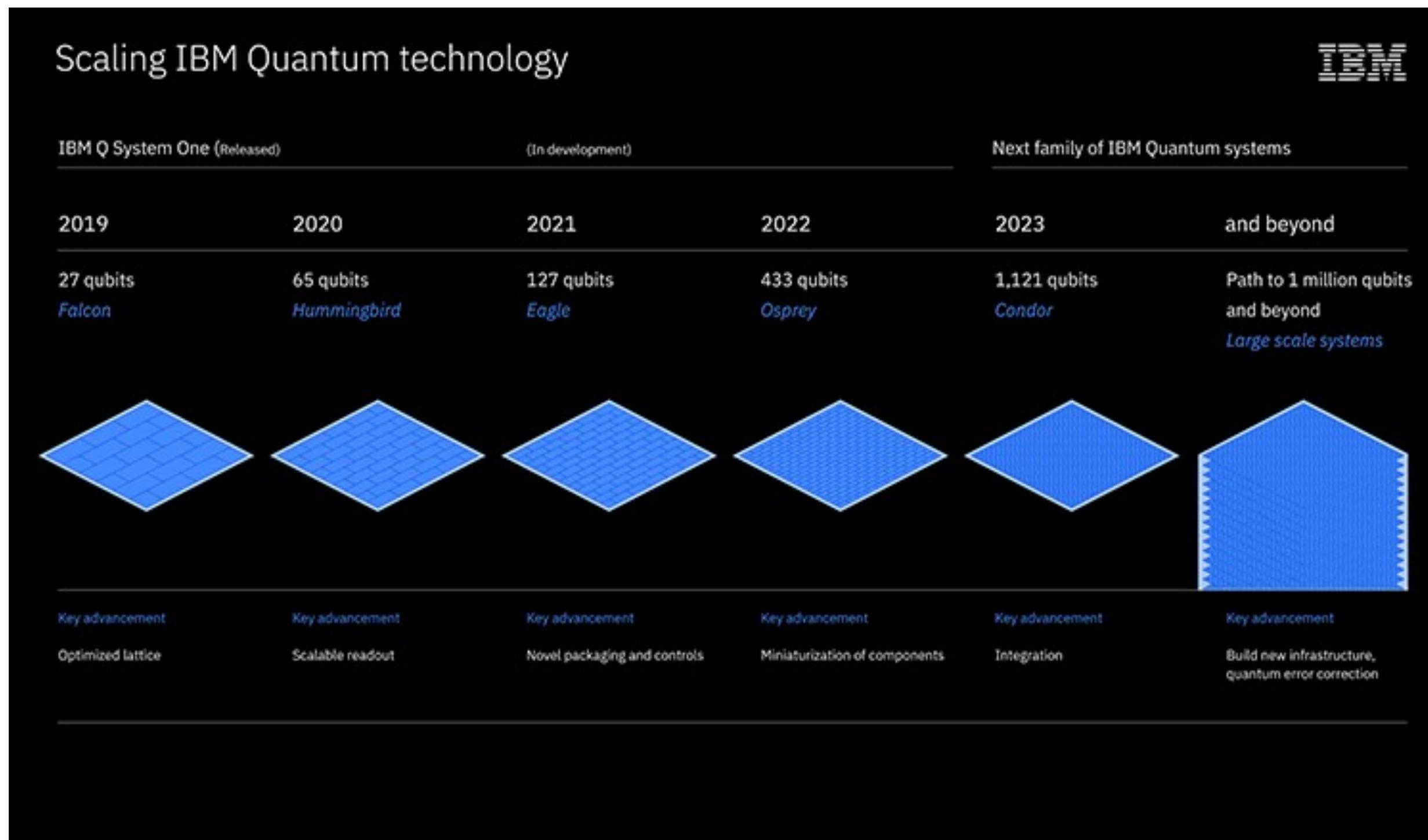
### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control 

3.2 Variational Pulse Learning 

# Variational Quantum Pulse Learning (VQP)

- Coherence time is limited in NISQ machines, which means we cannot perform a huge quantum circuit with enough depth and width in NISQ machines.
- Noise and compilation overhead seriously affect the performance of a quantum program.



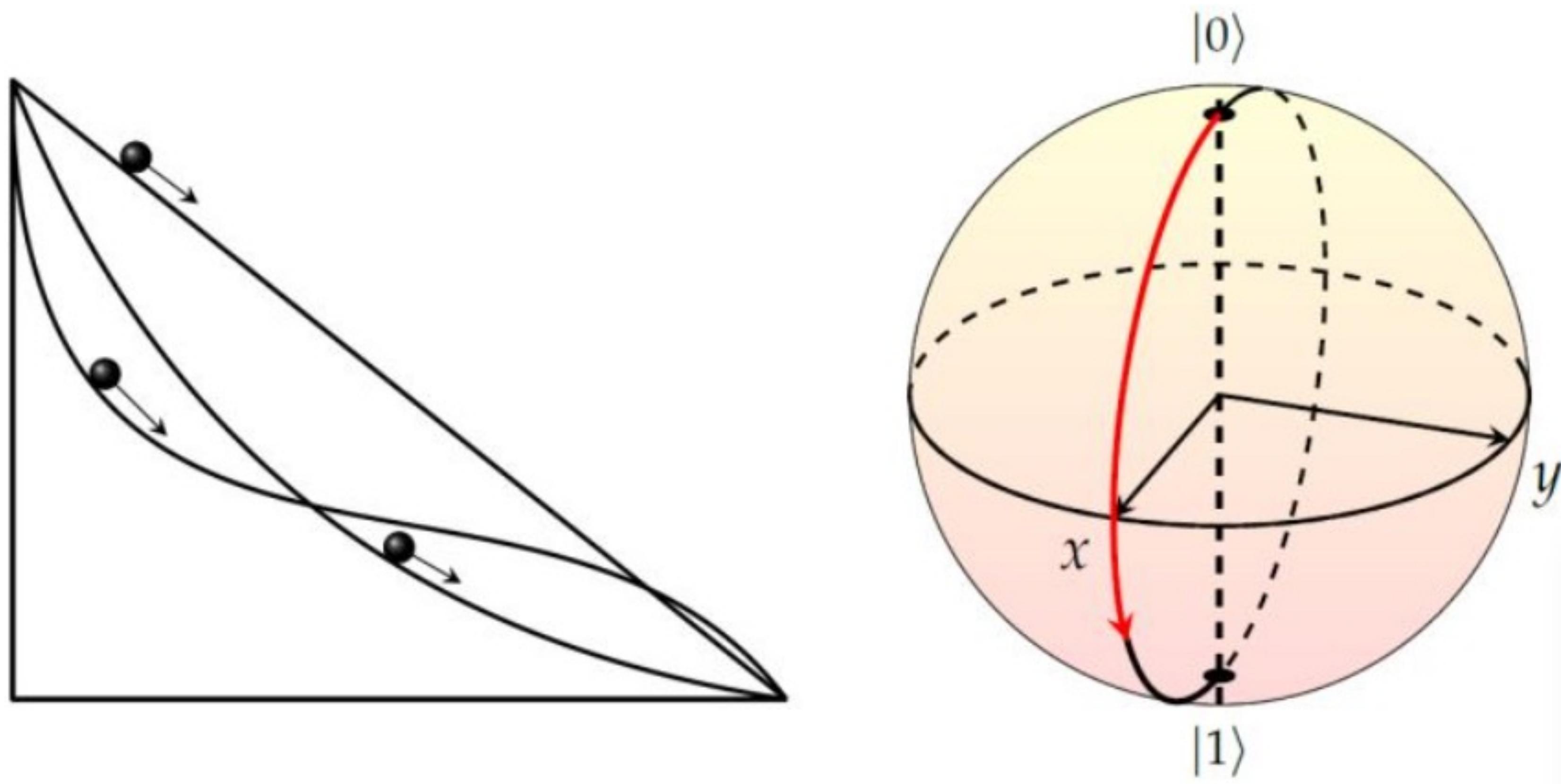
Error rate on a bit in CMOS Device  
error rate is about  $10^{-15}$

But error rate on a quantum bit  
reaches  $10^{-4}$  to  $10^{-2}$

HITMAN.I.A

# Motivation and Challenges

- QOC is limited to few qubits since it is computationally expensive.
- Most works demonstrate QOC on quantum simulators, however, it is hard to be evaluated on NISQ machines.



# Attempt on Quantum Neural Network

- Can we find an intermediate-level approach between the gate level operations and quantum optimal control?
- Can we achieve benefits by doing so?

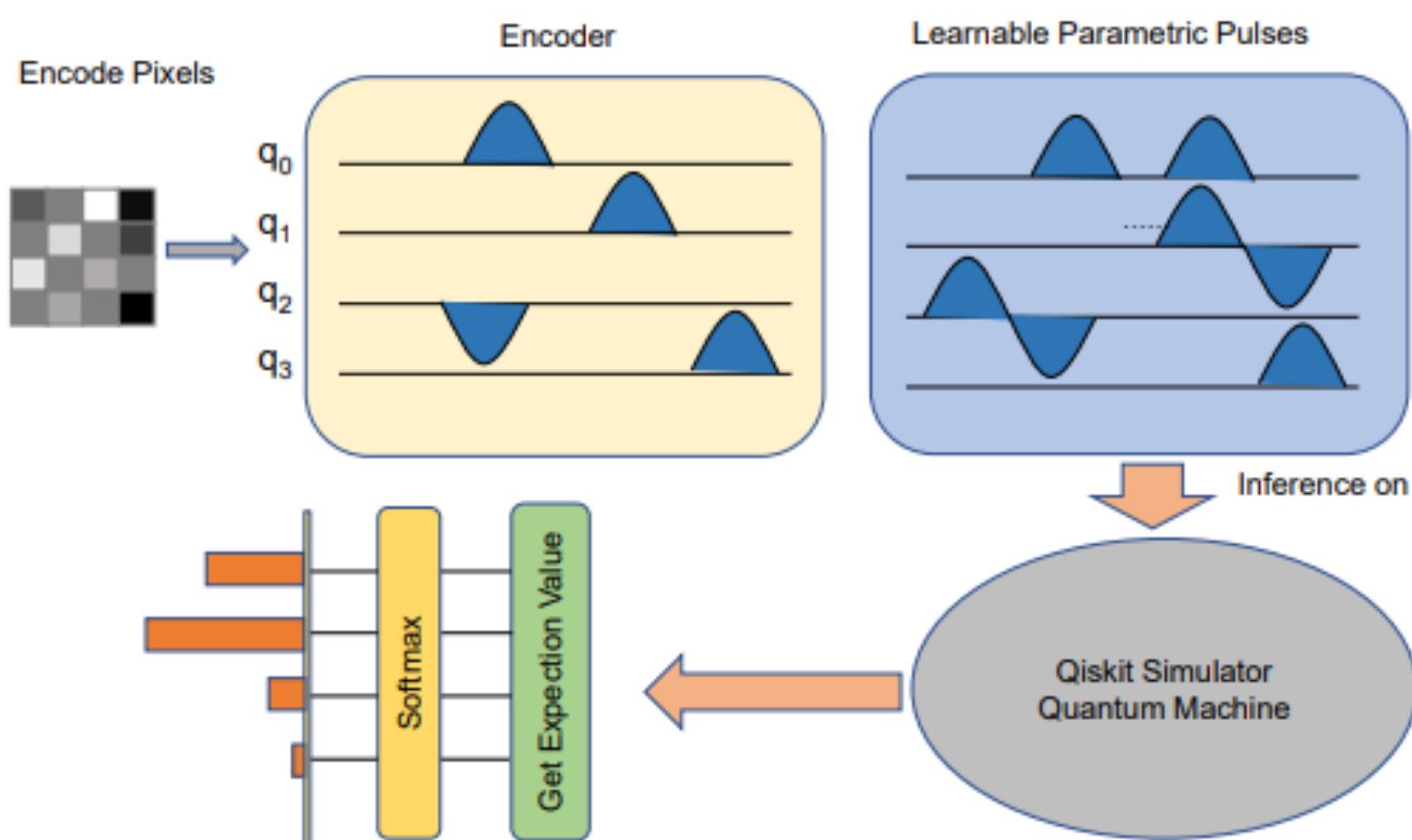


Fig. 1: Conceptual illustration of VQP for QML tasks.

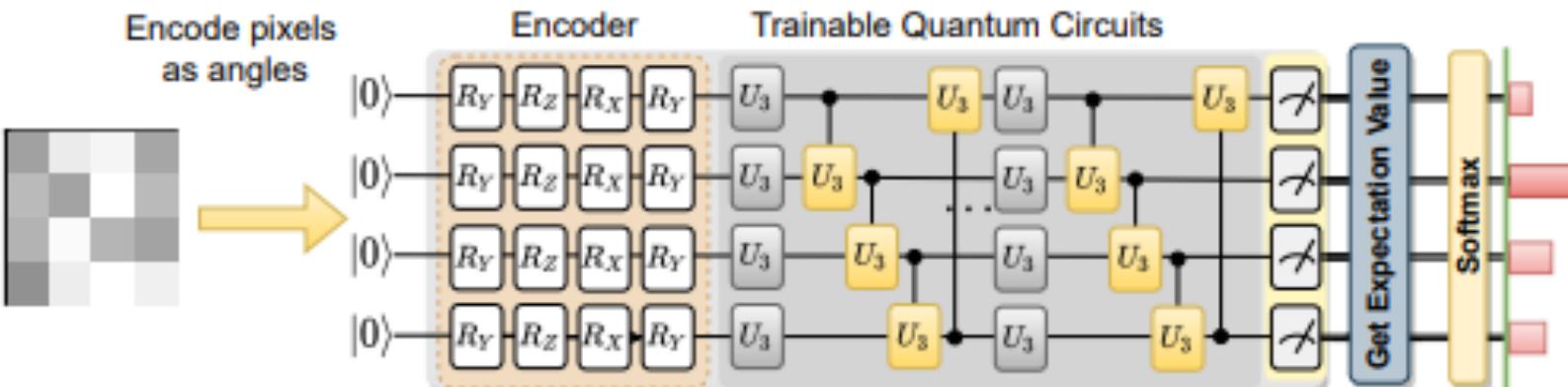
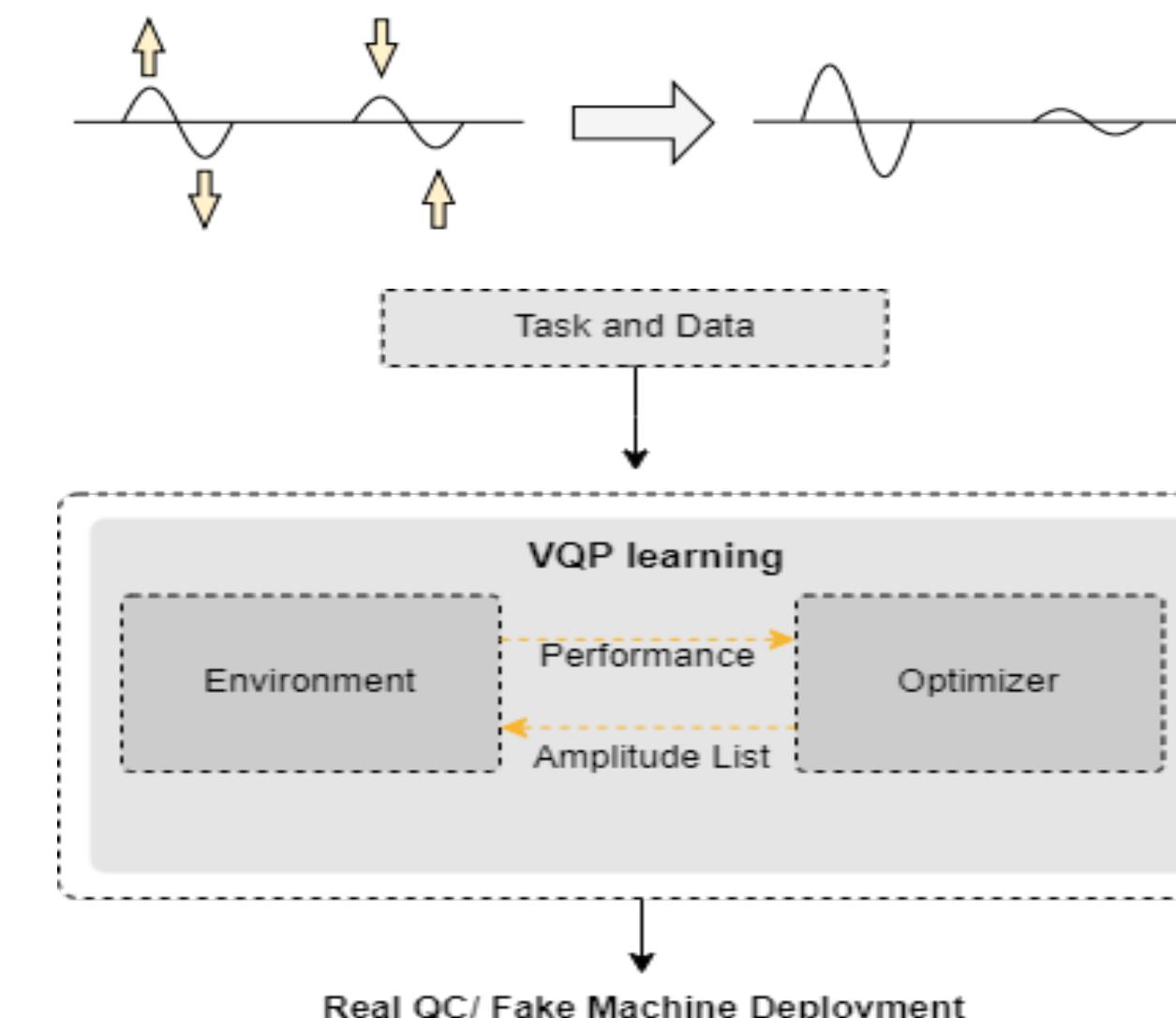


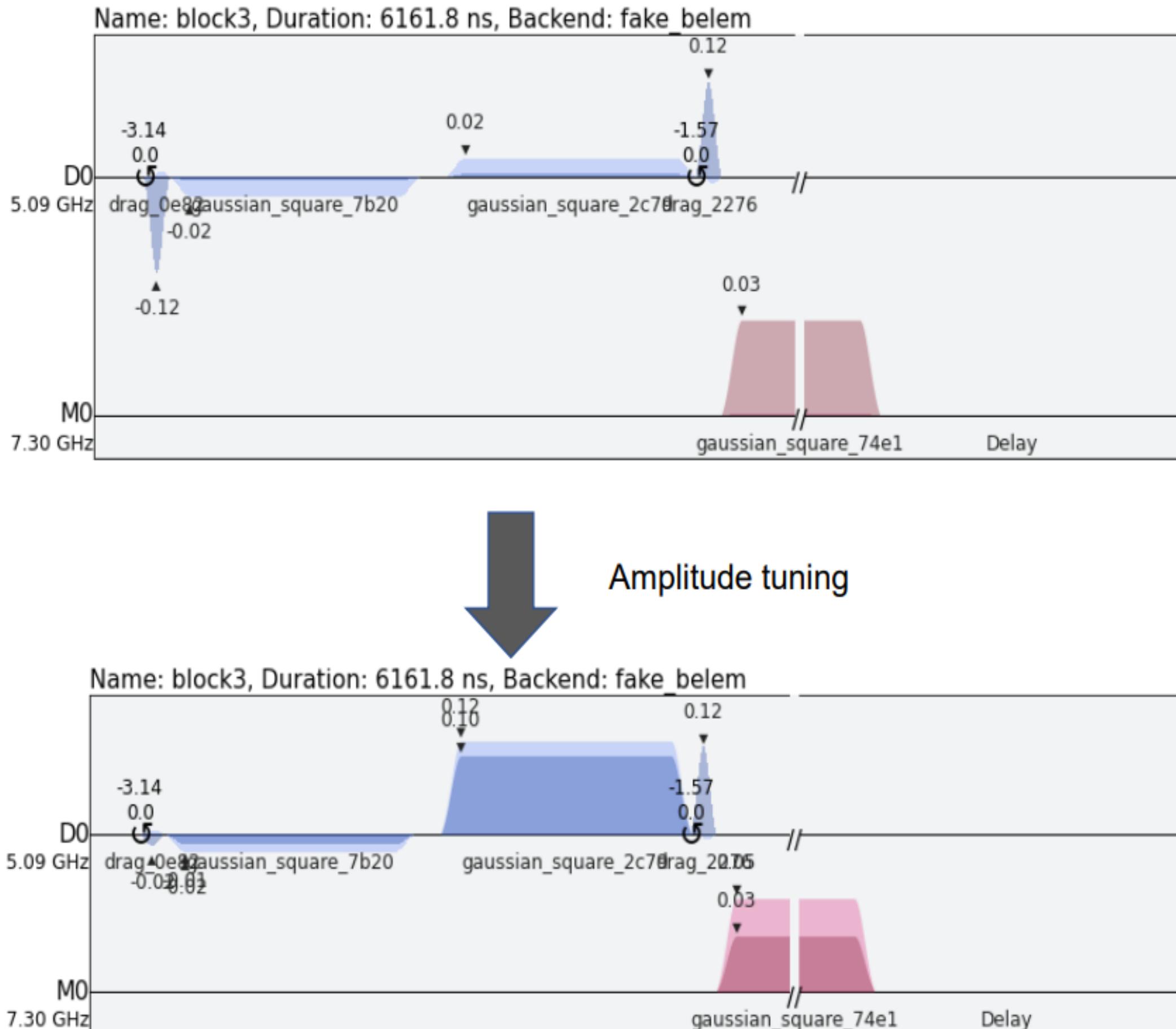
Fig. 2: An example QNN that uses VQC for QML tasks.



MIT HAN LA

# Why Variational Quantum Pulse Learning ?

- VQC with more gates has similar performance in terms of accuracy



$$H = \sum_{i=0}^1 (U_i(t) + D_i(t))\sigma_i^X + \sum_{i=0}^1 2\pi\nu_i(1 - \sigma_i^Z)/2$$

$$+ \omega_B a_B a_B^\dagger + \sum_{i=0}^1 g_i \sigma_i^X (a_B + a_B^\dagger)$$

$$D_i(t) = \text{Re}(d_i(t)e^{iw_{d_i}t})$$

$$U_i(t) = \text{Re}[u_i(t)e^{i(w_{d_i} - w_{d_j})t}] \quad (2)$$

Model	# of Gates	Accuracy
VQC_base	9	0.62
<b>VQP</b>	<b>9</b>	<b>0.71</b>
VQC*	12	0.68

# Optimization Framework

---

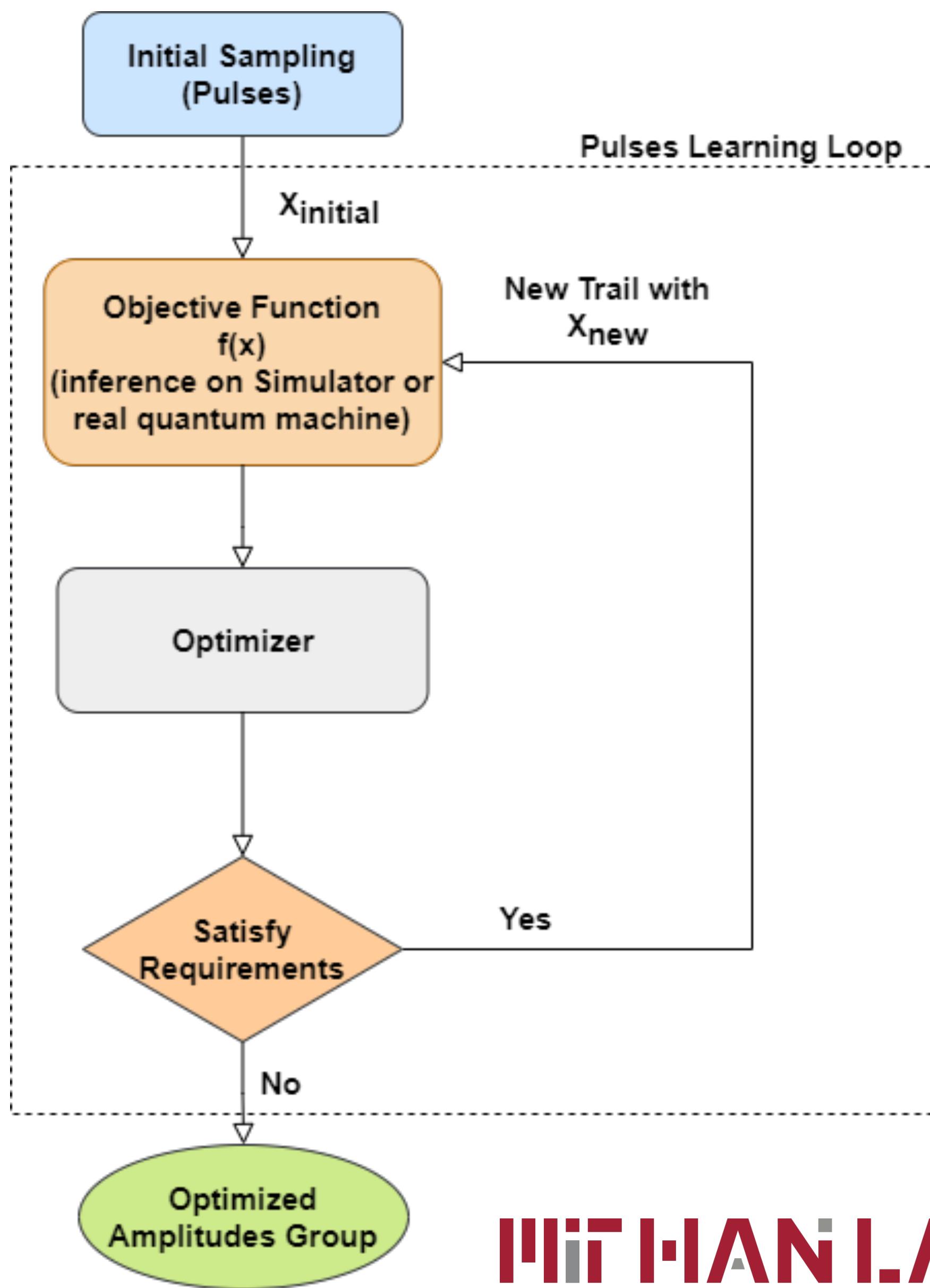
**Algorithm 1:** Variational Pulse BO Learning

---

**Data:**  $\rho, \chi, M, D$   
//  $\rho$  is the amplitude list,  $\chi$  is the search bound,  $D$  consists of  $x_i$  and  $y_i$ ,  $M$  is the Gaussian Process Regression model.

```
D ← InitPulses(ρ, χ);
for i ← |D| to N_total do
    // iterative optimization
    p(y|x, D) ← FitModel(M, D);
    // Acquisition function actively searches for the
    // next optimized amplitudes.
    xi ← argminx ∈ χ S(x, p(y|x, D));
    // Calculate corresponding error rate by processing
    // in quantum machine.
    yi ← f(xi);
    D ← D ∪ (xi, yi);
end
```

---



# Experiment Result

Model	Accuracy	
	Noise simulator (Belem)	ibmq_jakarta
VQC learning 20	0.57	0.58
<b>VQP learning 20</b>	<b>0.6</b>	<b>0.69</b>
VQC learning 100	0.61	0.59
<b>VQP learning 100</b>	<b>0.63</b>	<b>0.64</b>
VQC learning MNIST 20	0.6	0.56
<b>VQP learning MNIST 20</b>	<b>0.66</b>	<b>0.62</b>
VQC learning MNIST 100	0.57	0.62
<b>VQP learning MNIST 100</b>	<b>0.61</b>	<b>0.71</b>

Achieves higher accuracy under same condition

Model	# of Gates	Accuracy
VQC_base	9	0.62
<b>VQP</b>	<b>9</b>	<b>0.71</b>
VQC*	12	0.68

VQC with more gates has similar performance in terms of accuracy

# Benefits Observed from VQP

Form of CX gate	Noise simulator (Quito)	Time Duration		
		Noise simulator (Belem)	Noise simulator (Jakarta)	
CRX( $\pi$ ) gate	26832.0dt	32016.0dt	26832.0dt	
CX gate	25136.0dt	27728.0dt	25136.0dt	

**Advantage in specific gate**

Model	# of Gate	Time Duration	
		ibmq_jakarta	Noise simulator (Belem)
<b>VQP</b>	<b>9</b>	<b>40816.0dt</b>	<b>45168.0dt</b>
VQC*	12	58896.0dt	58768.0dt
<b>VQP_transpiled</b>	<b>11</b>	<b>32368.0dt</b>	<b>32816.0dt</b>
VQC*_transpiled	17	53008.0dt	46192.0dt

**Advantage in general circuit**

# Challenge for Pulse Learning

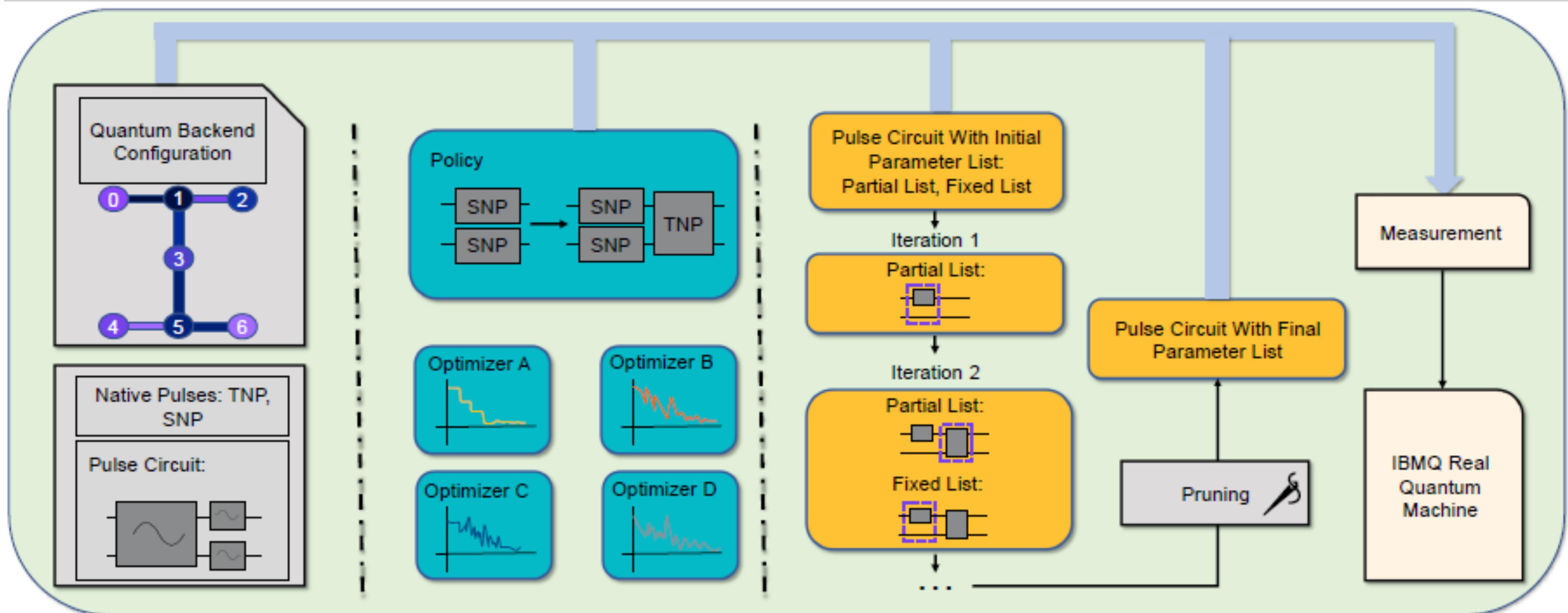
- Non-gradient-based optimizer has randomness when parameter in high dimensional space.
- Qiskit pulse simulator is not efficient, e.g., need around 3 mins to finish a 9-gate circuit.

Model	# of Gates	Accuracy
VQC_base	9	0.62
<b>VQP</b>	<b>9</b>	<b>0.71</b>
VQC*	12	0.68

Model	# of Gates	Accuracy
<b>VQP</b>	<b>9</b>	<b>0.71</b>
VQC with gradient	9	0.73
VQC* with gradient	12	0.77

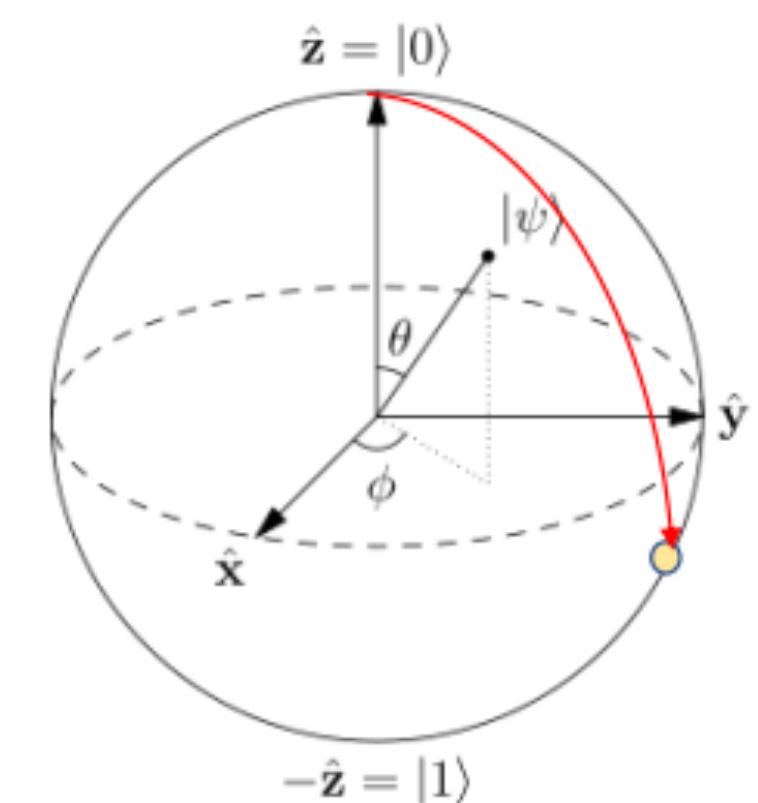
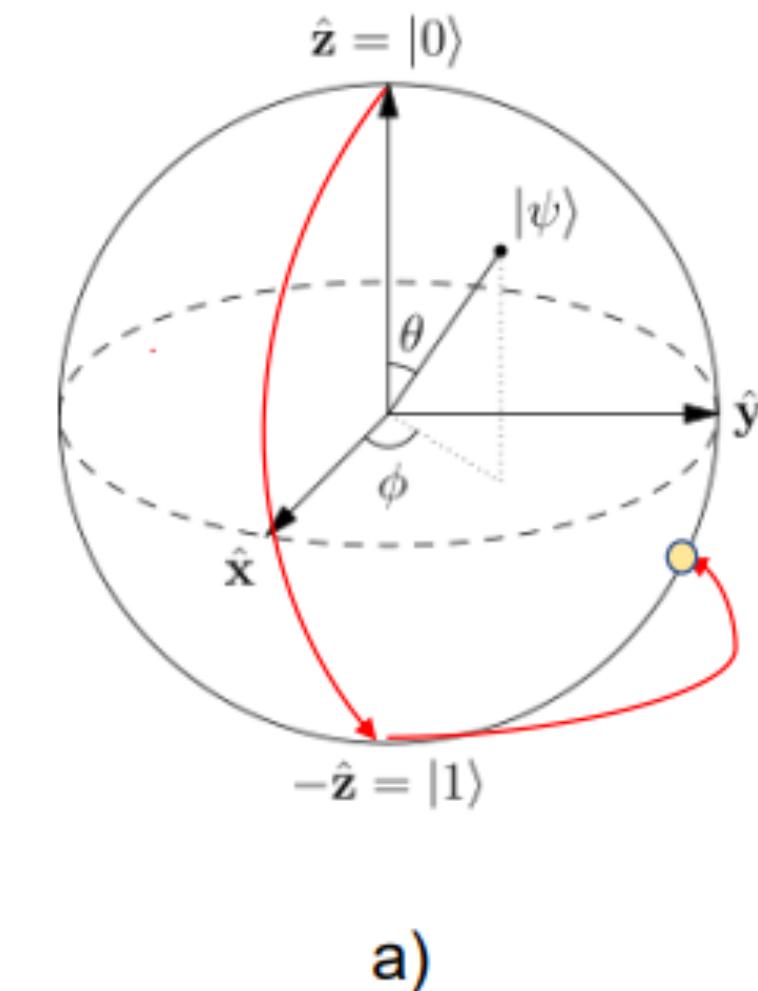
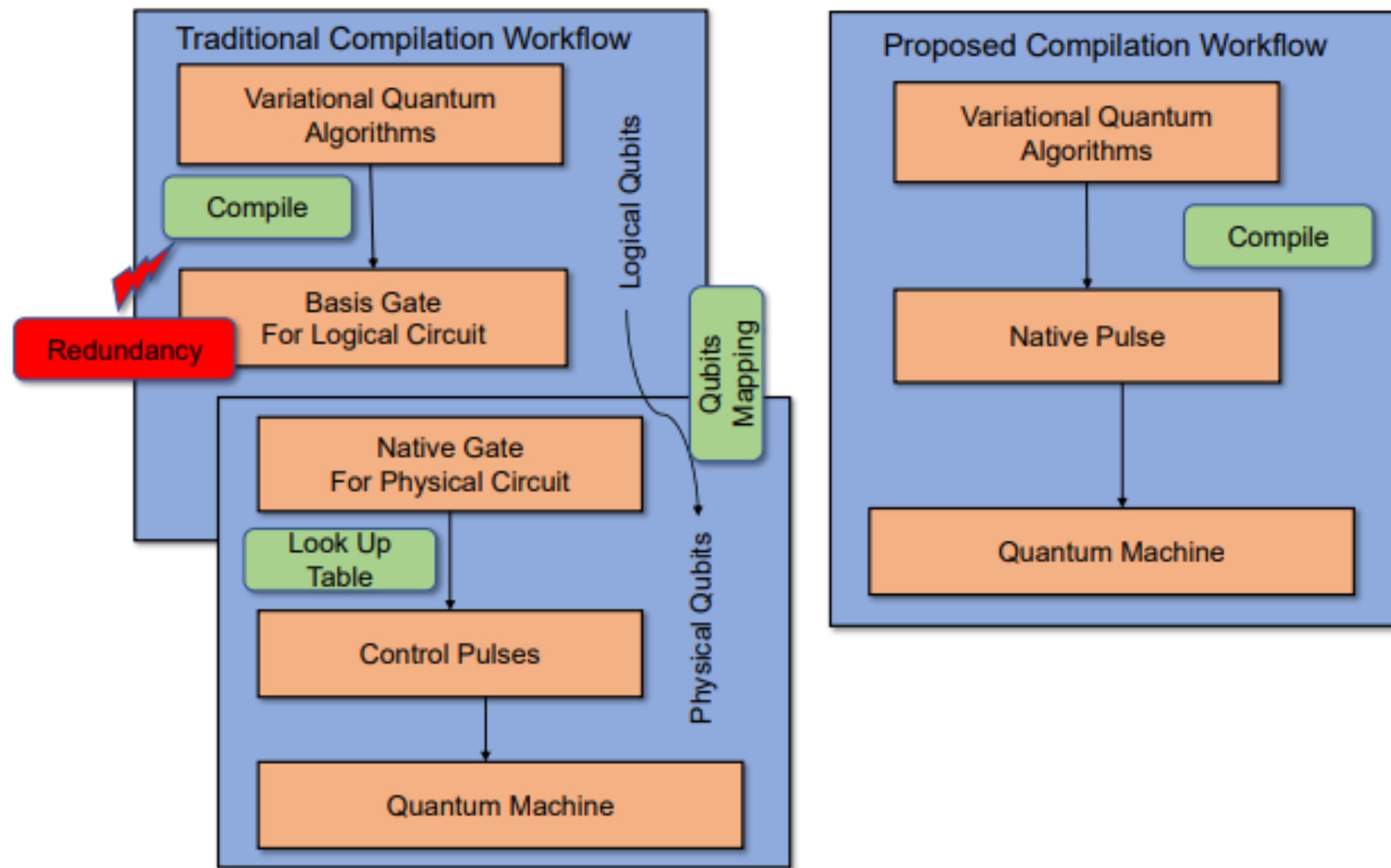
# Pulse Ansatz on VQAs

- Can we find a hardware efficient ansatz at the pulse level?
- What is a good pulse ansatz?



# Benefits of Pulse Ansatz

- Source of Advantages:
- Enable flexible and efficient compilation workflow on pulse-level.

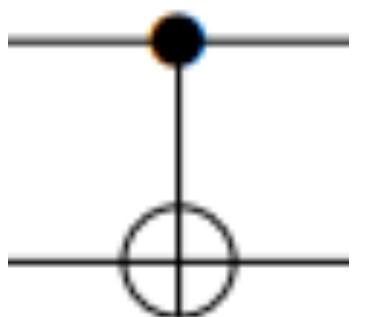


# Benefits of Pulse Ansatz

- Source of Advantages:
- Two-qubit pulse is tunable, whereas two-qubit gates have few flexibility.

COMPARISON OF TRAINABILITY FOR DIFFERENT PULSE CIRCUITS AND GATE CIRCUITS ON IBMQ\_JAKARTA.

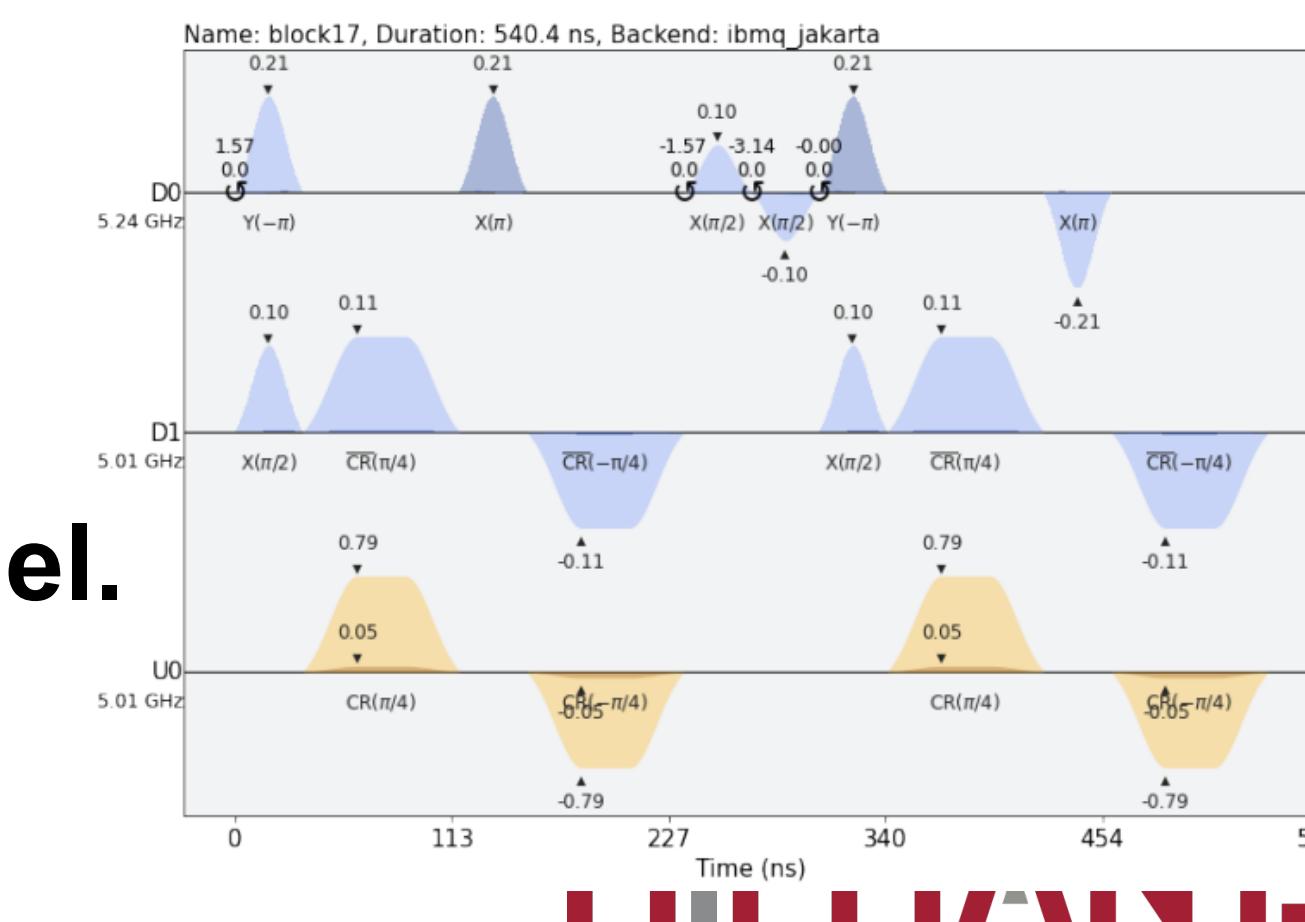
Operations	Circuit Level	Molecule Bond Length	Reference Energy	VQE ( $H_2$ ) Result	Duration(on ibmq_jakarta)
SNP	Pulse Circuit	0.1Å	2.710H	4.380H	71.1ns
TNP	Pulse Circuit	0.1Å	2.710H	2.927H	163.6ns
SNP	Pulse Circuit	0.75Å	-1.137H	-0.549H	71.1ns
TNP	Pulse Circuit	0.75Å	-1.137H	-1.032H	163.6ns
TNP + SNP	Pulse Circuit	0.75Å	-1.137H	-1.036H	234.7ns
Two Gate Ansatz	Gate Circuit	0.75Å	-1.137H	-0.534H	341.3ns



✗ Nothing can do with two qubits gate

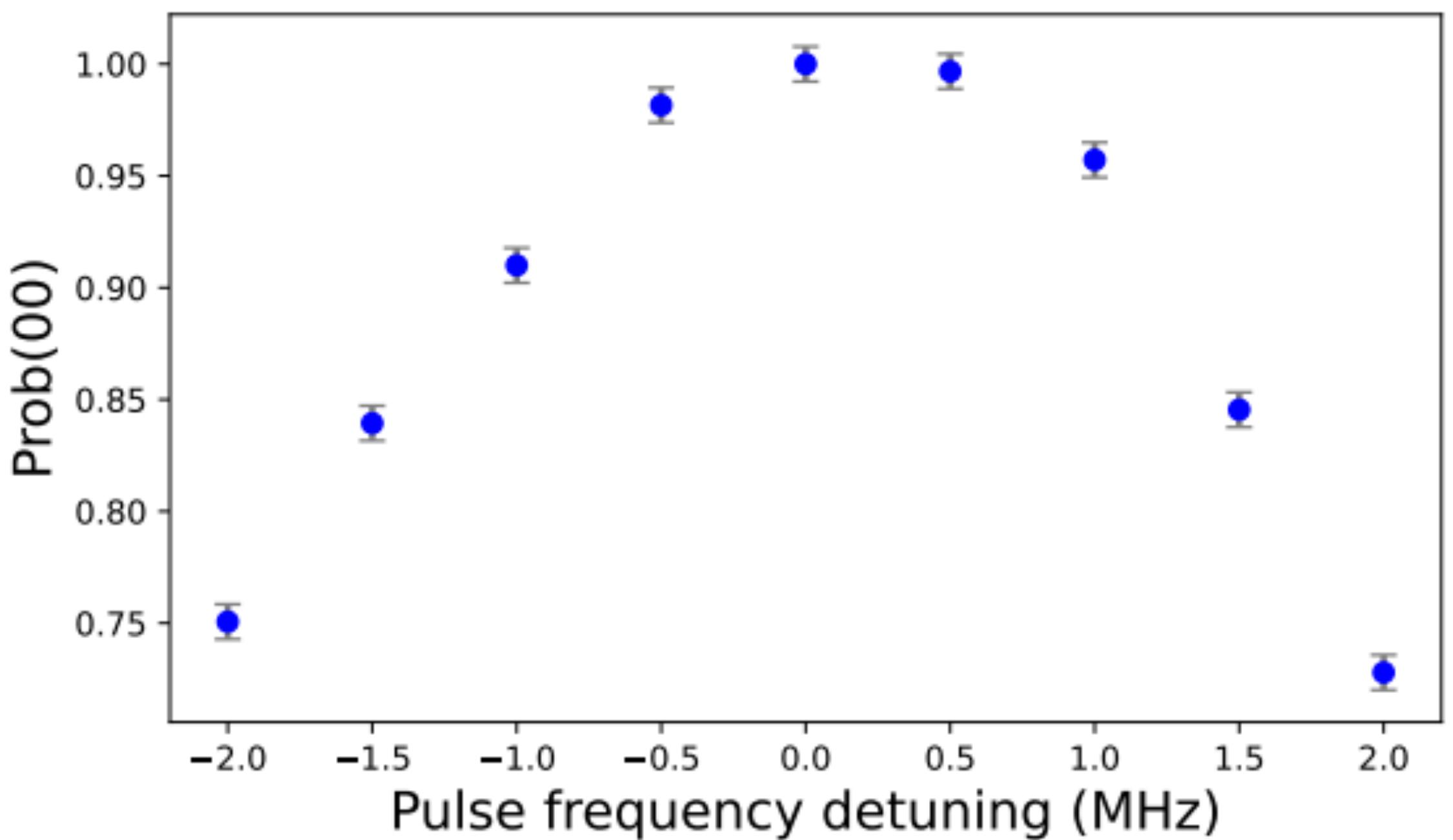
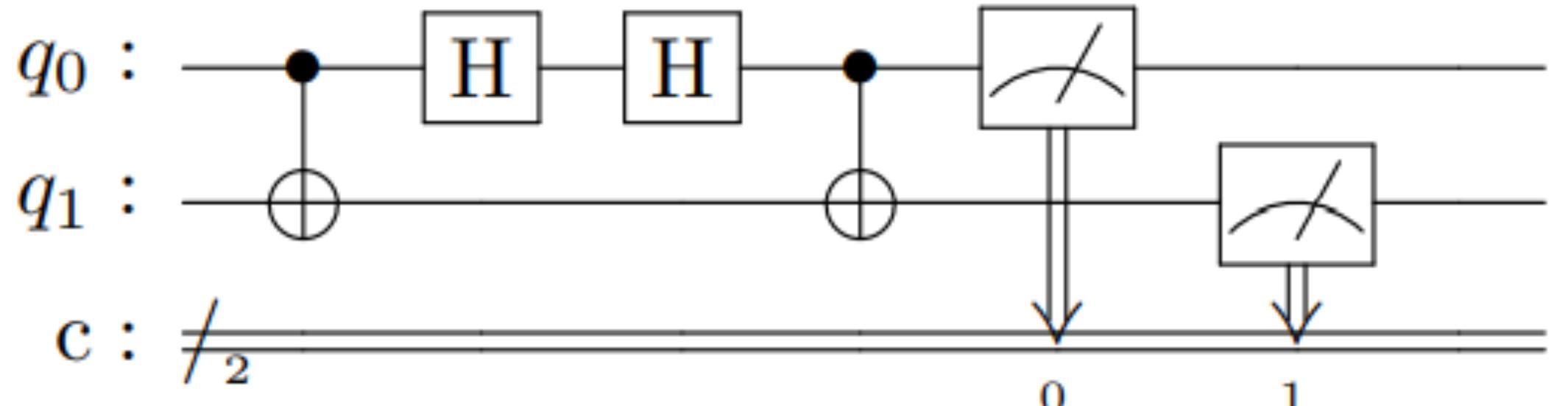


✓ Parameters are tunable on control channel.



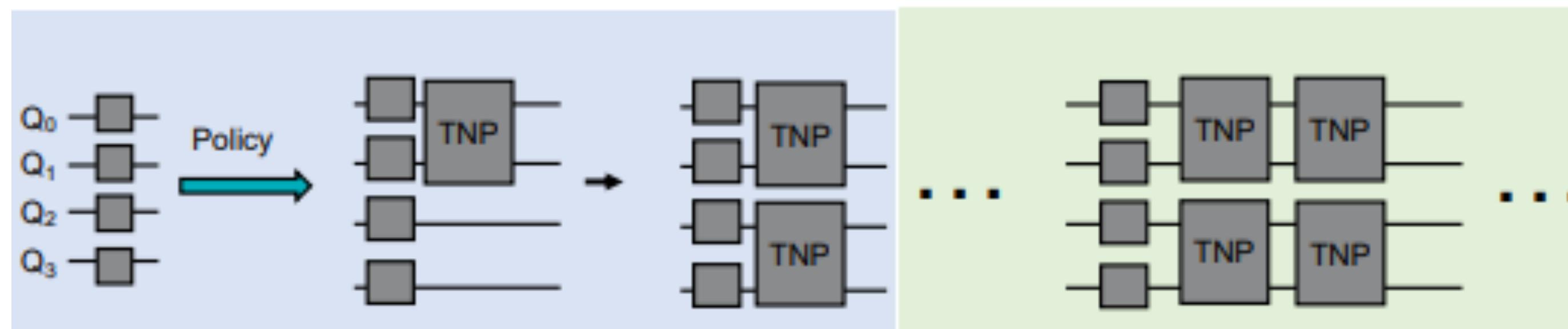
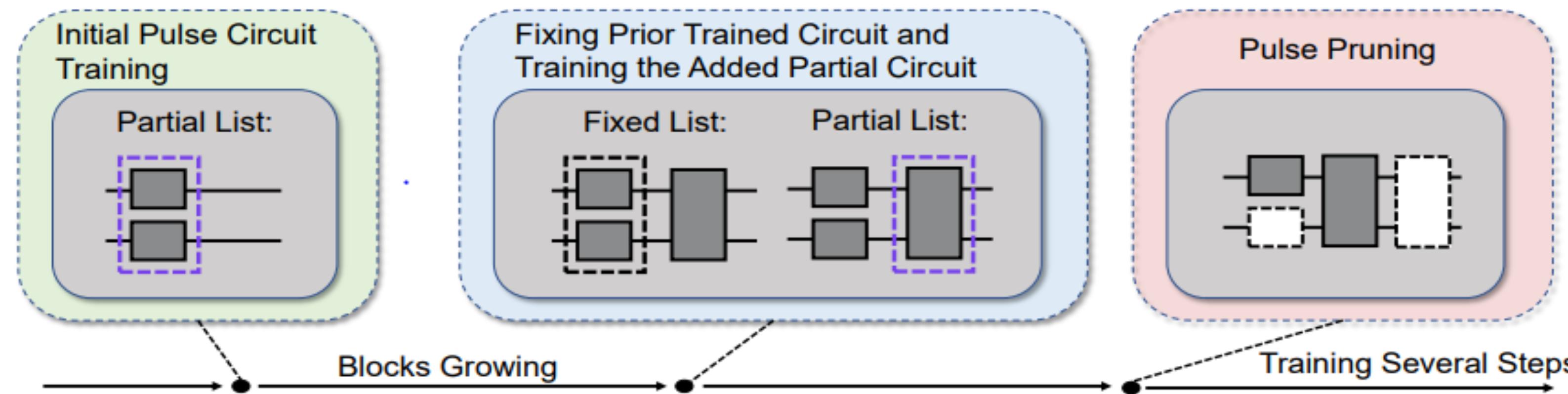
# Benefits of Pulse Ansatz

- Source of Advantages:
- Capability to tune frequency on pulse-level



# Framework and Evaluation

- Progressive learning fix the problem that non gradient optimizer cannot hold high dimensional parameters, and progressive approaching to target point is also fit the quantum speed limit (QSL) theory.

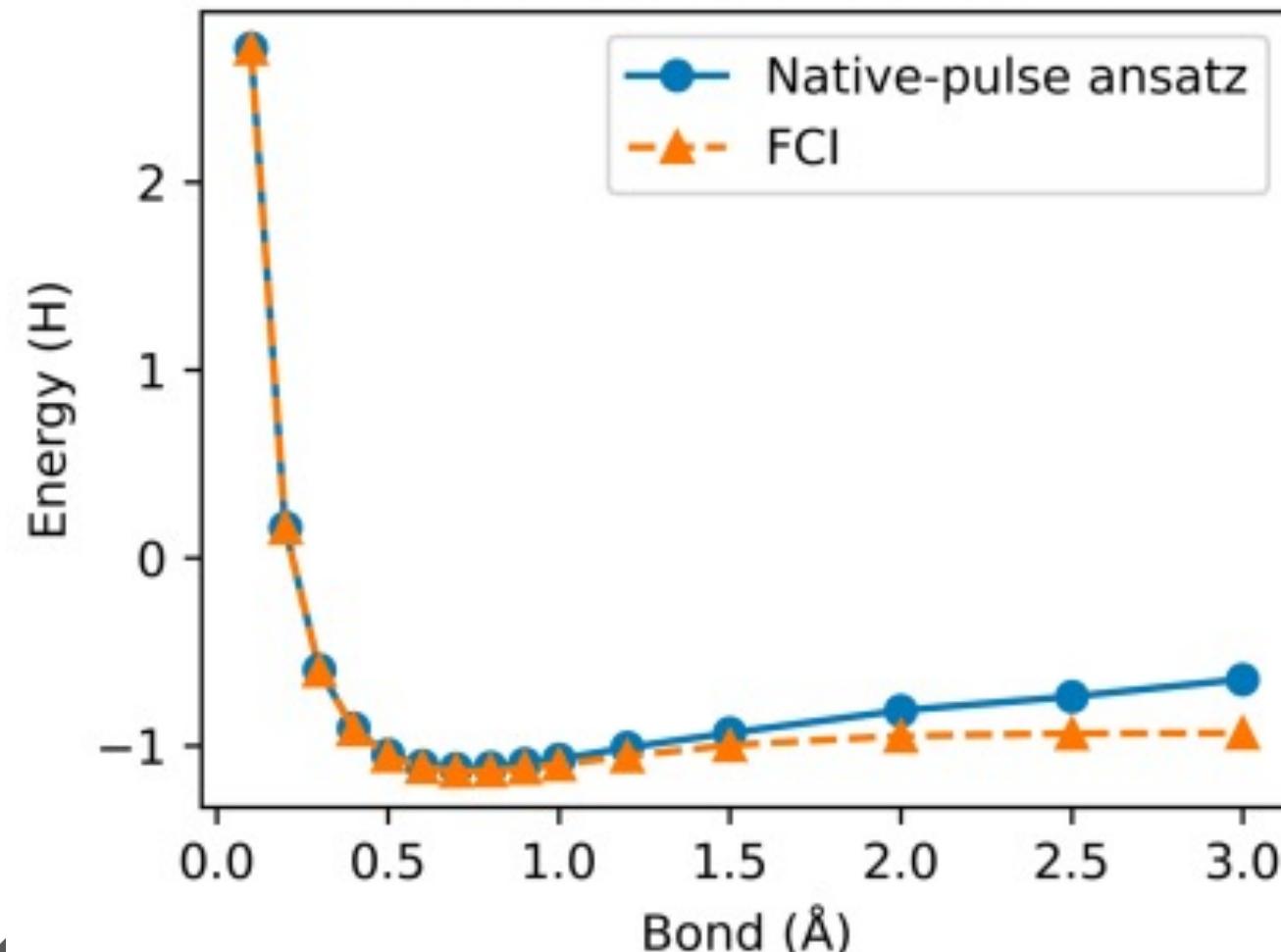


# Framework and Evaluation

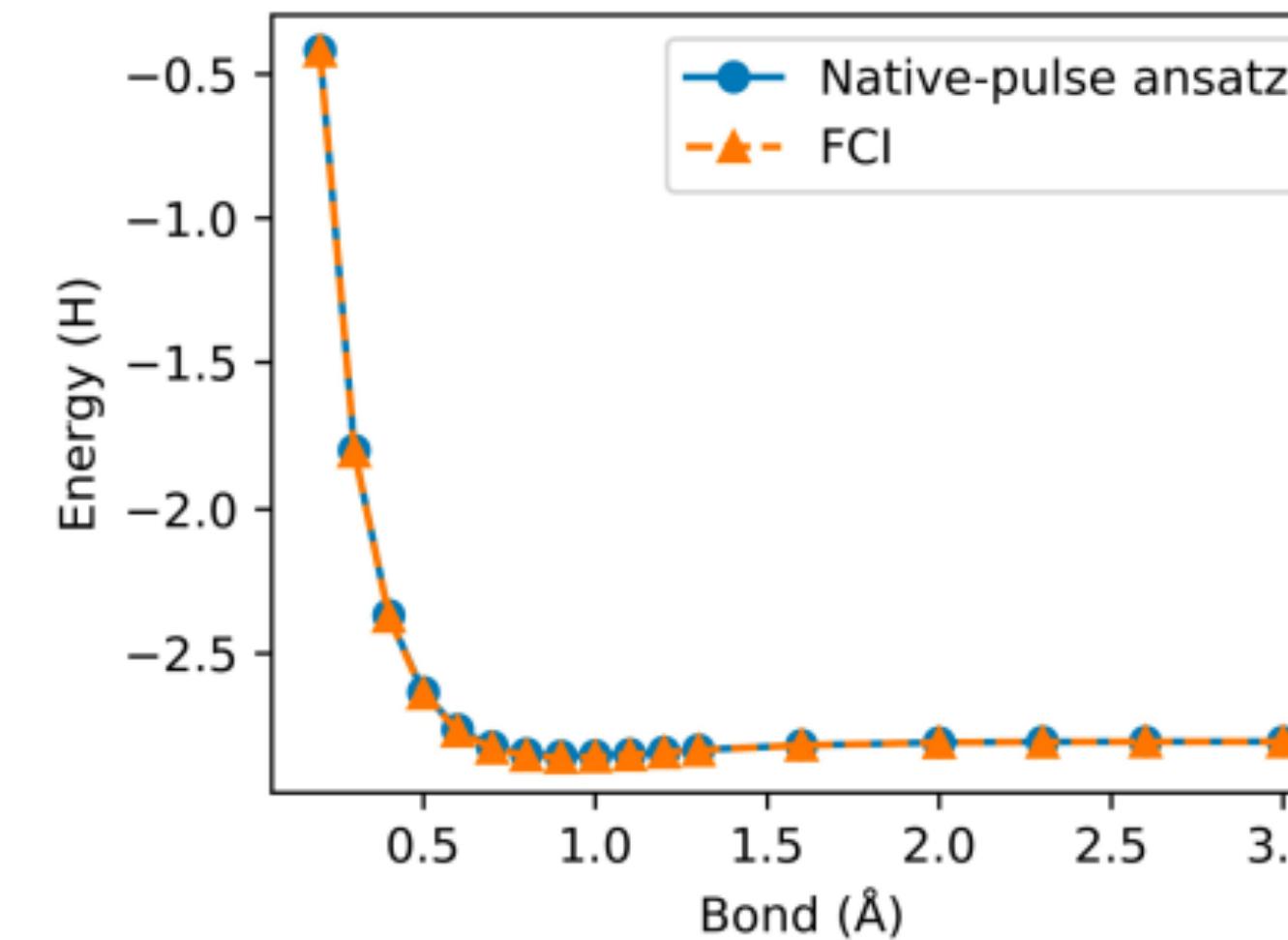
COMPARISON OF DURATION, PULSE COUNTS, AND ESTIMATED ENERGY OF GATE ANSATZES AND THE NATIVE-PULSE ANSATZ GENERATED BY PAN ON NISQ MACHINES.

Model	Ansatz Level	Qubits	Duration	Single-Qubit Pulse Count	Multi-Qubit Pulse Count	Molecule	Energy	Reference Energy
Random Generated Ansatz	Gate Ansatz	2	682.7ns	16	2	$H_2$	-0.853	-1.137
RealAmplitude Ansatz [2]	Gate Ansatz	2	376.9ns	12	1	$H_2$	-0.974	-1.137
QuantumNAS [75]	Gate Ansatz	2	682.7ns	16	2	$H_2$	-1.033	-1.137
PAN	Pulse Ansatz	2	71.1ns	3	0	$H_2$	-1.100	-1.137
RealAmplitude Ansatz	Gate Ansatz	2	753.8ns	24	2	$HeH+$	-2.691	-2.863
PAN	Pulse Ansatz	2	199.1ns	1	1	$HeH+$	-2.866	-2.863
QuantumNAS	Gate Ansatz	6	7296.0ns	40	12	$LiH$	-6.914	-7.882
PAN	Pulse Ansatz	4	199.1ns	4	2	$LiH$	-7.590	-7.882

Simulation results for  $H_2$



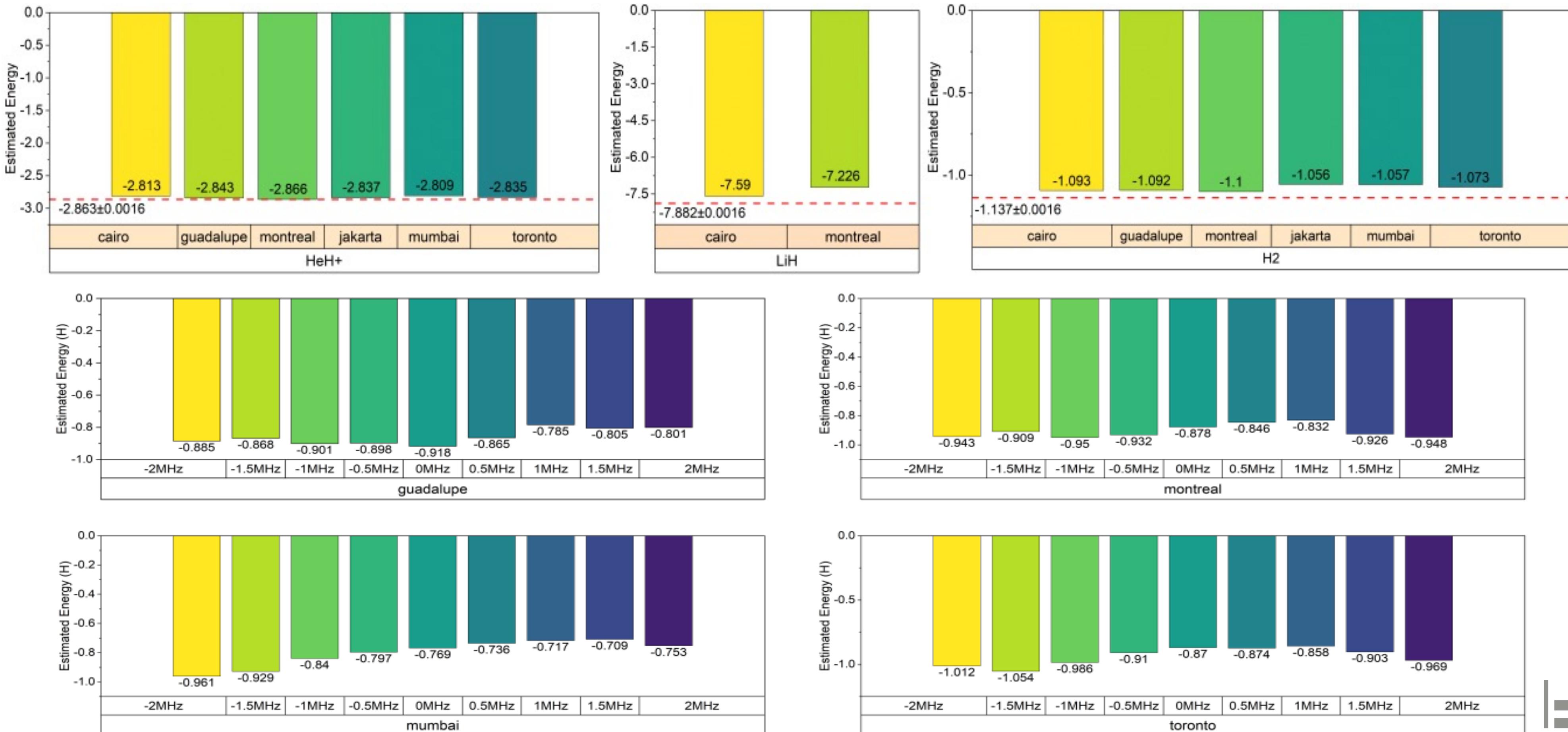
Simulation results for  $HeH^+$



**97.3% reduction in ansatz duration compared to QuantumNAS.**  
**Reduce duration by 73.6% compared to RealAmplitude Ansatz while maintaining ansatz performance.**

# Framework and Evaluation

- **HeH<sup>+</sup>**: average accuracy **99.336%**, with **99.895%** being the highest achievable accuracy. The absolute difference in energy is **0.003H**, close to the requirement of computational chemistry error (**0.0016H**) with only a toy model.

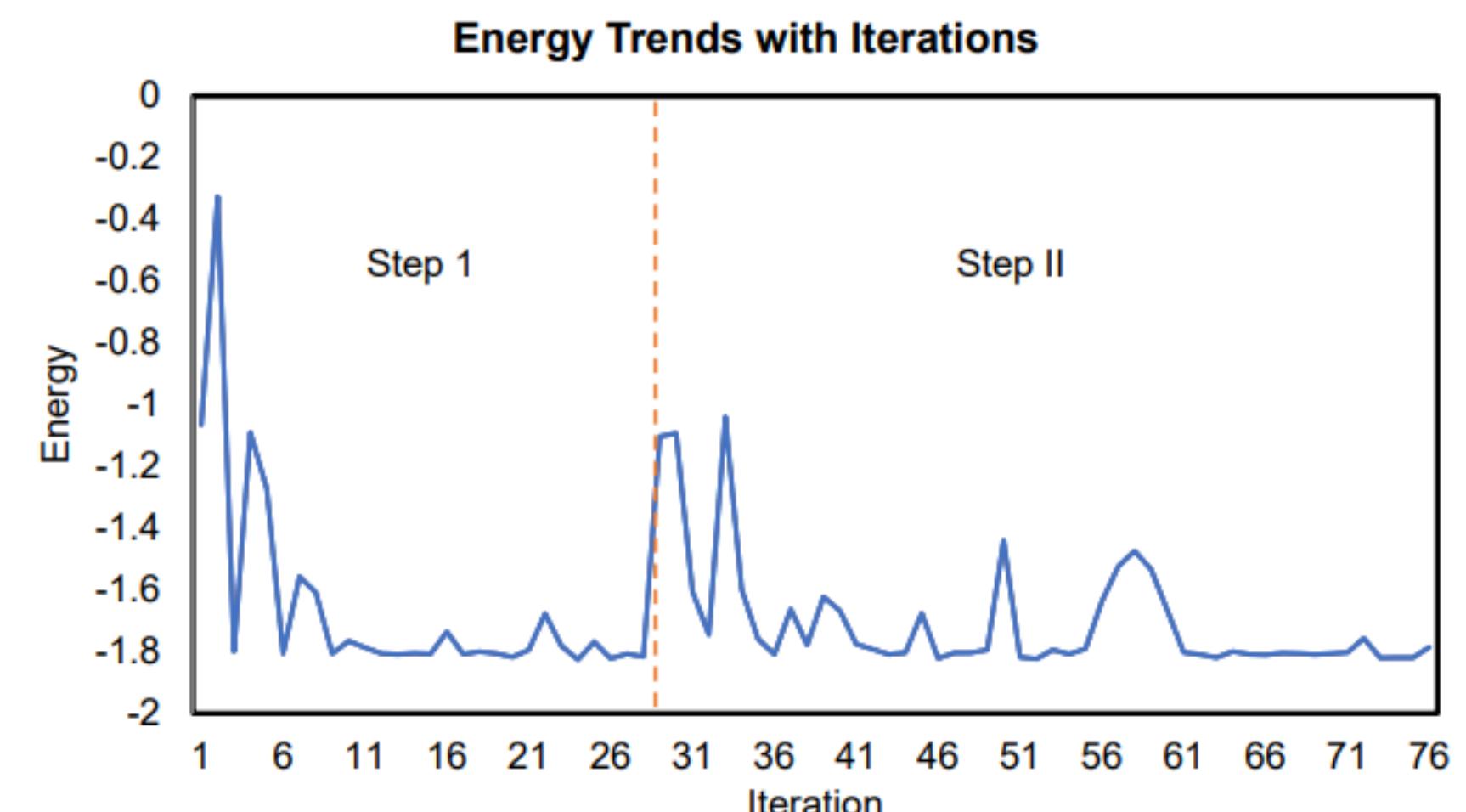


# Framework and Evaluation

- Verify the effectiveness of progressive learning scheme.

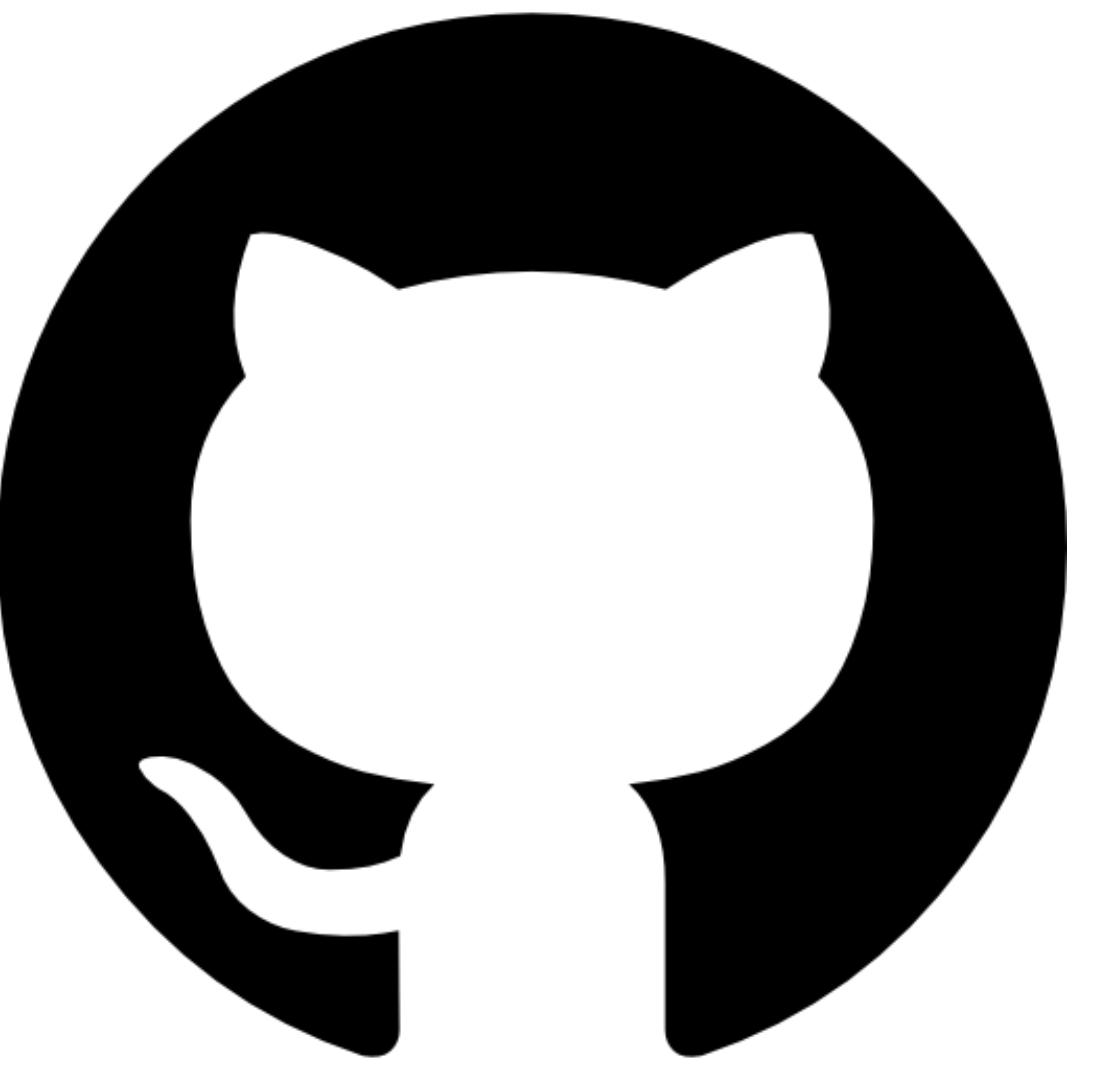
RESULTS OF ESTIMATED ENERGY FOR MOLECULES IN DIFFERENT STEPS

	Model	Cairo	Montreal	Toronto	NISQ machine Avg	Simulator	FCI
$H_2$	Step I	-1.093 (3.870%)	-1.087 (4.398%)	-1.073 (5.629%)	-1.084 (4.661%)	-1.121 (1.407%)	-1.137
	Step II	-1.107 (2.639%)	-1.110 (2.375%)	-1.073 (5.629%)	-1.097 (3.518%)	-1.123 (1.231%)	-1.137
	Inaccuracy Reduction	31.83%	46.00%	0.000%	24.52%	12.51%	-
$HeH^+$	Step I	-2.813 (1.746%)	-2.845 (0.663%)	-2.820 (1.485%)	-2.826 (1.292%)	-2.855 (0.279%)	-2.863
	Step II	-2.833 (1.047%)	-2.866 (0.105%)	-2.834 (1.013%)	-2.844 (0.664%)	-2.856 (0.244%)	-2.863
	Inaccuracy Reduction	40.03%	84.16%	31.78%	48.61%	12.54%	-



# Hands-On Section

## 3.1 Quantum Optimal Control

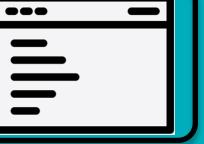


# Summary

## Section 1

### TorchQuantum Basic Usage

1.1 Quantum Basics

1.2 TQ operations 

1.3 TQ for State Prep 

1.4 TQ for VQE 

1.5 TQ for QNN 

## Section 2

### Use TorchQuantum on Gate level

2.1 QuantumNAS: Ansatz Search and Gate Pruning 

2.2 QuantumNAT: Noise Injection and Quantization 

2.3 QOC: On-Chip Training 

2.4 Transformer for Quantum Circuit Reliability Prediction

2.5 QNN Compression 

## Section 3

### Use TorchQuantum on Pulse level

3.1 Quantum Optimal Control 

3.2 Variational Pulse Learning 

# Thank you for listening!



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



[qmlsys.mit.edu](http://qmlsys.mit.edu)