TorchQuantum + cuQuantum Collaboration

Hanrui Wang, Song Han

Outline

- Overview
- TorchQuantum Introduction
- TorchQuantum Implementation
- TorchQuantum Interface
- TorchQuantum Examples
- TorchQuantum cuQuantum Collaboration
- QuantumNAS

Quantum Computer System Stack

High Level Framework

cuQuantum

Torch Quantum

Qiskit

Paddle Quantum

SILQ

BQSKit

Cirq

幺

Intermediate Representation

Quil

OpenQASM

Blackbird

QMASM

Classical Simulation

Real Hardware Deployment

Runtime / Control Signal Generation

cuDNN

OpenMP

Amazon Braket

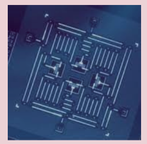
IBM Q

IONQ

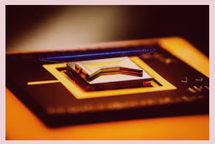
Device

GPU

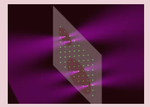
CPU



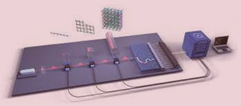
Superconducting



Trapped Ion



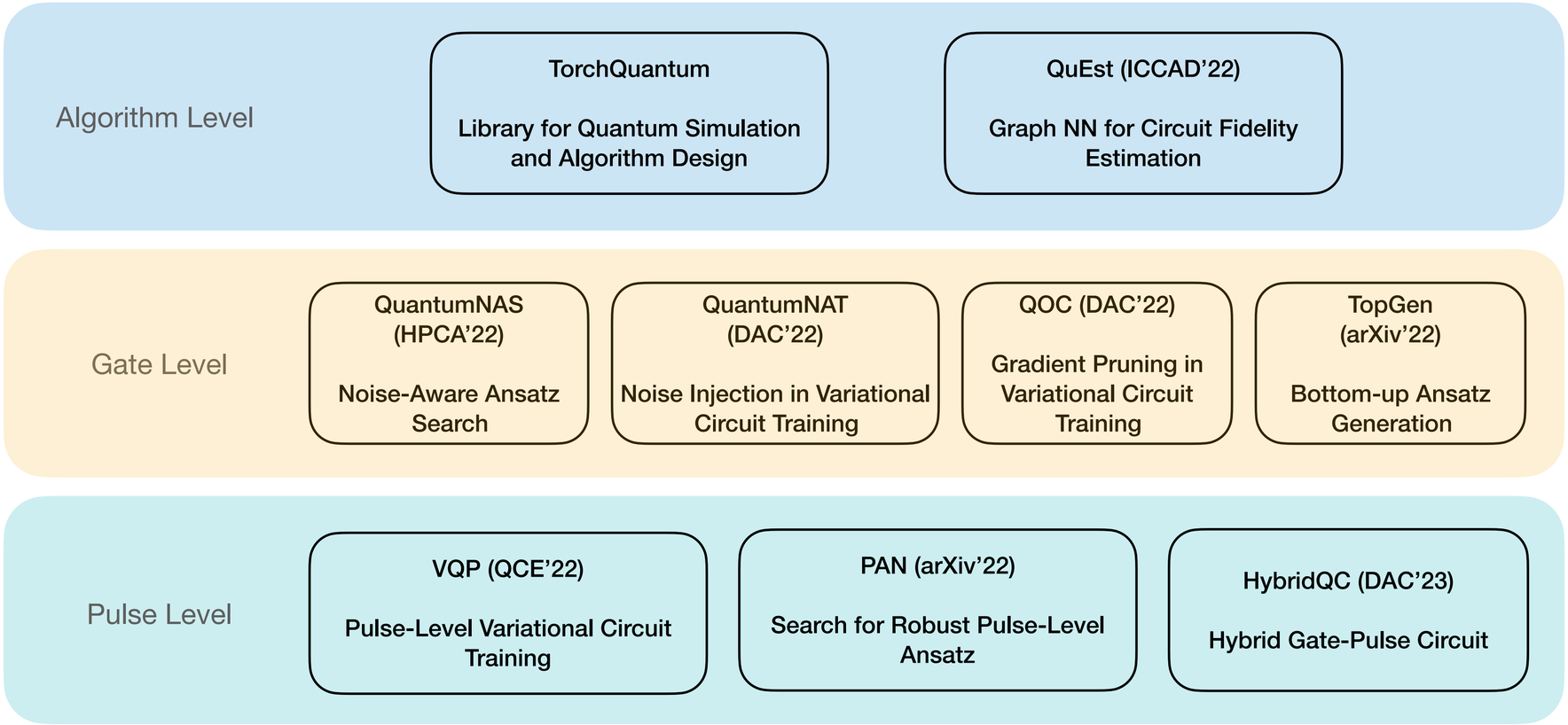
Neutral Atom



Photonics

Quantum Architecture Research at HAN Lab

- At HAN Lab, we focus on AI-assisted cross-layer codesign for efficient and practical quantum computer system



TorchQuantum

- Classical simulation of quantum circuit with PyTorch interface
  - For algorithm design and verification
  - Fast, PyTorch native, portable, scalable
  - Analyze circuit behavior
  - Study Noise impact
  - Develop ML model for quantum optimization

TorchQuantum

- Classical simulation of quantum circuit with PyTorch interface
  - Automatic **gradient computation** for training parameterized quantum circuit
  - **GPU**-accelerated tensor processing with **batch** mode support
  - **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 and OpenQASM

Implementation

- Statevector simulator

```
_state = torch.zeros(2**self.n_wires, dtype=C_DTYPE)
_state[0] = 1 + 0j  # type: ignore
_state = torch.reshape(_state, [2] * self.n_wires).to(self.device)
self.register_buffer("state", _state)

repeat_times = [bsz] + [1] * len(self.state.shape)  # type: ignore
self._states = self.state.repeat(*repeat_times)  # type: ignore
self.register_buffer("states", self._states)
```

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

```
mat = mat.type(C_DTYPE).to(state.device)

devices_dims = [w + 1 for w in device_wires]
permute_to = list(range(state.dim()))
for d in sorted(devices_dims, reverse=True):
    del permute_to[d]
permute_to = permute_to[:1] + devices_dims + permute_to[1:]
permute_back = list(np.argsort(permute_to))
original_shape = state.shape
permuted = state.permute(permute_to).reshape([original_shape[0], mat.shape[-1], -1])

if len(mat.shape) > 2:
    # both matrix and state are in batch mode
    new_state = mat.bmm(permuted)
else:
    # matrix no batch, state in batch mode
    bsz = permuted.shape[0]
    expand_shape = [bsz] + list(mat.shape)
    new_state = mat.expand(expand_shape).bmm(permuted)

new_state = new_state.view(original_shape).permute(permute_back)
```

Implementation

- Density matrix simulator

```
    """
    _matrix = torch.zeros(2 ** (2 * self.n_wires), dtype=C_DTYPE)
    _matrix[0] = 1 + 0j
    _matrix = torch.reshape(_matrix, [2] * (2 * self.n_wires))
    self.register_buffer("matrix", _matrix)

    repeat_times = [bsz] + [1] * len(self.matrix.shape)
    self._matrix = self.matrix.repeat(*repeat_times)
    self.register_buffer("matrix", self._matrix)
```

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

- Body Level One

```
device_wires = wires
n_qubit = int((density.dim() - 1) / 2)

mat = mat.type(C_DTYPE).to(density.device)      yezhuoyang, 5 months ago •
"""
Compute U \rho
"""
devices_dims = [w + 1 for w in device_wires]
permute_to = list(range(density.dim()))
for d in sorted(devices_dims, reverse=True):
    del permute_to[d]
permute_to = permute_to[:1] + devices_dims + permute_to[1:]
permute_back = list(np.argsort(permute_to))
original_shape = density.shape
permuted = density.permute(permute_to).reshape(
    [original_shape[0], mat.shape[-1], -1]
)
if len(mat.shape) > 2:
    # both matrix and state are in batch mode
    new_density = mat.bmm(permuted)
else:
    # matrix no batch, state in batch mode
    bsz = permuted.shape[0]
    expand_shape = [bsz] + list(mat.shape)
    new_density = mat.expand(expand_shape).bmm(permuted)
new_density = new_density.view(original_shape).permute(permute_back)
"""
Compute U*rho*U^\dagger
"""
devices_dims = [w + 1 + n_qubit for w in device_wires]
permute_to = list(range(density.dim()))
for d in sorted(devices_dims, reverse=True):
    del permute_to[d]
permute_to = permute_to + devices_dims
permute_back = list(np.argsort(permute_to))
original_shape = density.shape
permuted = new_density.permute(permute_to).reshape(
    [original_shape[0], -1, mat.shape[-1]]
)
if len(mat.shape) > 2:
    # both matrix and state are in batch mode
    # matdag is the dagger of mat
    matdag = torch.conj(mat.permute([0, 2, 1]))
    new_density = permuted.bmm(matdag)
else:
    # matrix no batch, state in batch mode
    matdag = torch.conj(mat.permute([1, 0]))
    bsz = permuted.shape[0]
    expand_shape = [bsz] + list(matdag.shape)
    new_density = permuted.bmm(matdag.expand(expand_shape))
new_density = new_density.view(original_shape).permute(permute_back)
return new_density
```

Interfaces

- import torchquantum as tq
- import torchquantum.functional as tqf

- qdev = tq.QuantumDevice(n_wires=2, bsz=5, device="cpu", record_op=True) # use device='cuda' for GPU

- # use qdev.op
- qdev.h(wires=0)
- qdev.cnot(wires=[0, 1])

- Body Level One

Interfaces

- # use tqf
- tqf.h(qdev, wires=1)
- tqf.x(qdev, wires=1)

- # use tq.Operator
- op = tq.RX(has_params=True, trainable=True, init_params=0.5)
- op(qdev, wires=0)

- # print the current state (dynamic computation graph supported)
- print(qdev)

- # obtain gradients of expval w.r.t. trainable parameters
- expval[0].backward()
- print(op.params.grad)

- Body Level One

Interfaces

- # obtain the qasm string
- from torchquantum.plugins import op_history2qasm
- print(op_history2qasm(qdev.n_wires, qdev.op_history))

- # measure the state on z basis
- print(tq.measure(qdev, n_shots=1024))

- # obtain the expval on a observable
- from torchquantum.measurement import expval_joint_analytical
- expval = expval_joint_analytical(qdev, 'ZX')
- print(expval)

- Body Level One

Features

- Dynamic computation graph
- GPU support, batch mode processing

```
>>> import torchquantum as tq

>>> import torchquantum.functional as tqf
>>> qdev = tq.QuantumDevice(n_wires=2, bsz=5, device="cpu", record_op=True) # use device='cuda' for GPU
>>>
>>> qdev.h(wires=0)
>>> qdev.cnot(wires=[0, 1])
>>> qdev
 class: QuantumDevice
 device name: default
 number of qubits: 2
 batch size: 5
 current computing device: cpu
 recording op history: True
 current states: [[0.70710677+0.j 0.        +0.j 0.        +0.j 0.70710677+0.j]
 [0.70710677+0.j 0.        +0.j 0.        +0.j 0.70710677+0.j]
 [0.70710677+0.j 0.        +0.j 0.        +0.j 0.70710677+0.j]
 [0.70710677+0.j 0.        +0.j 0.        +0.j 0.70710677+0.j]
 [0.70710677+0.j 0.        +0.j 0.        +0.j 0.70710677+0.j]]
```

- Body Level One

Features

- Converters to OpenQASM, Qiskit etc

```
>>> op = tq.RX(has_params=True, trainable=True, init_params=0.5)
>>> op(qdev, wires=0)
>>> from torchquantum.plugins import op_history2qasm
>>> print(op_history2qasm(qdev.n_wires, qdev.op_history))
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
h q[0];
cx q[0],q[1];
rx(0.5) q[0];

>>>
```

- Body Level One

Features

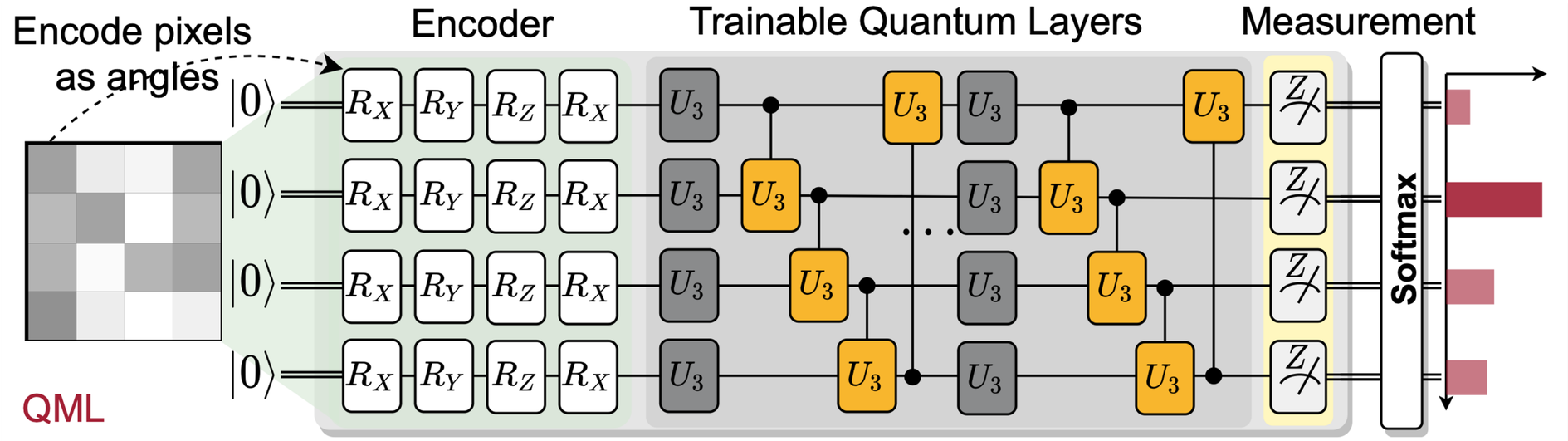
- Automatic gradient computation

```
>>> # obtain the expval on a observable
>>> from torchquantum.measurement import expval_joint_analytical
>>> expval = expval_joint_analytical(qdev, 'ZX')
>>> print(expval)
tensor([0.8776, 0.8776, 0.8776, 0.8776, 0.8776], grad_fn=<SelectBackward0>)
>>>
>>> # obtain gradients of expval w.r.t. trainable parameters
>>> expval[0].backward()
>>> print(op.params.grad)
tensor([[-0.4794]])
```

- Body Level One

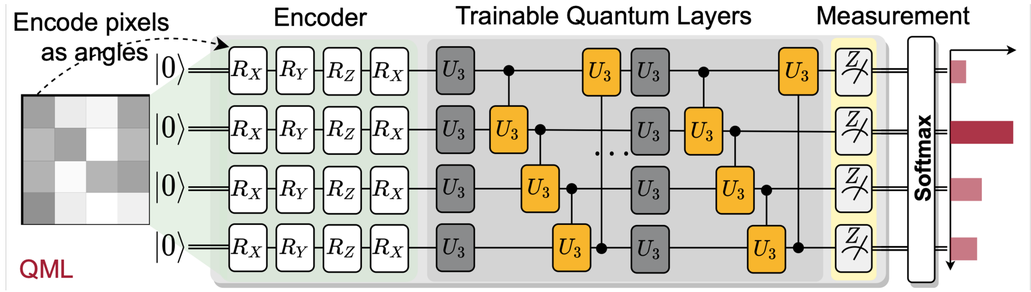
Example

- MNIST classification



Example

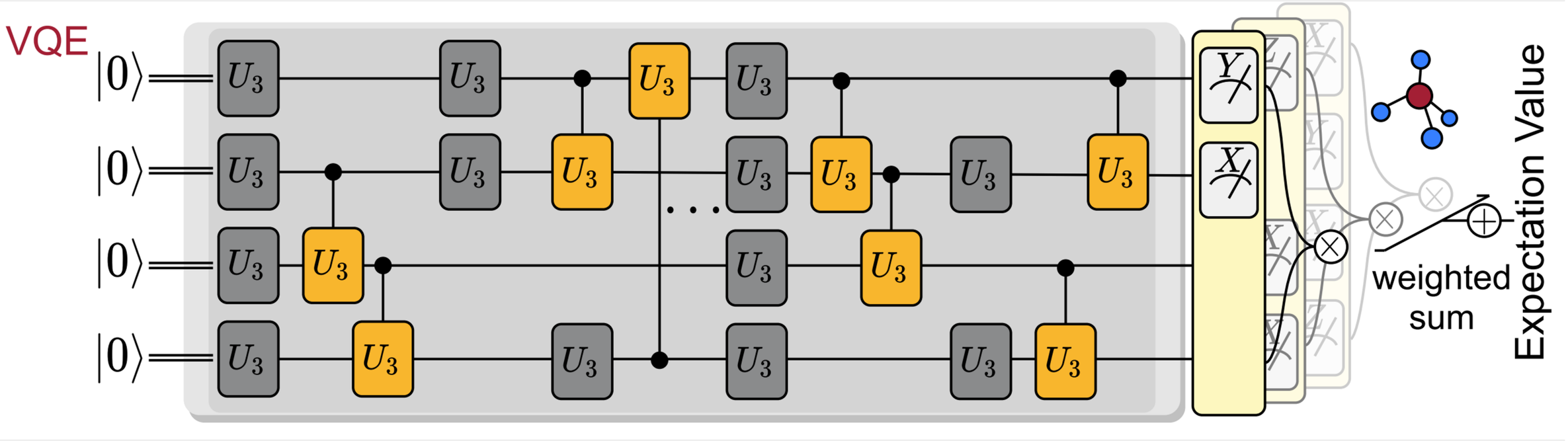
```python
def __init__(self):
    super().__init__()
    self.n_wires = 4
    self.encoder = tq.GeneralEncoder(tq.encoder_op_list_name_dict["4x4_u3rx"])

    self.q_layer = self.QLayer()
    self.measure = tq.MeasureAll(tq.PauliZ)

def forward(self, x, use_qiskit=False):
    qdev = tq.QuantumDevice(
        n_wires=self.n_wires, bsz=x.shape[0], device=x.device, record_op=True
    )

    bsz = x.shape[0]
    x = F.avg_pool2d(x, 6).view(bsz, 16)
    devi = x.device

    if use_qiskit:
        self.encoder(qdev, x)
        op_history_parameterized = qdev.op_history
        qdev.reset_op_history()
        encoder_circs = op_history2qiskit_expand_params(self.n_wires, op_history_parameterized, bsz=bsz)

        self.q_layer(qdev)
        op_history_fixed = qdev.op_history
        qdev.reset_op_history()
        q_layer_circ = op_history2qiskit(self.n_wires, op_history_fixed)

        measurement_circ = tq2qiskit_measurement(qdev, self.measure)
        assembled_circs = qiskit_assemble_circs(
            encoder_circs, q_layer_circ, measurement_circ
        )
        x0 = self.qiskit_processor.process_ready_circs(qdev, assembled_circs).to(
            devi
        )
        x = x0
    You, 2 days ago • [major] no binding mnist as default …
    else:
        self.encoder(qdev, x)
        op_history_parameterized = qdev.op_history
        qdev.reset_op_history()
        self.q_layer(qdev)
        op_history_fixed = qdev.op_history
        x = self.measure(qdev)

    x = x.reshape(bsz, 2, 2).sum(-1).squeeze()
    x = F.log_softmax(x, dim=1)

    return x
```




```python
class QLayer(tq.QuantumModule):
    def __init__(self):
        super().__init__()
        self.n_wires = 4
        self.random_layer = tq.RandomLayer(
            n_ops=50, wires=list(range(self.n_wires))
        )

        # gates with trainable parameters
        self.rx0 = tq.RX(has_params=True, trainable=True)
        self.ry0 = tq.RY(has_params=True, trainable=True)
        self.rz0 = tq.RZ(has_params=True, trainable=True)
        self.crx0 = tq.CRX(has_params=True, trainable=True)

    def forward(self, qdev: tq.QuantumDevice):
        self.random_layer(qdev)

        # some trainable gates (instantiated ahead of time)
        self.rx0(qdev, wires=0)
        self.ry0(qdev, wires=1)
        self.rz0(qdev, wires=3)
        self.crx0(qdev, wires=[0, 2])

        # add some more non-parameterized gates (add on-the-fly)
        qdev.h(wires=3)  # type: ignore
        qdev.sx(wires=2)  # type: ignore
        qdev.cnot(wires=[3, 0])  # type: ignore
        qdev.rx(
            wires=1,
            params=torch.tensor([0.1]),
            static=self.static_mode,
            parent_graph=self.graph,
        )  # type: ignore
```

Example

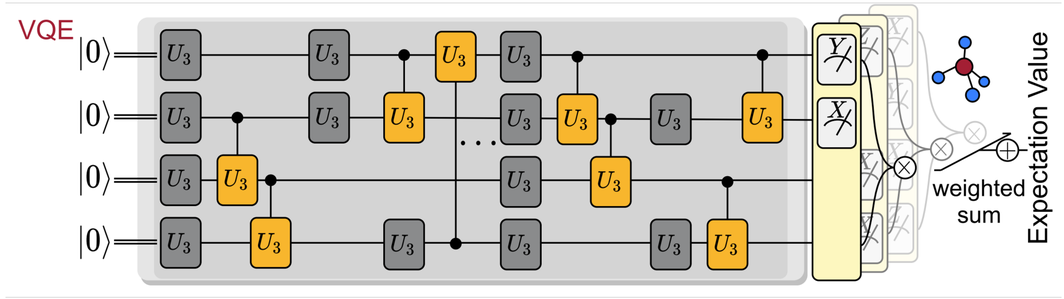
- Variational Quantum Eigensolver



Example

- MNIST classification

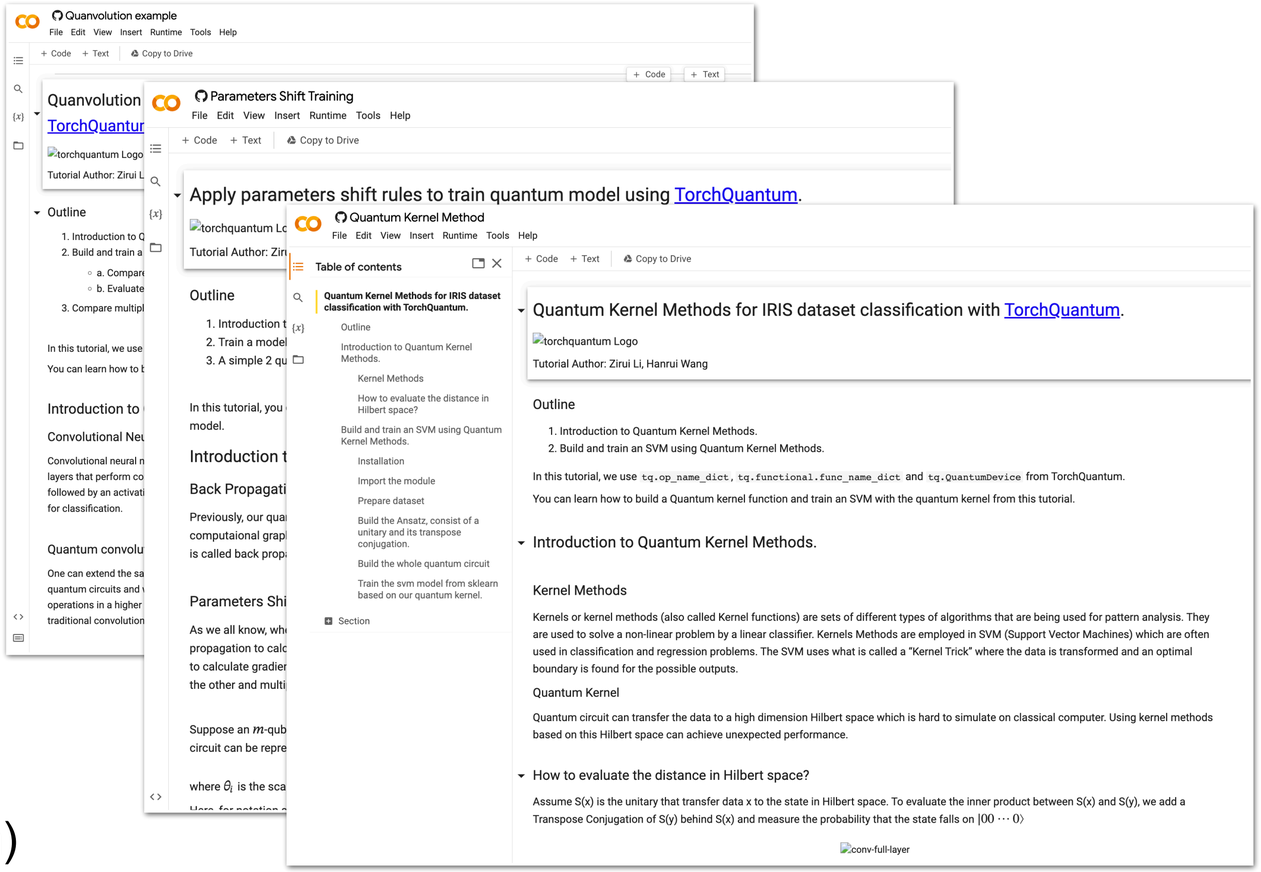
```python
class QVQEModel(tq.QuantumModule):
    def __init__(self, arch, hamil_info):
        super().__init__()
        self.arch = arch
        self.hamil_info = hamil_info
        self.n_wires = hamil_info["n_wires"]
        self.n_blocks = arch["n_blocks"]
        self.u3_layers = tq.QuantumModuleList()
        self.cu3_layers = tq.QuantumModuleList()
        for _ in range(self.n_blocks):
            self.u3_layers.append(
                tq.Op1QAllLayer(
                    op=tq.U3,
                    n_wires=self.n_wires,
                    has_params=True,
                    trainable=True,
                )
            )
            self.cu3_layers.append(
                tq.Op2QAllLayer(
                    op=tq.CU3,
                    n_wires=self.n_wires,
                    has_params=True,
                    trainable=True,
                    circular=True,
                )
            )
```




```python
    def forward(self):
        qdev = tq.QuantumDevice(
            n_wires=self.n_wires, bsz=1, device=next(self.parameters()).device
        )

        for k in range(self.n_blocks):
            self.u3_layers[k](qdev)
            self.cu3_layers[k](qdev)

        expval = 0
        for hamil in self.hamil_info["hamil_list"]:
            expval += (
                expval_joint_analytical(qdev, observable=hamil["pauli_string"])
                * hamil["coeff"]
            )

        return expval
```

More Examples

- QNN for MNIST
- Quantum Convolution (Quanvolution)
- Quantum Kernel Method
- Quantum Regression.
- Amplitude Encoding for MNIST
- Clifford gate QNN
- Parameter Shift on-chip Training
- VQA Gradient Pruning
- VQE
- VQA for State Prepration
- QAOA (Quantum Approximate Optimization Algorithm)



- Body Level One

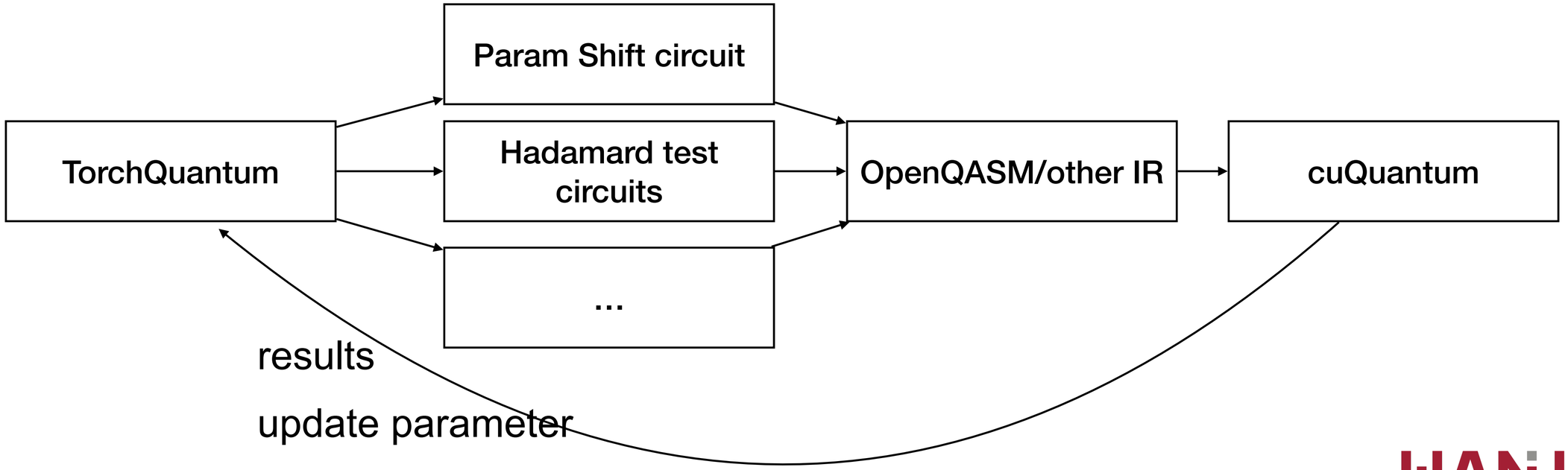
TorchQuantum cuQuantum collaboration

- TorchQuantum implements a statevector simulator using matrix multiplications with the cuDNN/ cuBLAS library as backend.
- Potential advantages of using cuQuantum as an alternative backend (cuTensorNet and cuStatevec)
  - cuQuantum supports tensor network contraction which brings higher scalability
    - Explore the sparsity nature of the state vector. Not all values in the statevector are equally important.
    - Statevector of N qubits can be thought as a rank-N tensor, that could be broken down into low-rank tensors.
    - cuTensorNet provides an automatic way to do the contraction
  - The cuStatevec has **multi-GPU** simulation support which also improves scalability

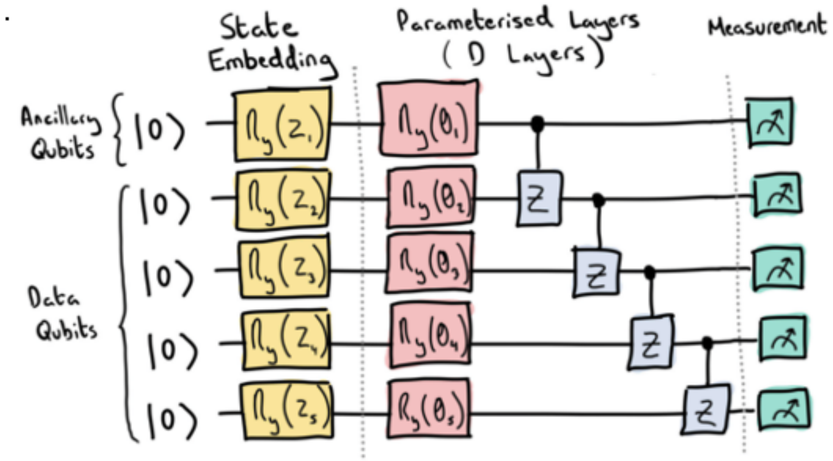
TorchQuantum cuQuantum collaboration

- For application with no need for parameter training

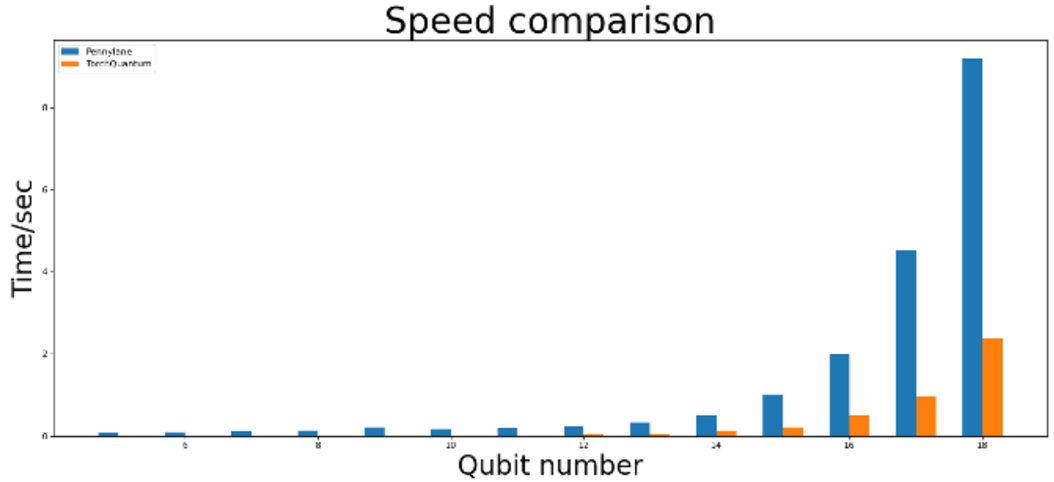
- For application need for parameter training

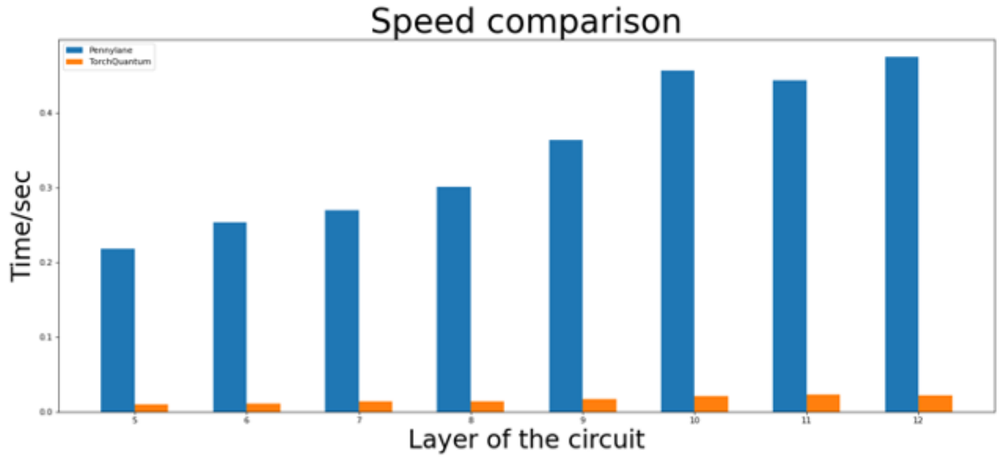


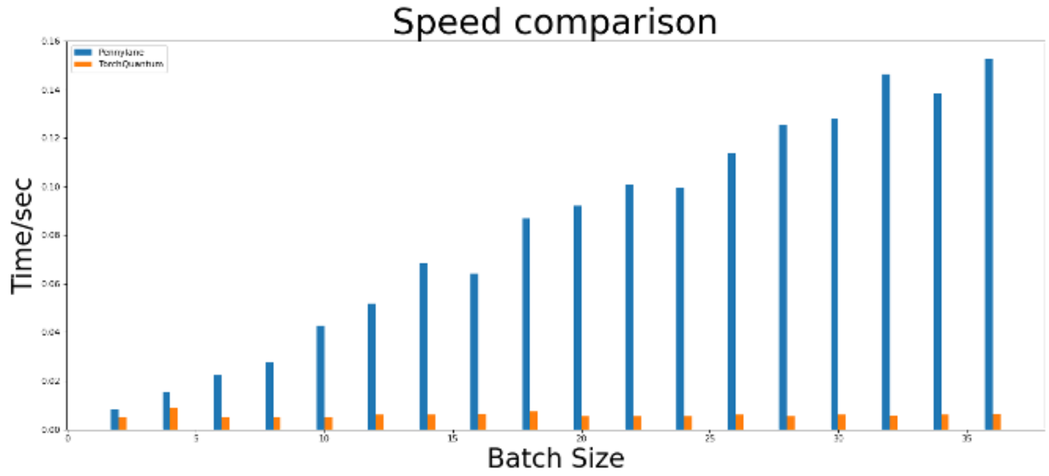
q



https://pennylane.ai/qml/_images/qcircuit.jpeg

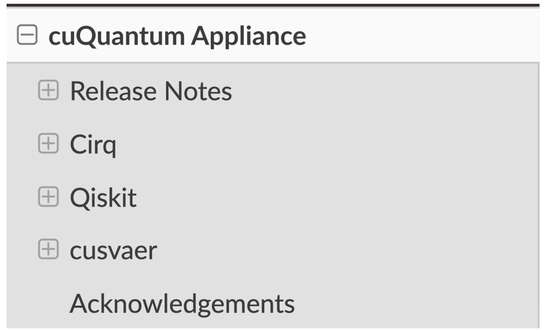






Integration of cuQuantum as backend of TorchQuantum

- Feasibility
  - cuQuantum has Python interfaces which can be wrapped into TorchQuantum. The circuit construction uses torchQuantum interface (PyTorch style). Then calling cuQuantum functions to perform state vector simulation or tensor network contraction.
- Format
  - cuQuantum can have one TorchQuantum Appliance



	When Building	When Running
Python	3.8+	3.8+
pip	22.3.1+	N/A
setuptools	>=61.0.0	N/A
wheel	>=0.34.0	N/A
Cython	>=0.29.22,<3	N/A
cuStateVec	1.2.0	~=1.1
cuTensorNet	2.0.0	~=2.0
NumPy	N/A	v1.19+
CuPy (see CuPy installation guide)	N/A	v9.5.0+
PyTorch (optional, see PyTorch installation guide)	N/A	v1.10+
Qiskit (optional, see Qiskit installation guide)	N/A	v0.24.0+
Cirq (optional, see Cirq installation guide)	N/A	v0.6.0+
mpi4py (optional, see mpi4py installation guide)	N/A	v3.1.0+

QuantumNAS: Noise-Adaptive Search for Robust Quantum Circuits

Hanrui Wang[1], Yongshan Ding[2], Jiaqi Gu[3], Yujun Lin[1],
David Z. Pan[3], Frederic T. Chong[4], Song Han[1]

[1]Massachusetts Institute of Technology, [2]Yale University, [3]University of Taxes at Austin,
[4]University of Chicago

Torch Quantum

HPCA 2022

MIT HAN LAB

Outline

- Overview
- Background
- QuantumNAS
- Evaluation
- TorchQuantum Library
- Conclusion

Outline

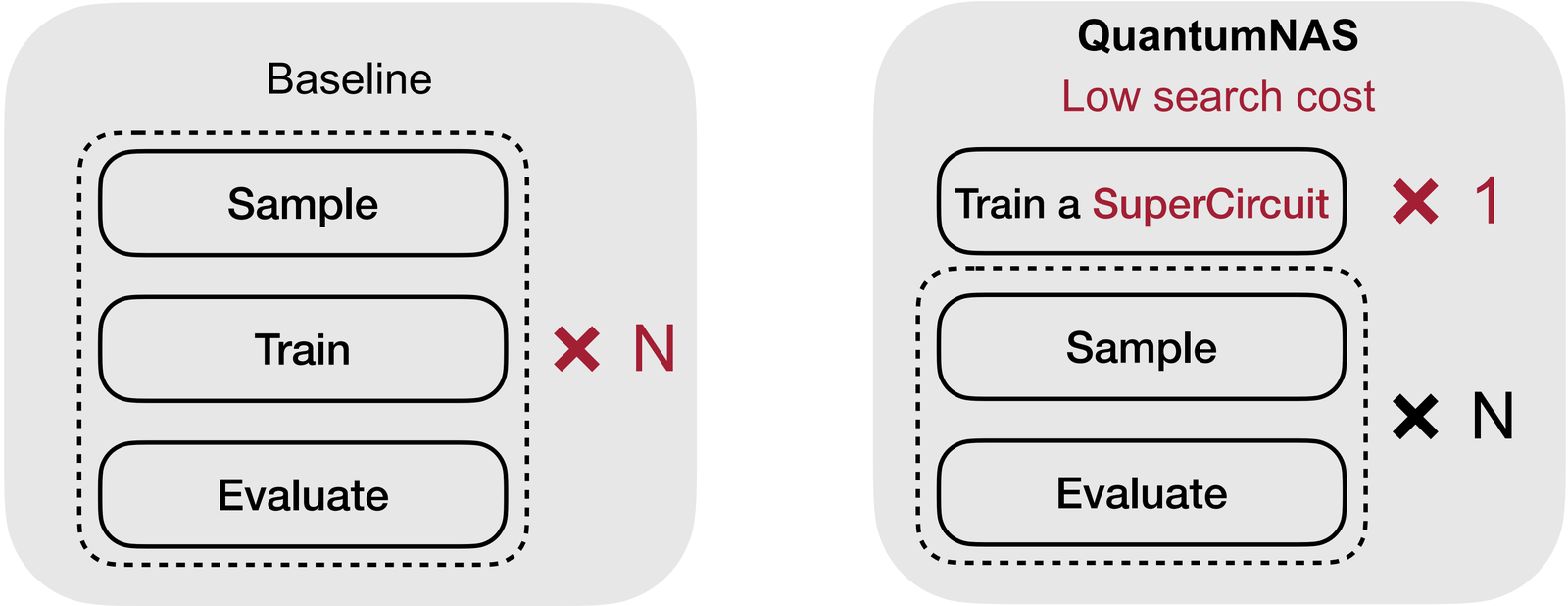
- Overview
- Background
- QuantumNAS
- Evaluation
- TorchQuantum Library
- Conclusion

Overview — QuantumNAS: Noise-Adaptive Search

- Quantum circuits are noisy
  - More gates: higher capacity, but also higher noise

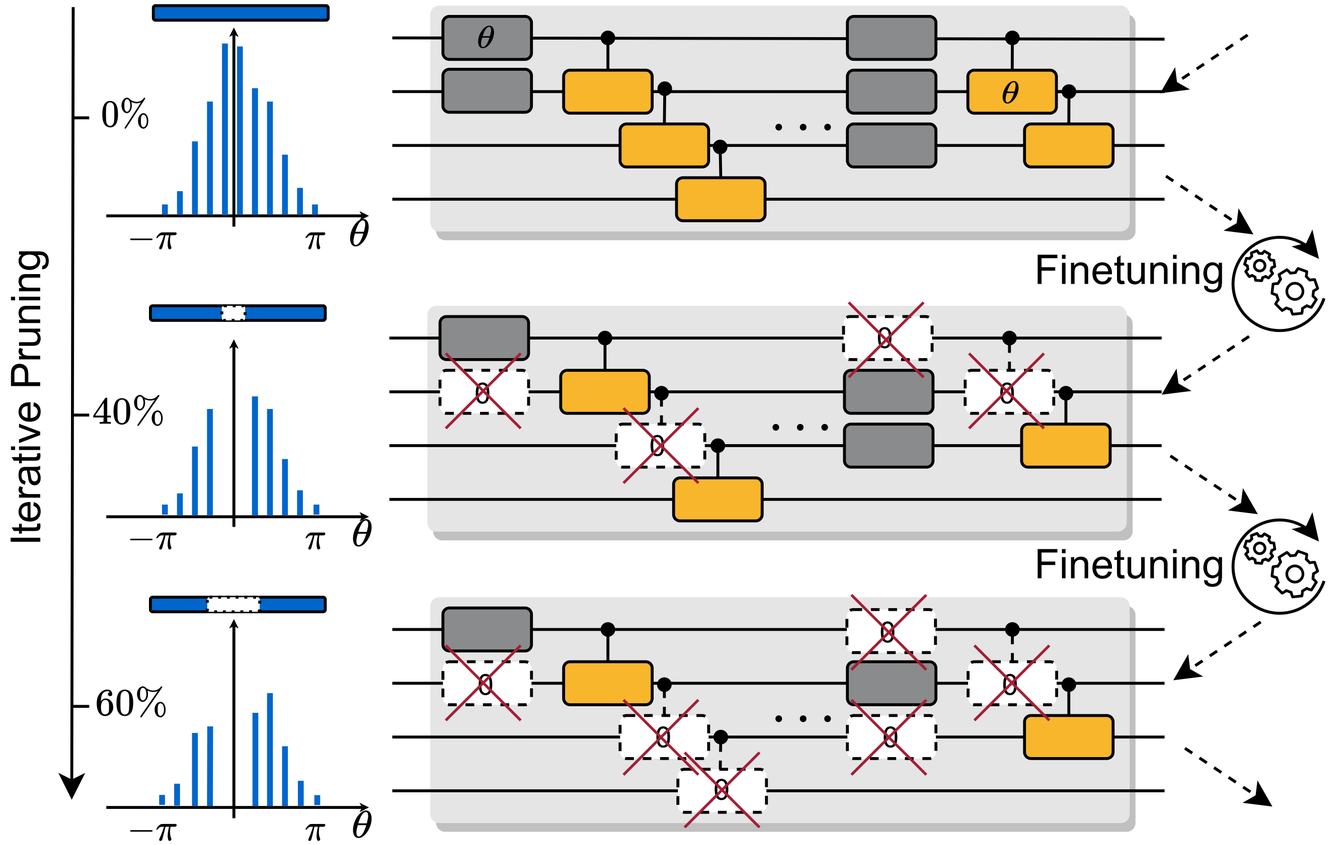
Overview — QuantumNAS: Noise-Adaptive Search

- Quantum circuits are noisy
  - More gates: higher capacity, but also higher noise
- Need to **search** for noise-robust circuit architecture
  - Naive search: train **each** possible circuit individually
  - QuantumNAS: train all circuits at **once,** amortize training cost



Overview — Iterative Quantum Gate Pruning

- Gates with small parameter have small impact on results
- **Iteratively prune** the gates with small parameters and fine-tune the remaining ones

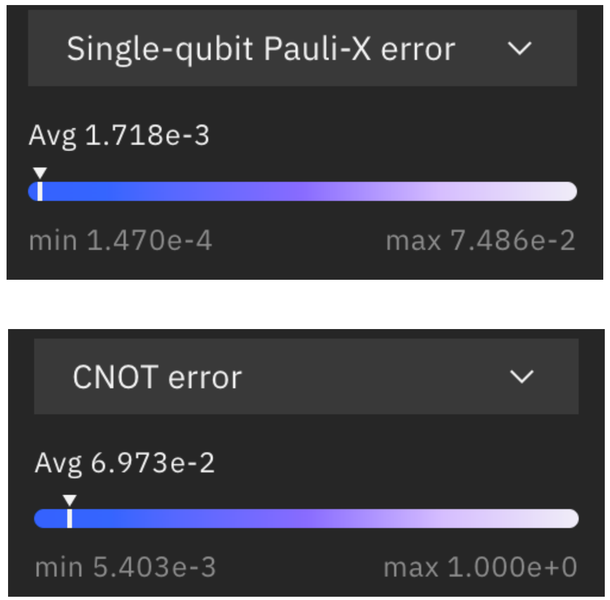


Outline

- Overview
- Background
- QuantumNAS
- Evaluation
- TorchQuantum Library
- Conclusion

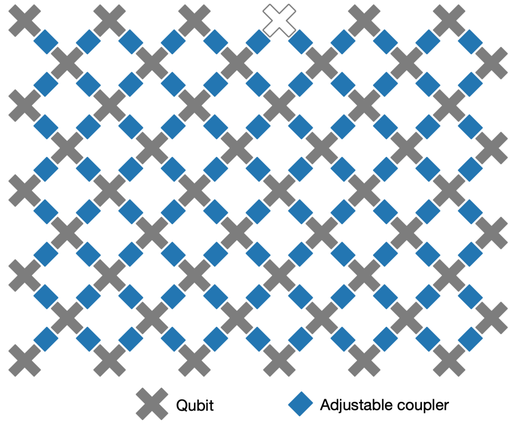
NISQ Era

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



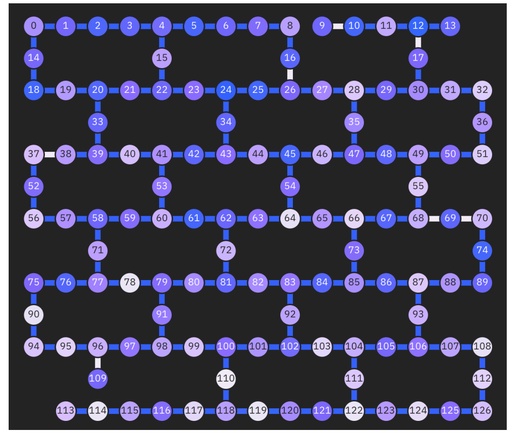
Gate Error Rate

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



Google Sycamore

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

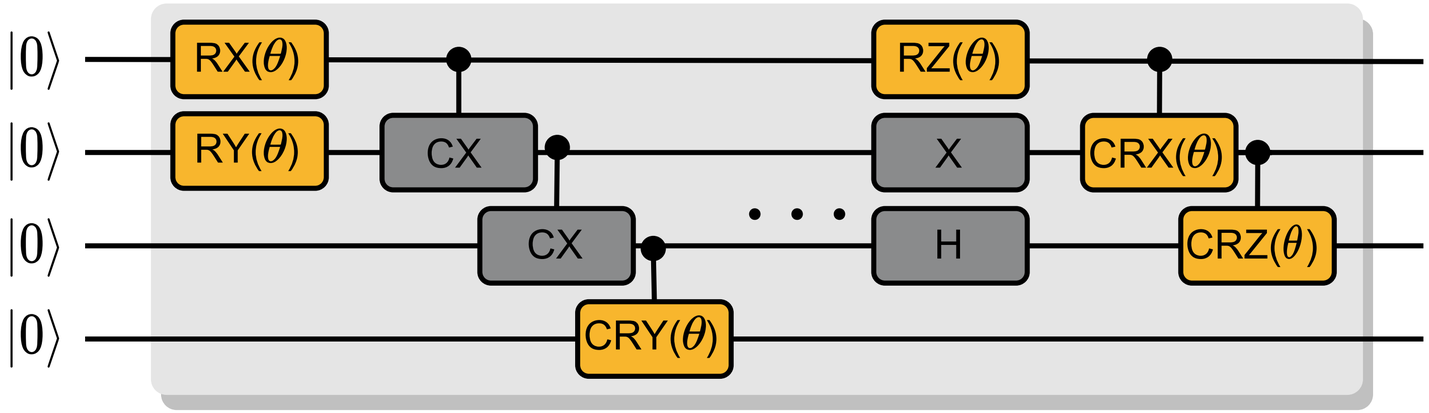


IBM Washington

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

Parameterized Quantum Circuits

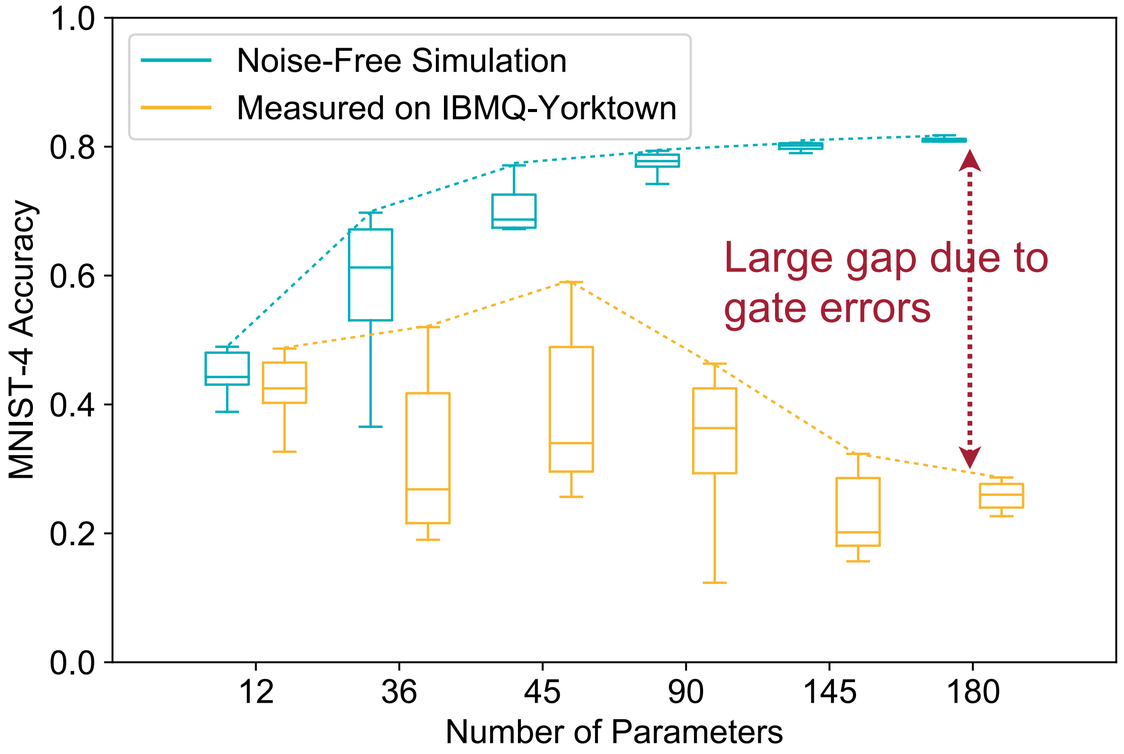
- Parameterized Quantum Circuits (PQC)
- Quantum circuit with fixed gates and parameterized gates



- PQCs are commonly used in **hybrid classical-quantum models** and show promises to achieve quantum advantage
  - Variational Quantum Eigensolver (VQE)
  - Quantum Neural Networks (QNN)
  - Quantum Approximate Optimization Algorithm (QAOA)

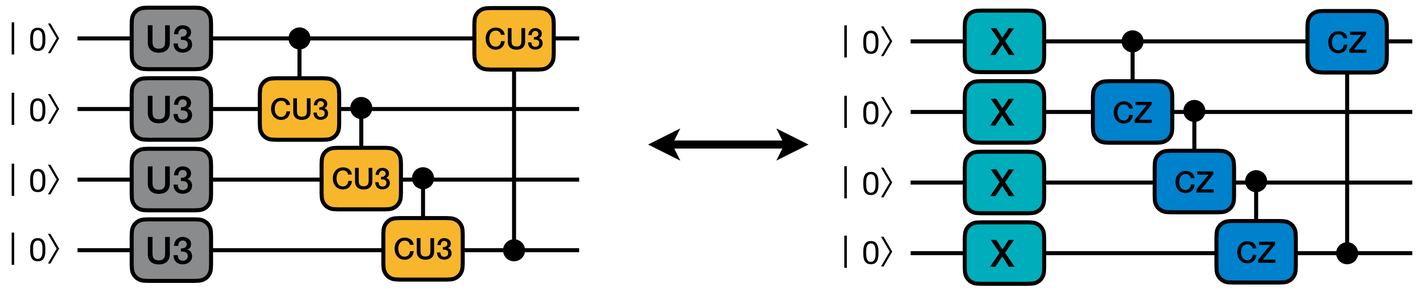
Challenges of PQC — Noise

- Noise **degrades** the Parameterized Quantum Circuit (PQC) reliability
- More parameters increase the noise-free accuracy but degrade the measured accuracy
- Under same #parameters, measured accuracy of different circuit architecture (ansatz) varies a lot
- Therefore, circuit architecture is critical



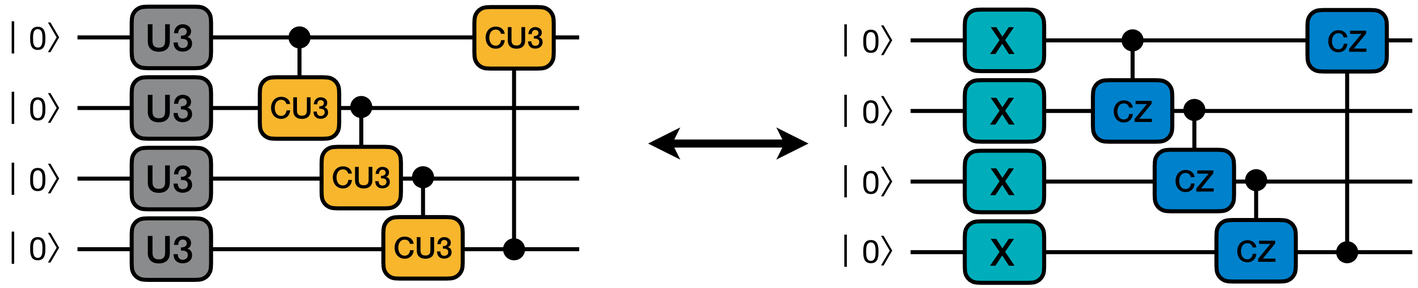
Challenges of PQC — Large Design Space

- Large design space for circuit architecture
  - Type of gates

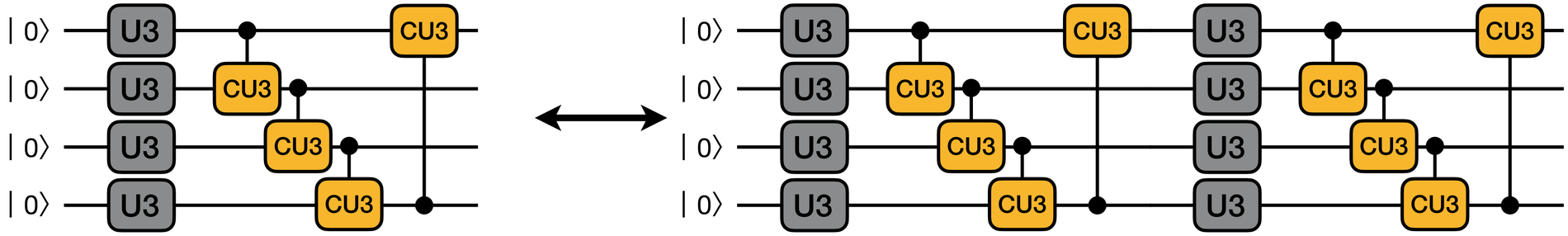


Challenges of PQC — Large Design Space

- Large design space for circuit architecture
  - Type of gates

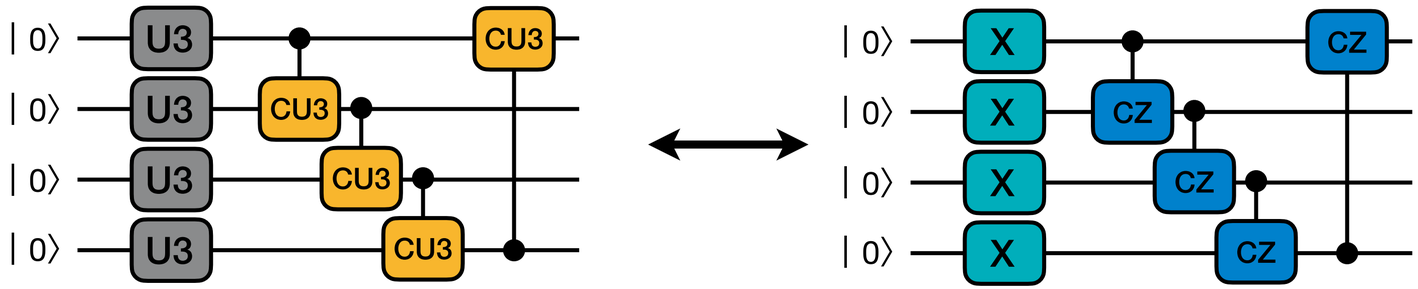


  - Number of gates

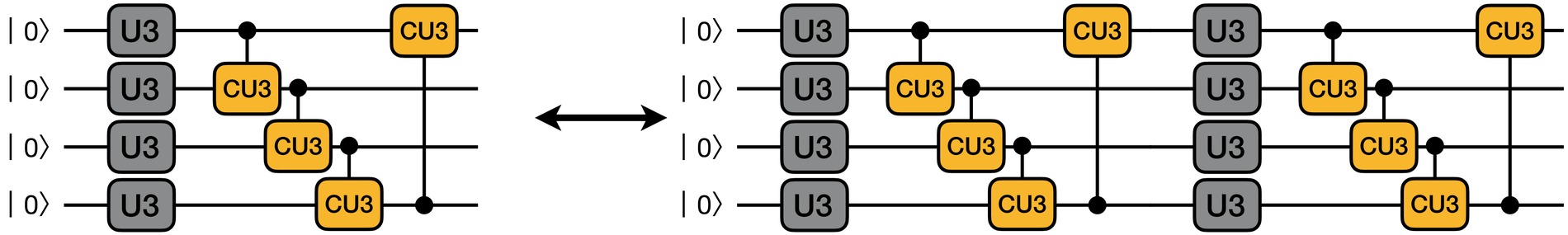


Challenges of PQC — Large Design Space

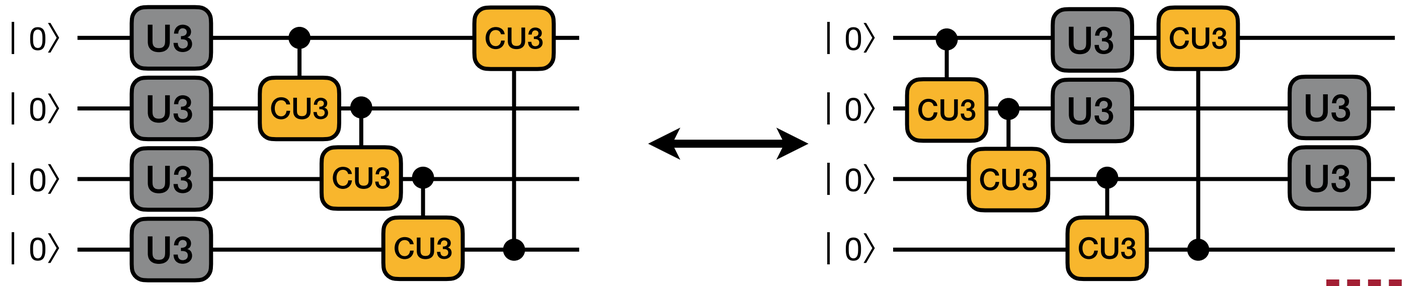
- Large design space for circuit architecture
  - Type of gates



  - Number of gates



  - Position of gates



Outline

- Overview
- Background
- QuantumNAS
- Evaluation
- TorchQuantum Library
- Conclusion

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: Decouple the Training and Search

Naive Search

For q_devices:
  For search episodes: // meta controller
    For circuit training iterations:
      update_parameters(); **Expensive**
    If good_circuit: **break;**

QuantumNAS: Decouple the Training and Search

Naive Search

For q_devices:
  For search episodes: // meta controller
    For circuit training iterations:
      update_parameters(); **Expensive**
    If good_circuit: **break;**

=>

QuantumNAS

For *SuperCircuit* training iterations: **Expensive**
  update_parameters(); **training**

decouple

For q_devices:
  For search episodes:
    sample from ***SuperCircuit***; **Light-Weight**
    If good_circuit: **break;**
    //no training

search

QuantumNAS

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

QuantumNAS

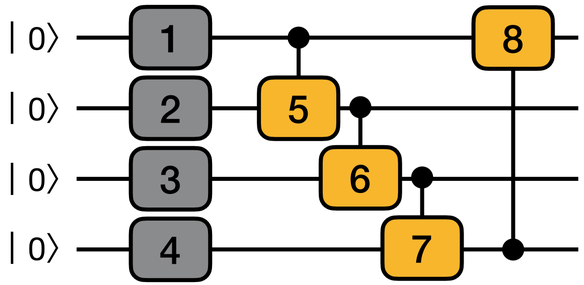
- SuperCircuit Construction and Training
- Noise-Adaptive Evolutionary Co-Search of SubCircuit and Qubit Mapping
- Train the Searched SubCircuit
- Iterative Quantum Gate Pruning

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
  - Contains 2^8 = 256 candidates

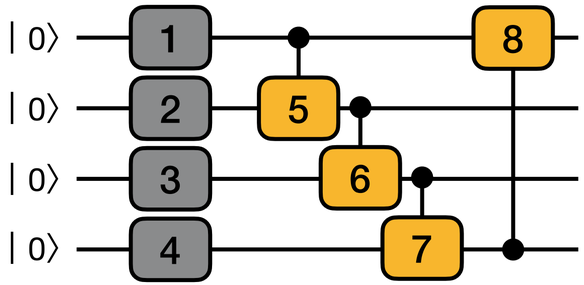
SuperCircuit & SubCircuit

- 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
  - Contains 2^8 = 256 candidates
- SuperCircuit: the circuit with the **largest** number of gates in the design space
  - Example: SuperCircuit in U3+CU3 space

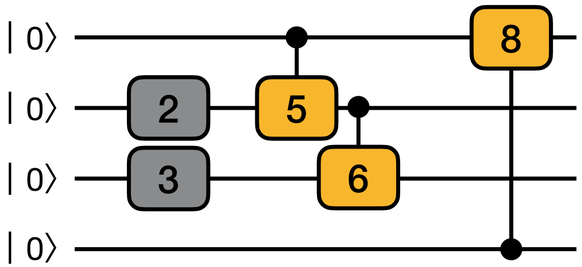


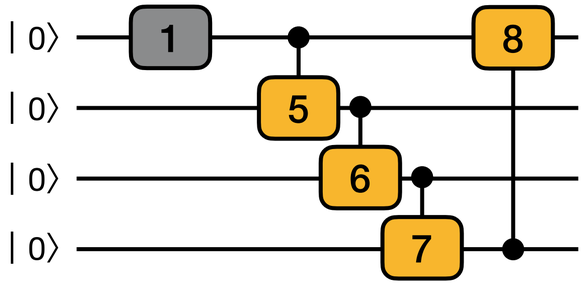
SuperCircuit & SubCircuit

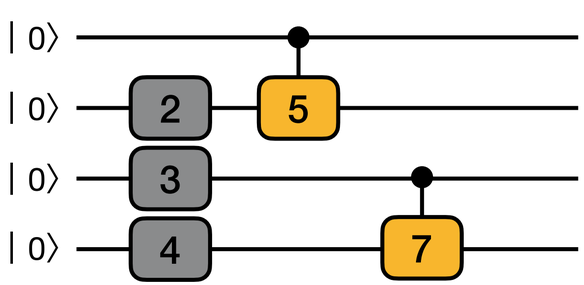
- Firstly construct a design space. For example, a design space of maximum 4 U3 in the first layer and 4 CU3 gates in the second layer
  - Contains 2^8 = 256 candidates
- SuperCircuit: the circuit with the **largest** number of gates in the design space
  - Example: SuperCircuit in U3+CU3 space



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







SuperCircuit Construction

- Why use a SuperCircuit?
  - It enables **efficient** search of circuit architecture candidates with no need of training each of them individually
  - For one SubCircuit candidate, we can directly inherit parameters from SuperCircuit and consider that the SubCircuit can operate **as if it is trained individually from scratch**

SuperCircuit Construction

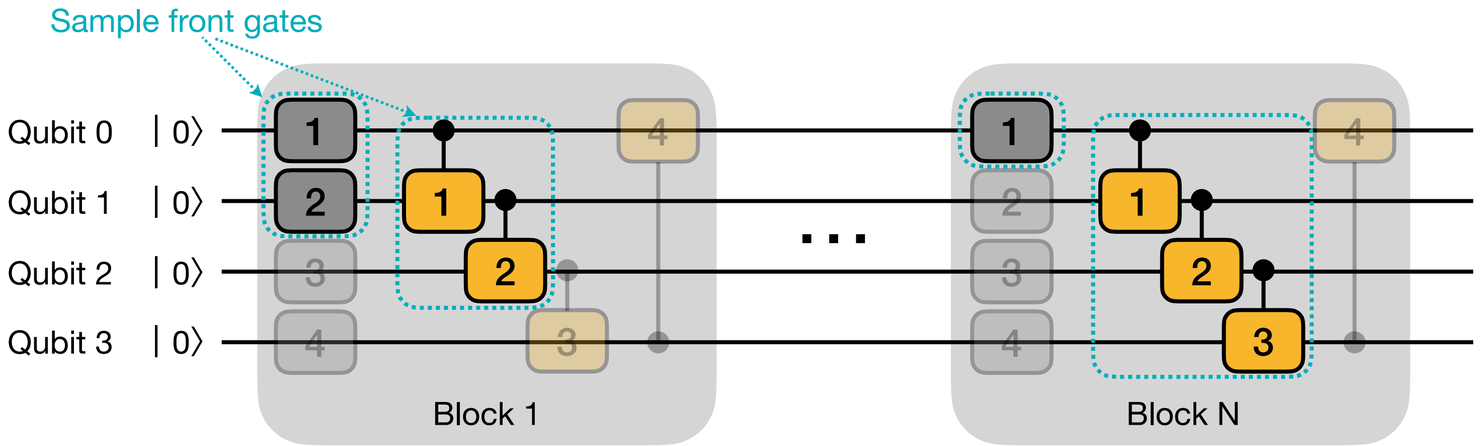
- Why use a SuperCircuit?
  - It enables **efficient** search of circuit architecture candidates with no need of training each of them individually
  - For one SubCircuit candidate, we can directly inherit parameters from SuperCircuit and consider that the SubCircuit can operate **as if it is trained individually from scratch**
- Need to prevent interference of SubCircuits from each other

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

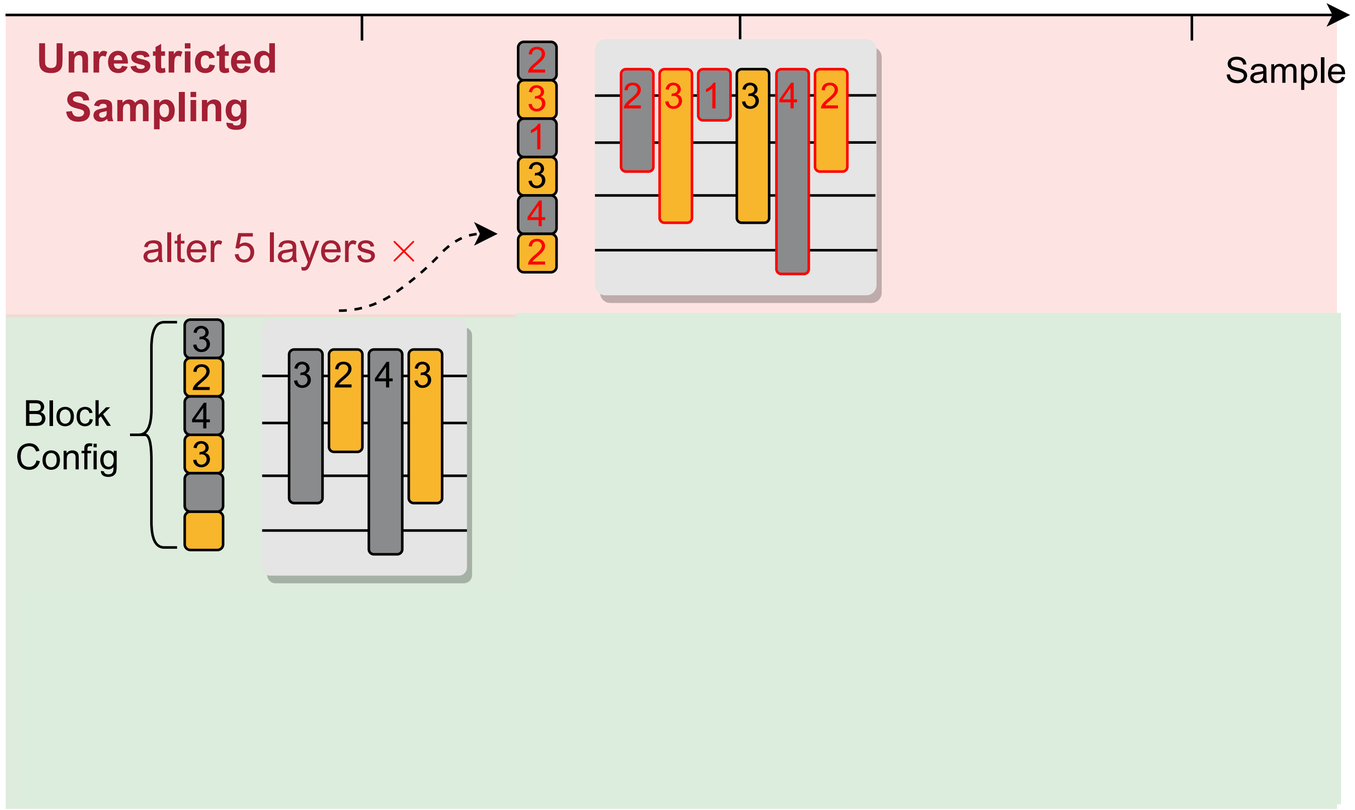
Front Sampling

- 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



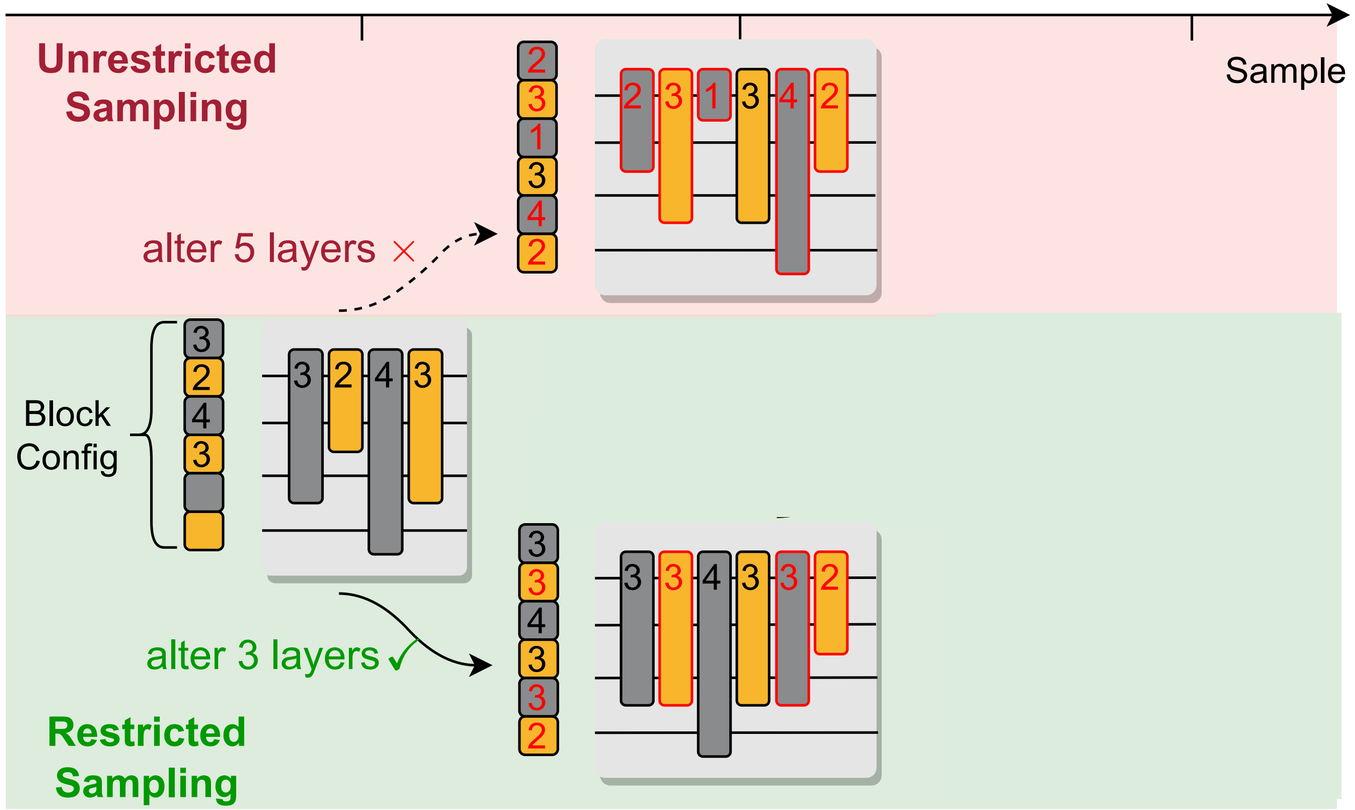
Restricted Sampling

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



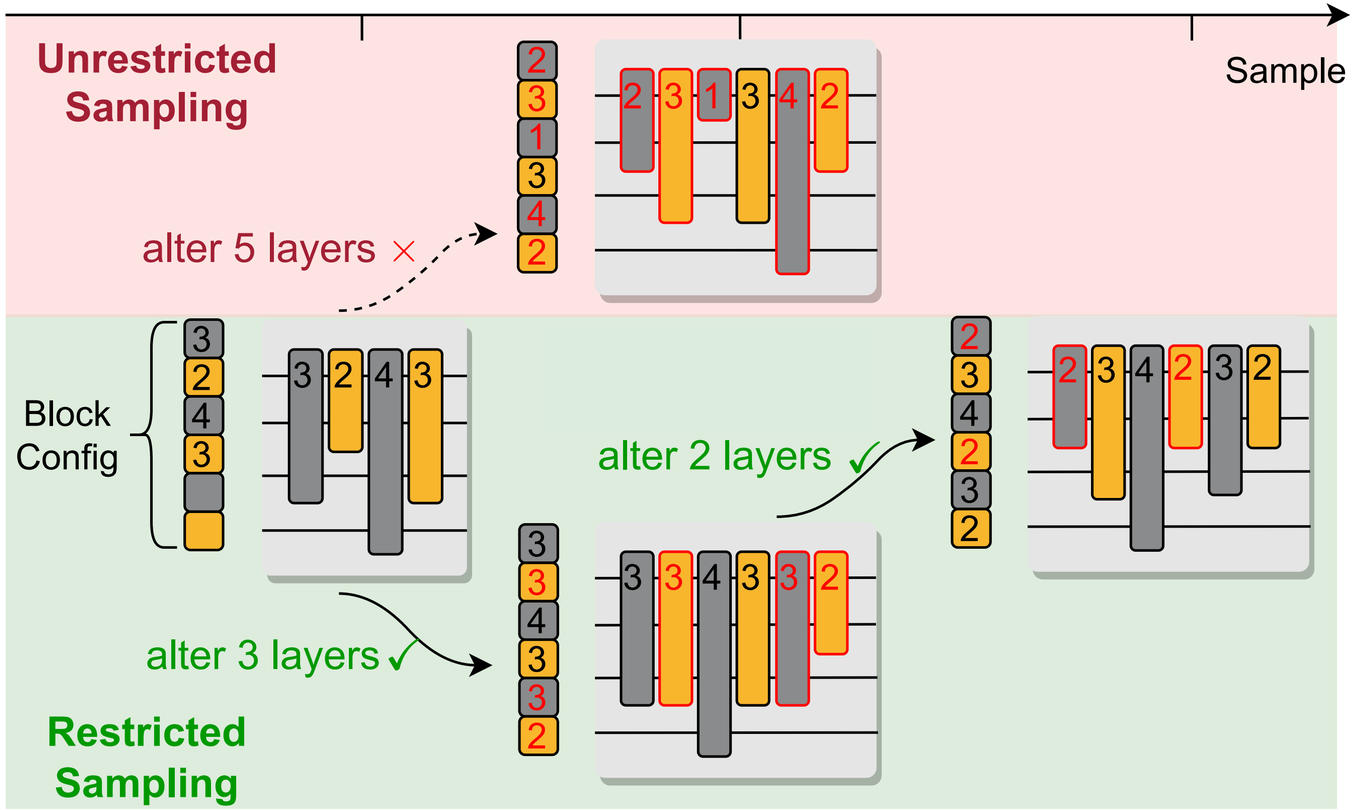
Restricted Sampling

- Restricted Sampling:
  - Restrict the difference between two consecutively sampled SubCircuits
  - For example: restrict to at most 4 different layers



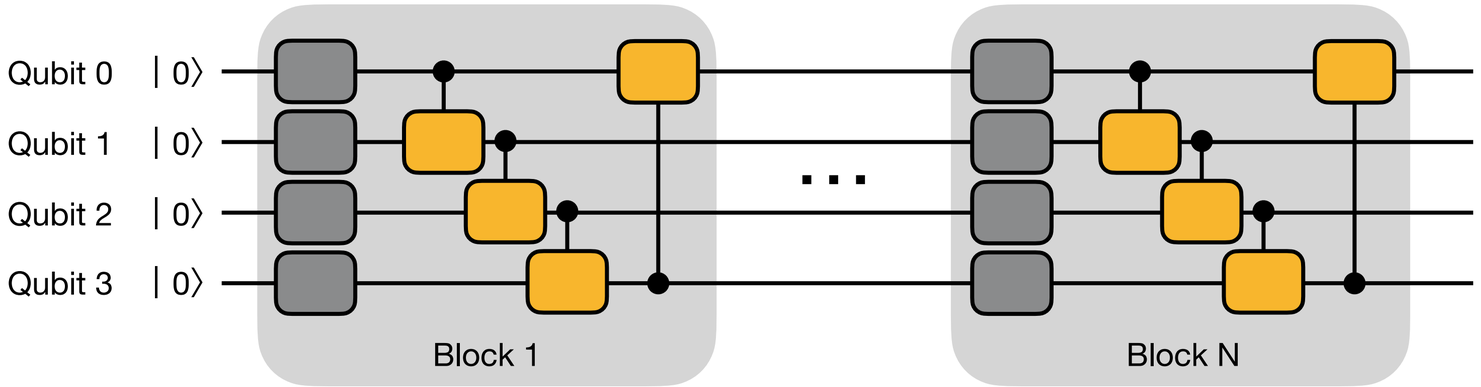
Restricted Sampling

- Restricted Sampling:
  - Restrict the difference between two consecutively sampled SubCircuits
  - For example: restrict to at most 4 different layers



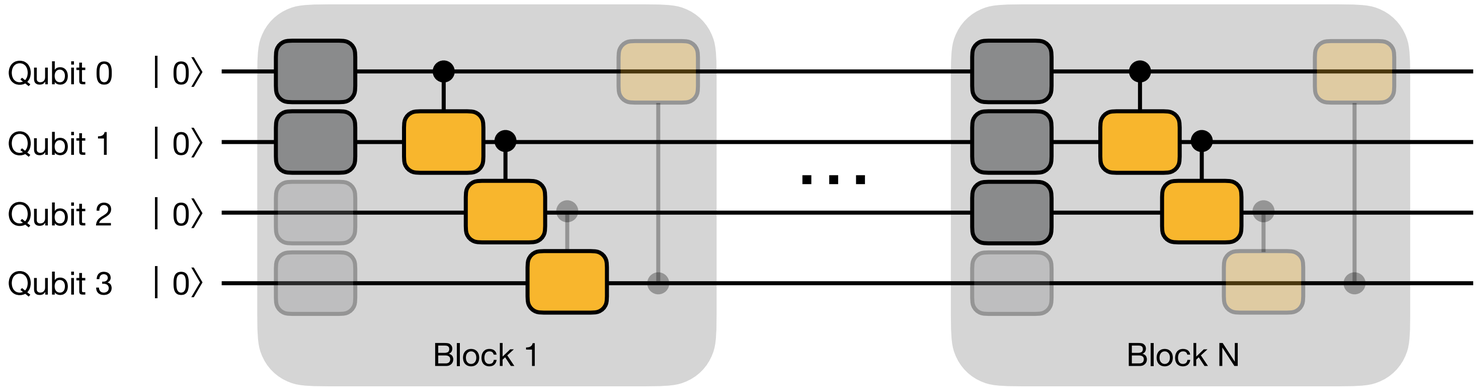
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



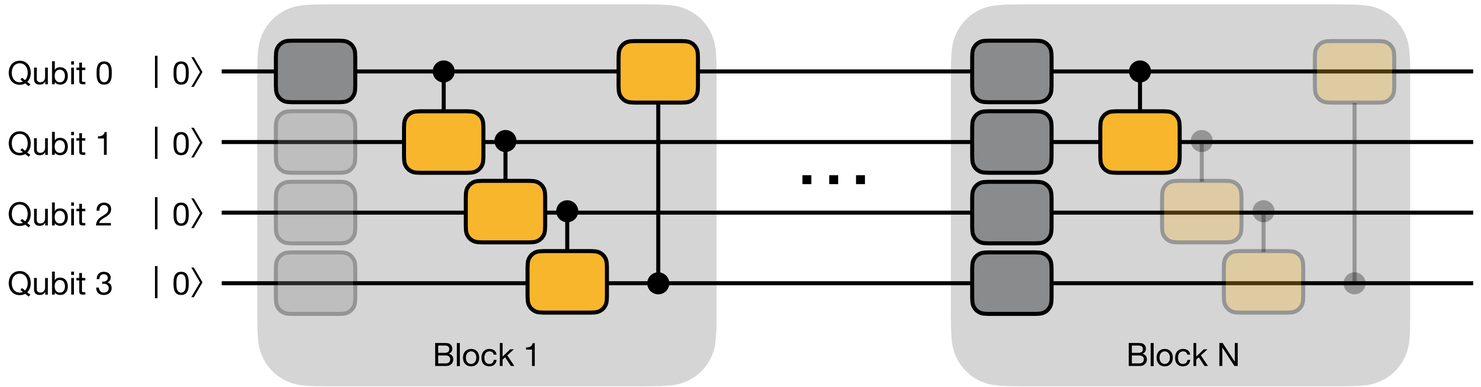
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



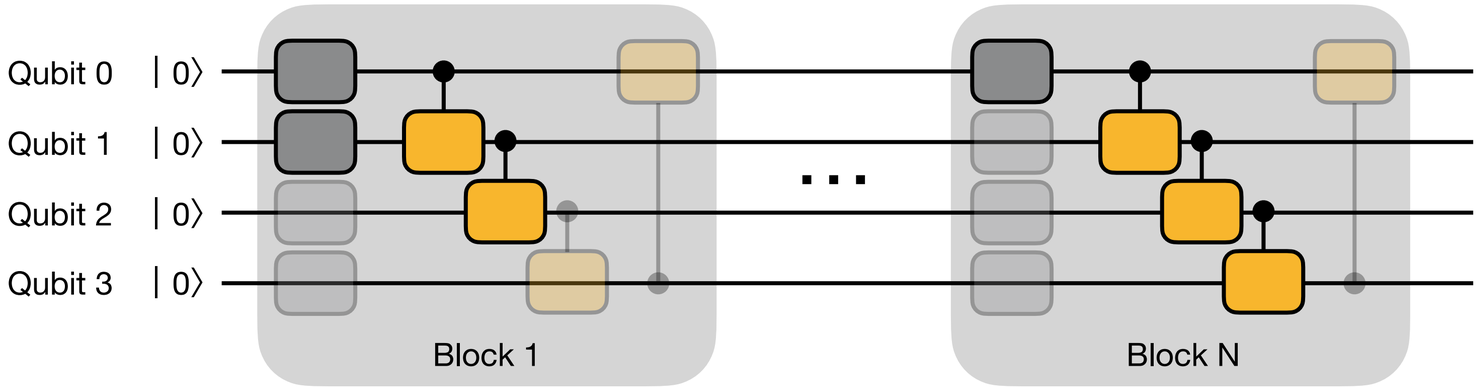
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



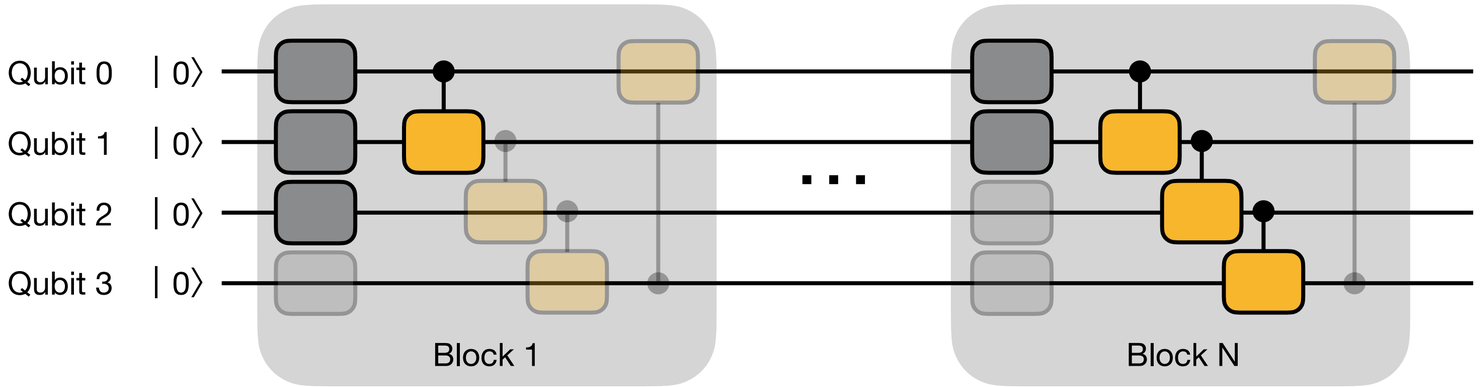
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



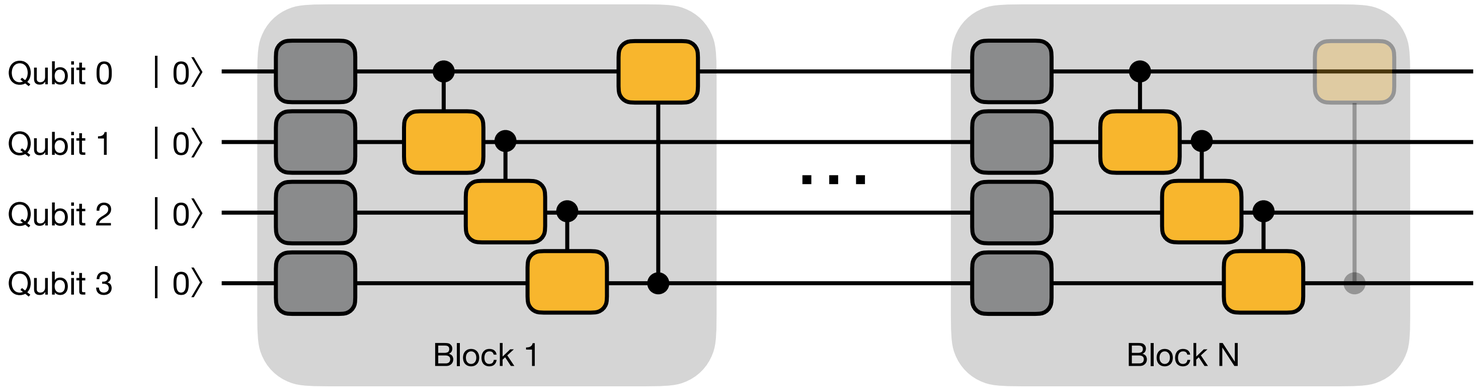
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



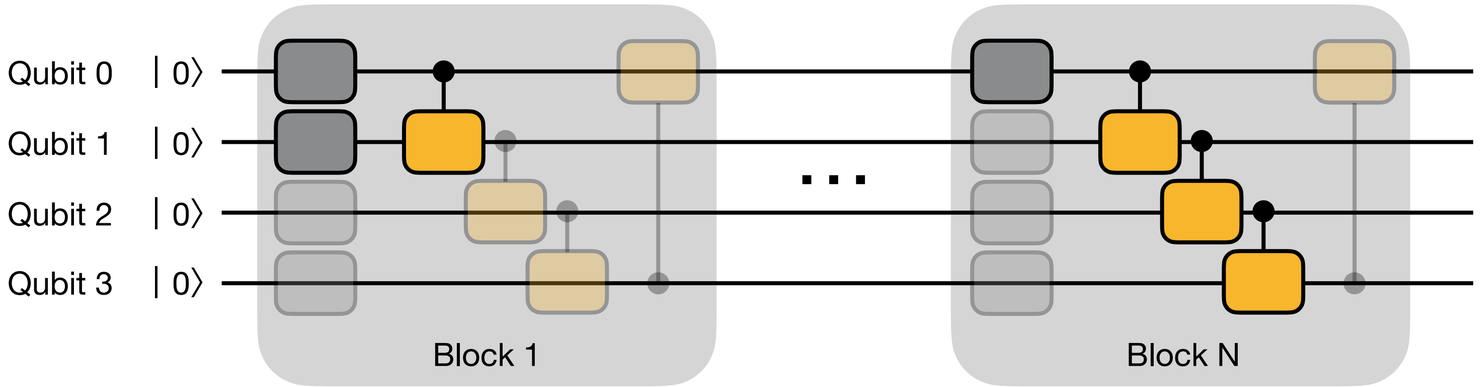
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



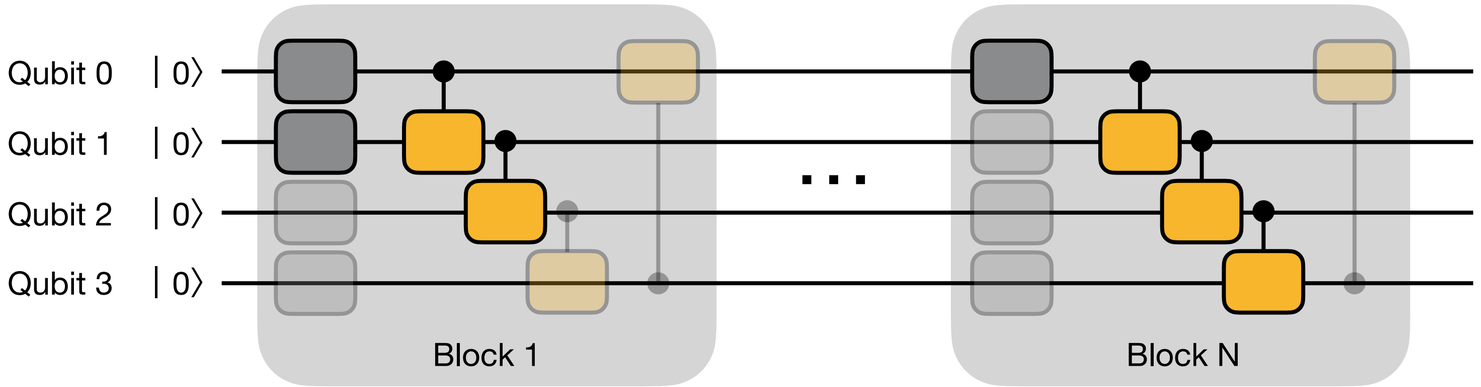
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



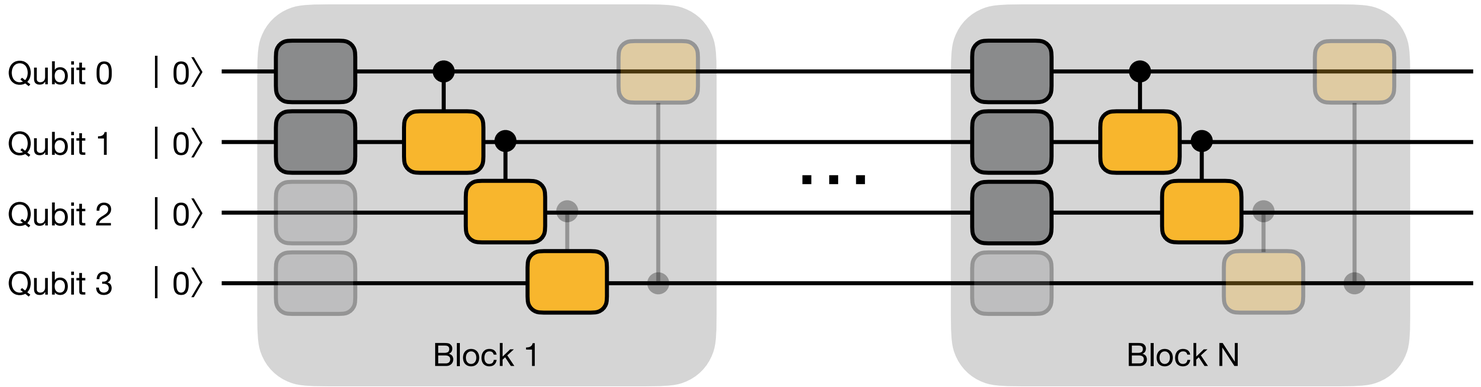
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



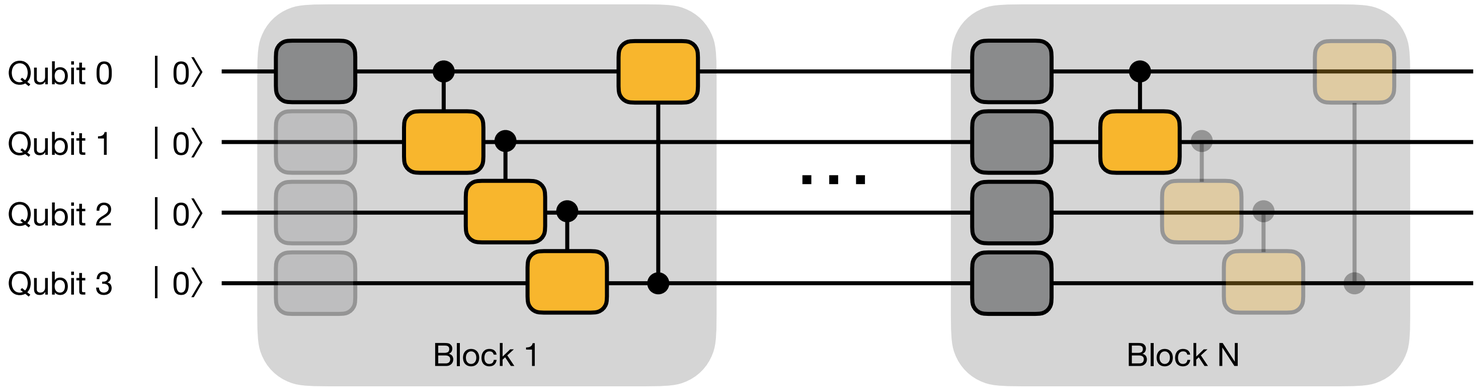
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



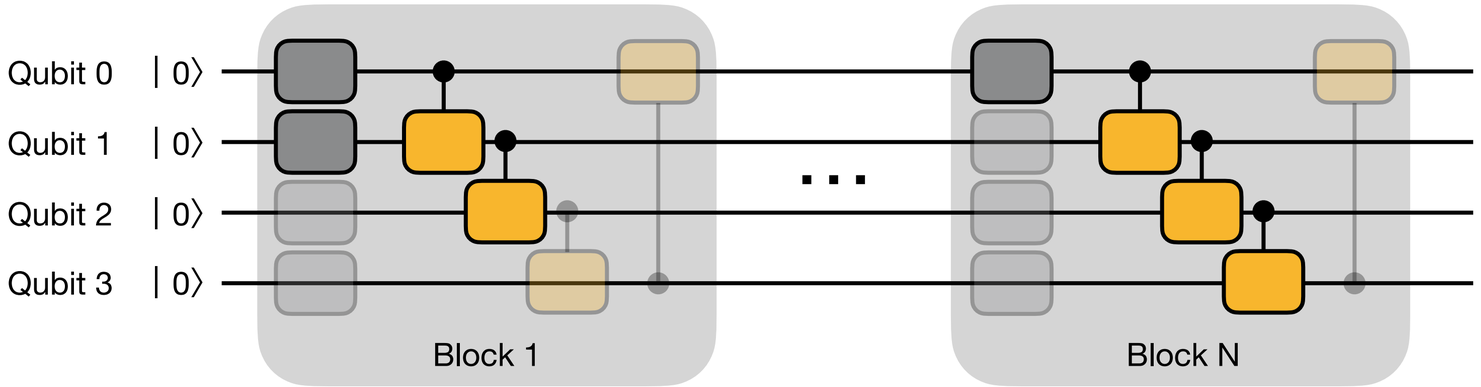
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



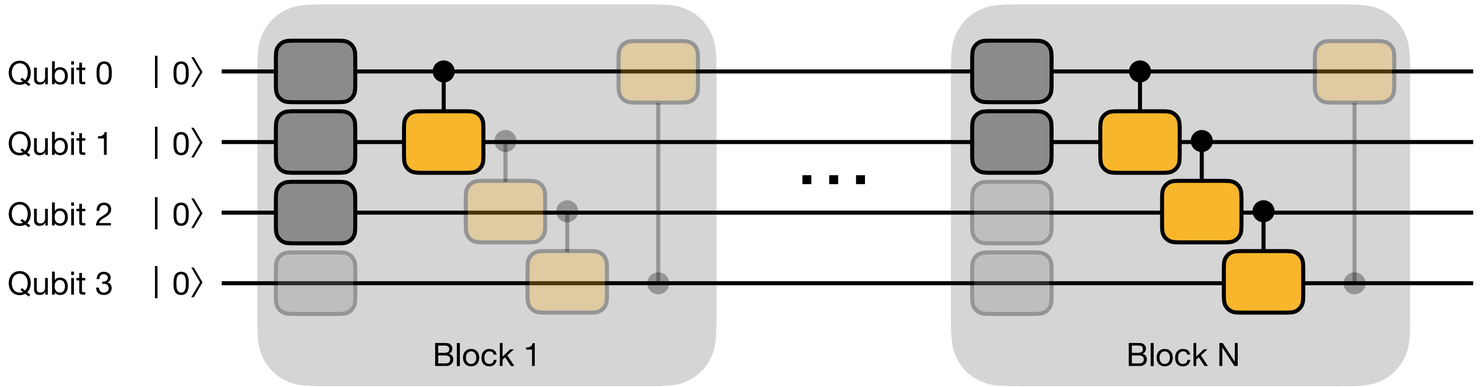
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



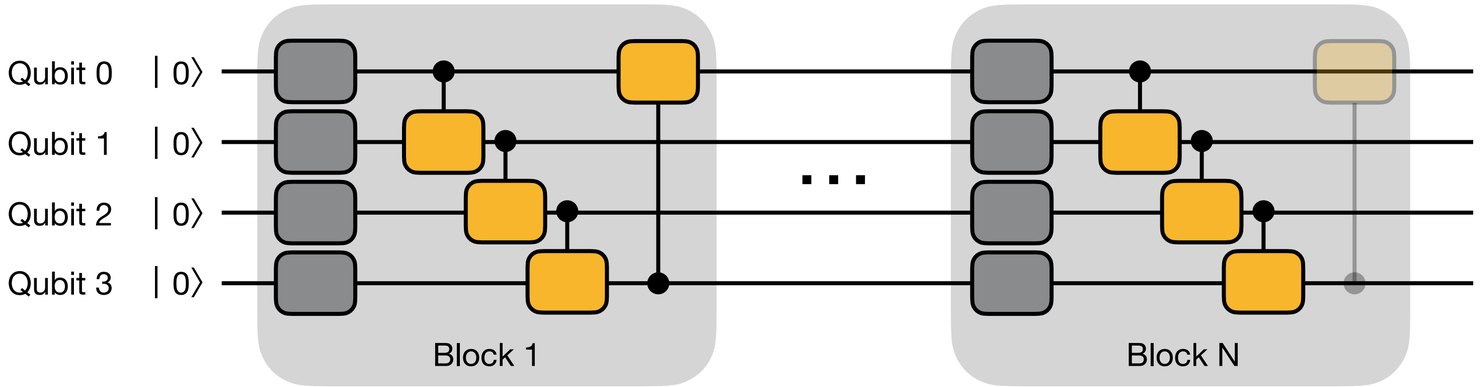
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



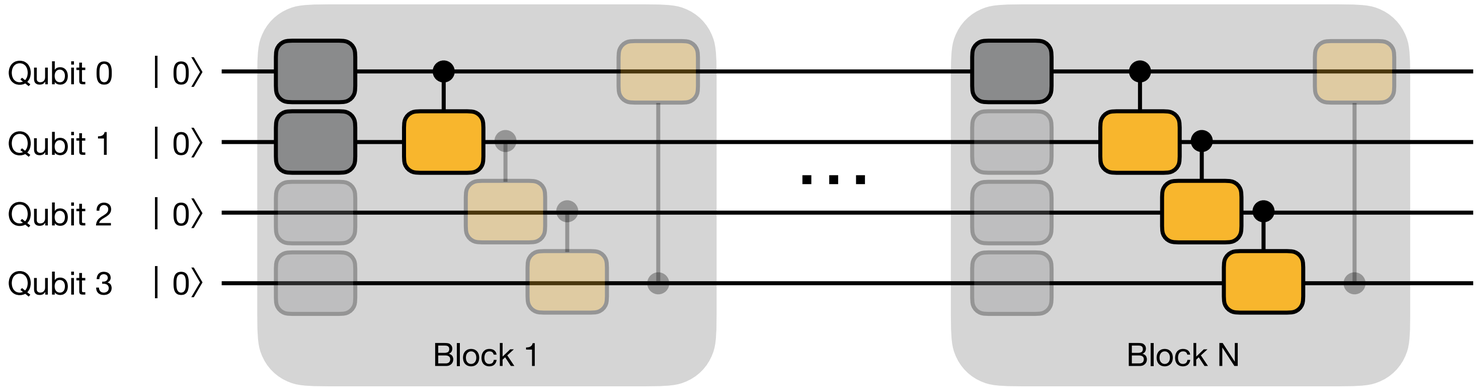
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



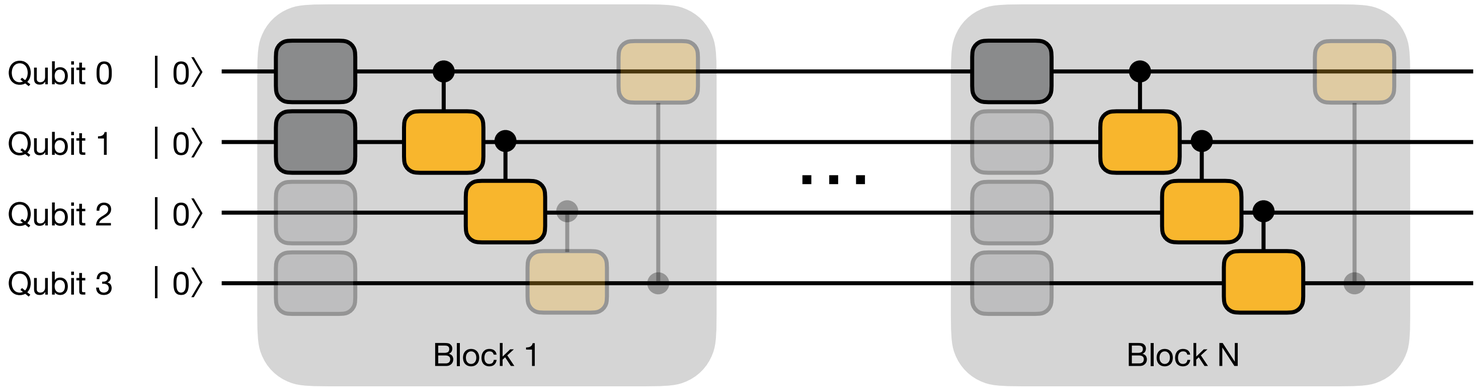
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



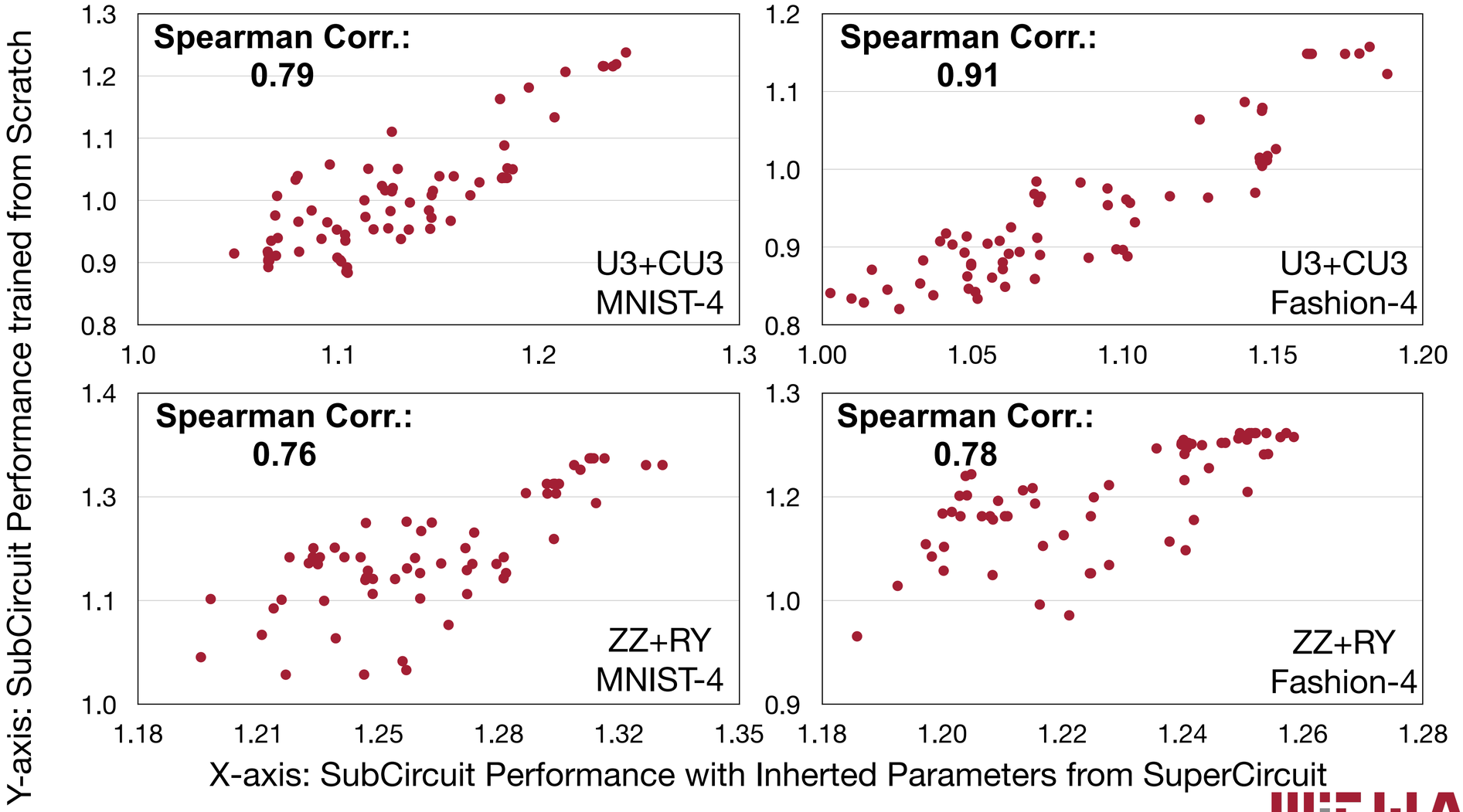
Train SuperCircuit for Multiple Steps

- In one SuperCircuit Training step: Sample and Train



How Reliable is the SuperCircuit?

- Inherited parameters from SuperCircuit can provide accurate relative performance

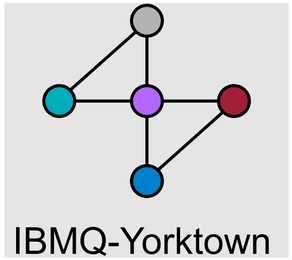


QuantumNAS

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

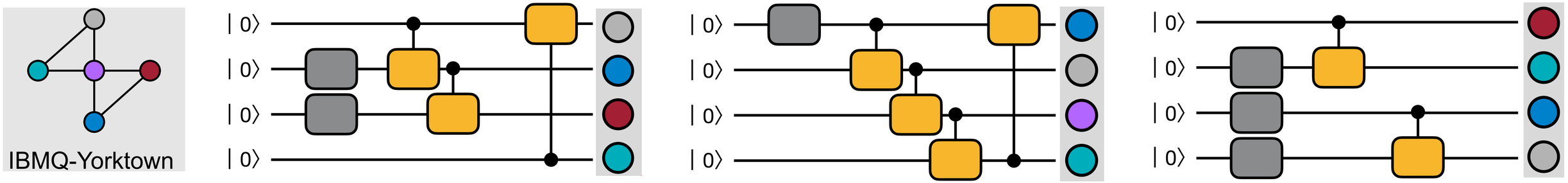
Noise-Adaptive Evolutionary Co-Search

- 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



Evolutionary Search Engine
Mutation
Crossover

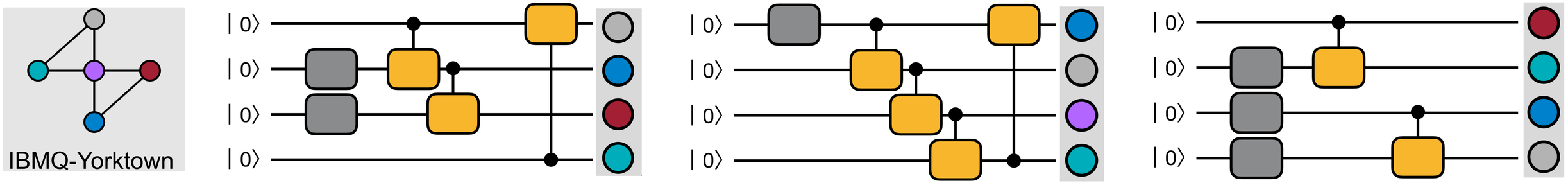
SubCircuit and Qubit Mapping pairs

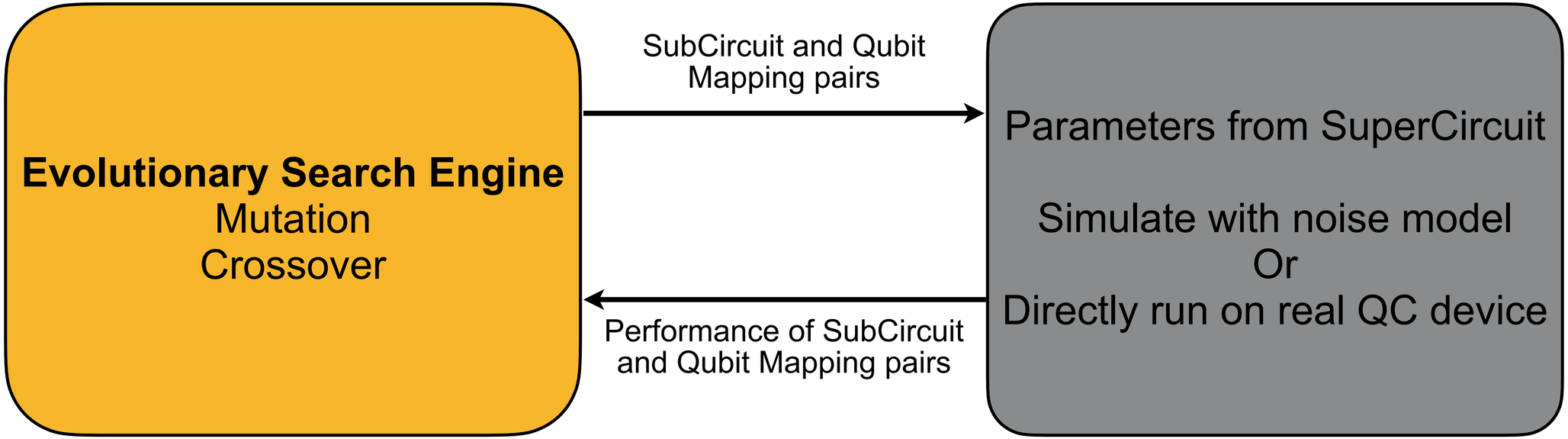
Parameters from SuperCircuit

Simulate with noise model
Or
Directly run on real QC device

Noise-Adaptive Evolutionary Co-Search

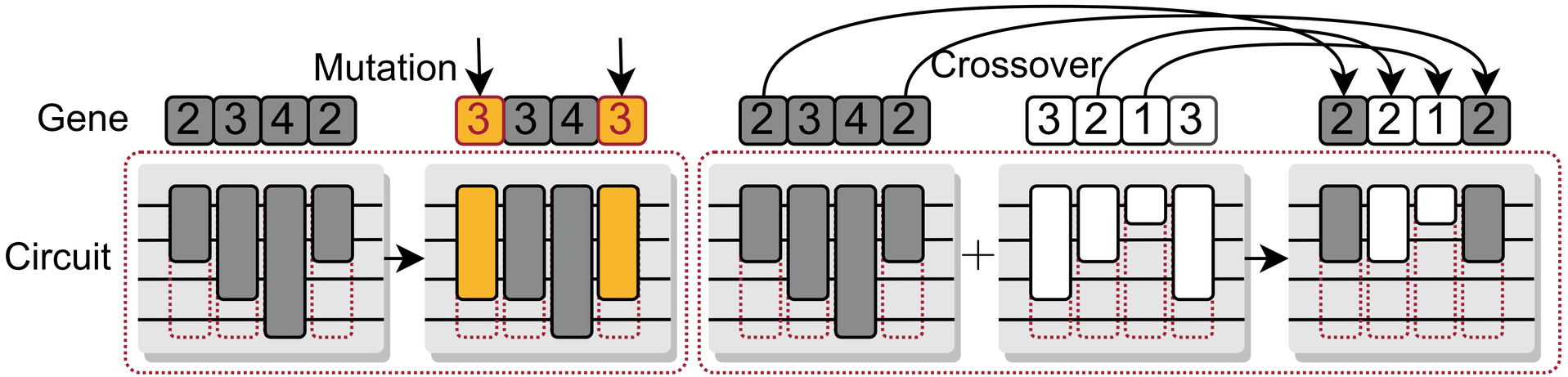
- Search the best SubCircuit and its qubit mapping on target device





Mutation and Crossover

- Mutation and crossover create new SubCircuit candidates



QuantumNAS

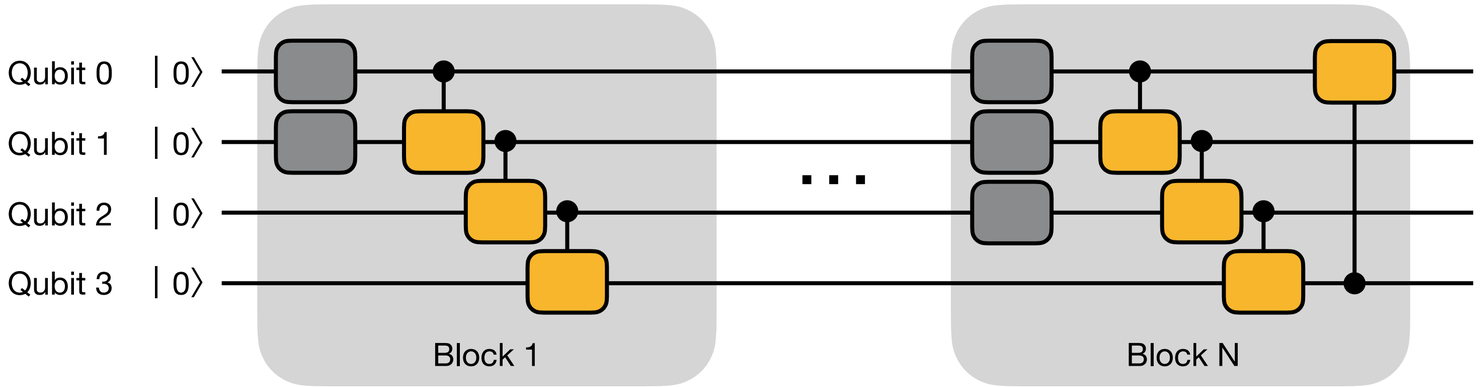
- SuperCircuit Construction and Training
- Noise-Adaptive Evolutionary Co-Search of SubCircuit and Qubit Mapping
- Train the Searched SubCircuit
- Iterative Quantum Gate Pruning

QuantumNAS

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

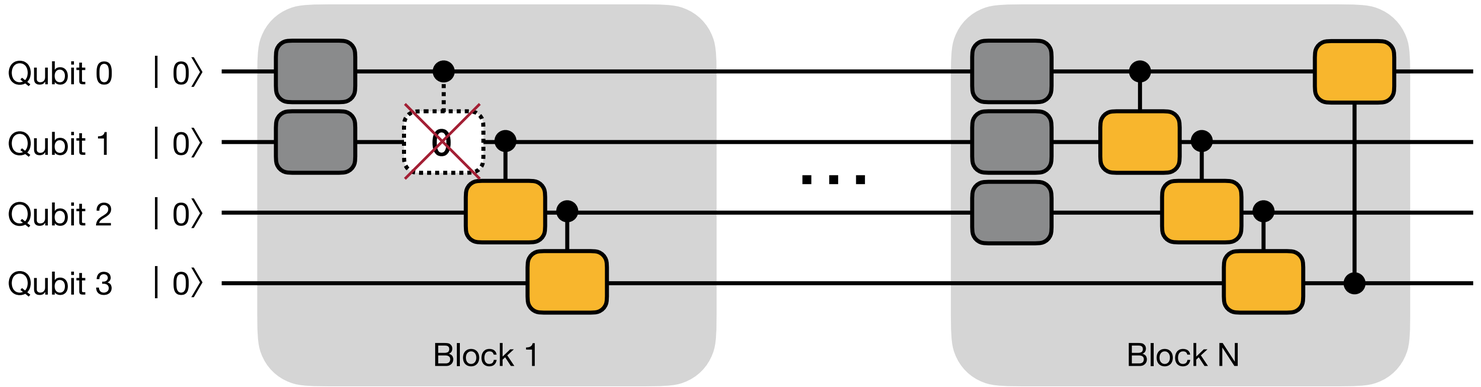
Iterative Pruning

- 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



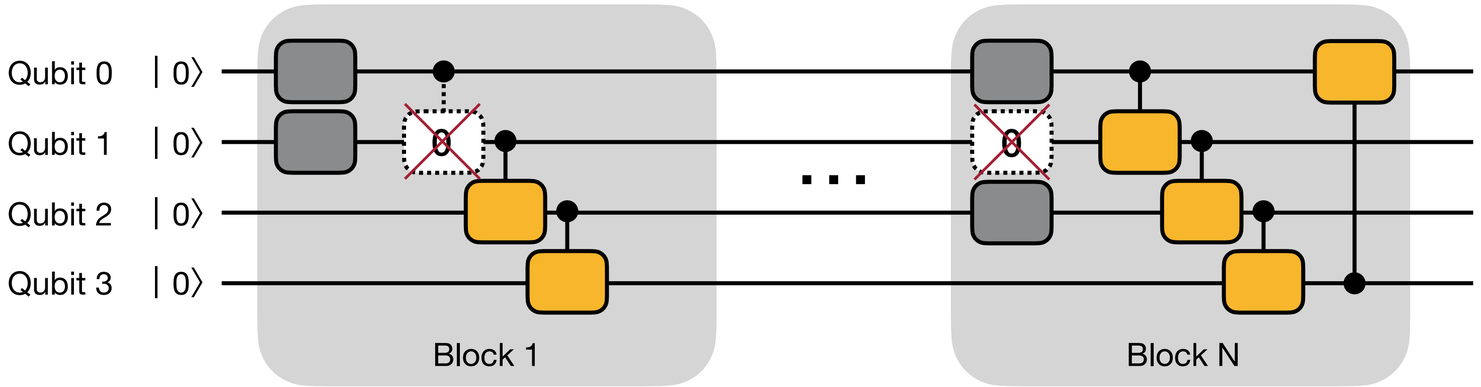
Iterative Pruning

- 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



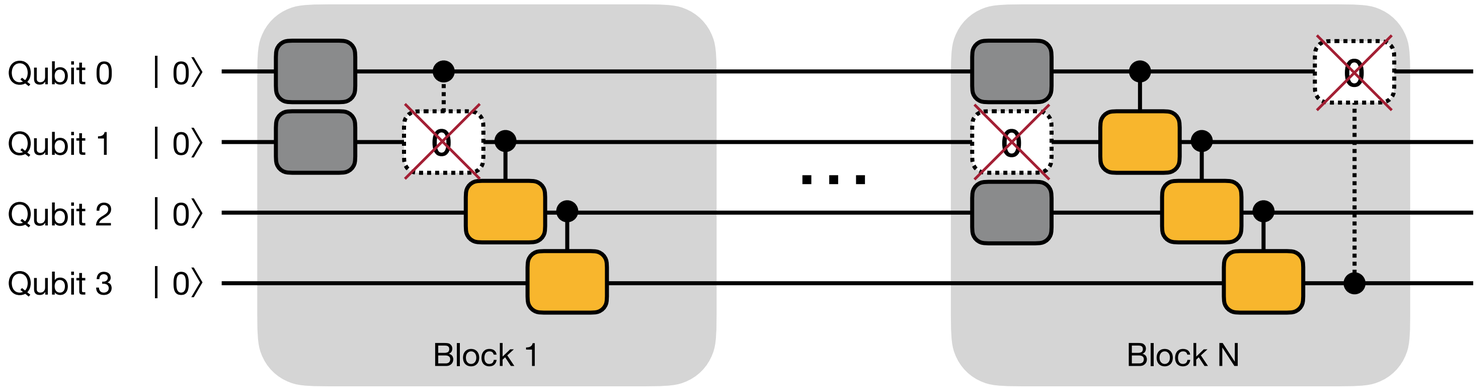
Iterative Pruning

- 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



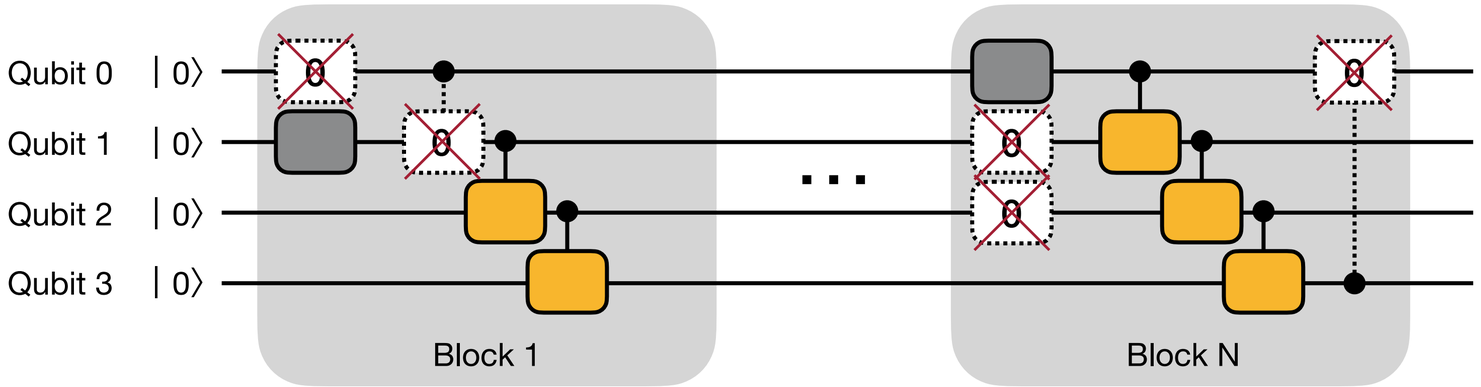
Iterative Pruning

- 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



Outline

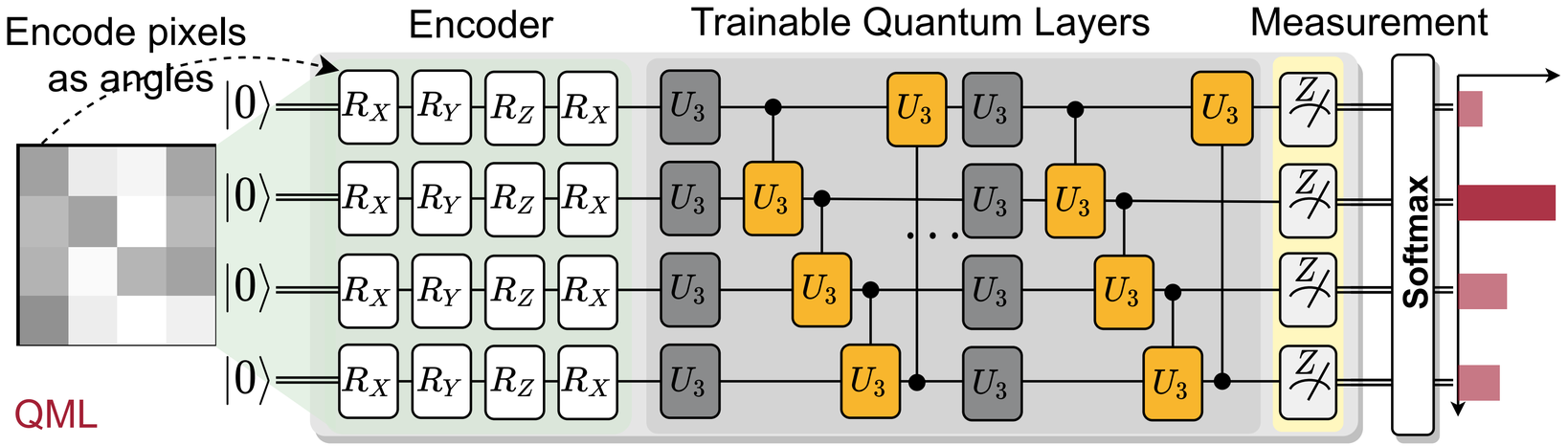
- Overview
- Background
- QuantumNAS
- Evaluation
- TorchQuantum Library
- Conclusion

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: H2, H2O, LiH, CH4, BeH2
- Quantum Devices
  - IBMQ
  - #Qubits: 5 to 65
  - Quantum Volume: 8 to 128

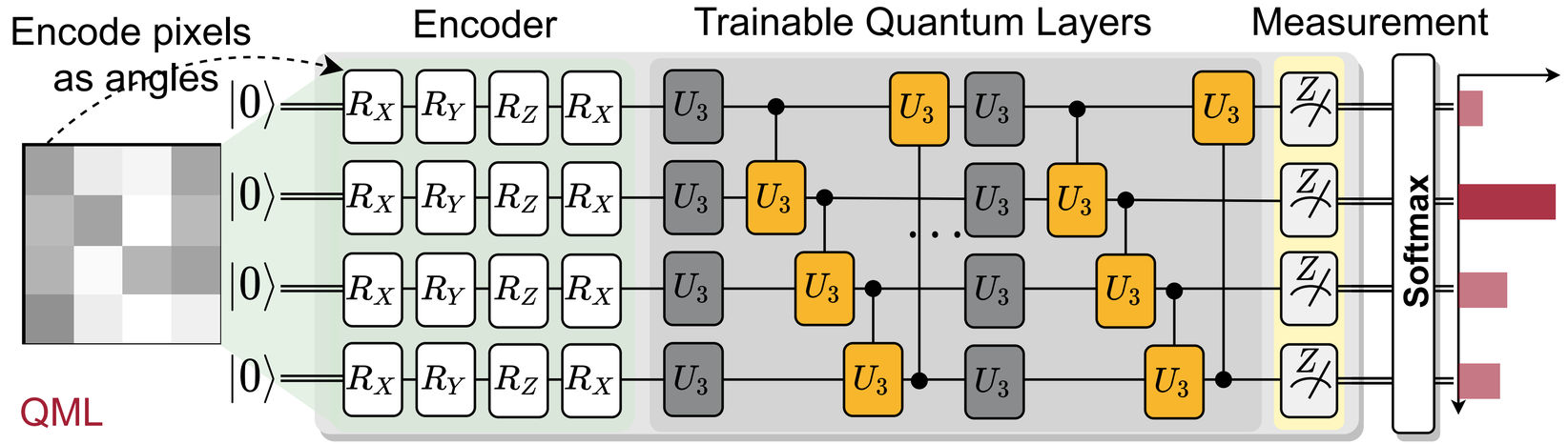
Benchmarks: QNN and VQE

- Quantum Neural Networks: classification

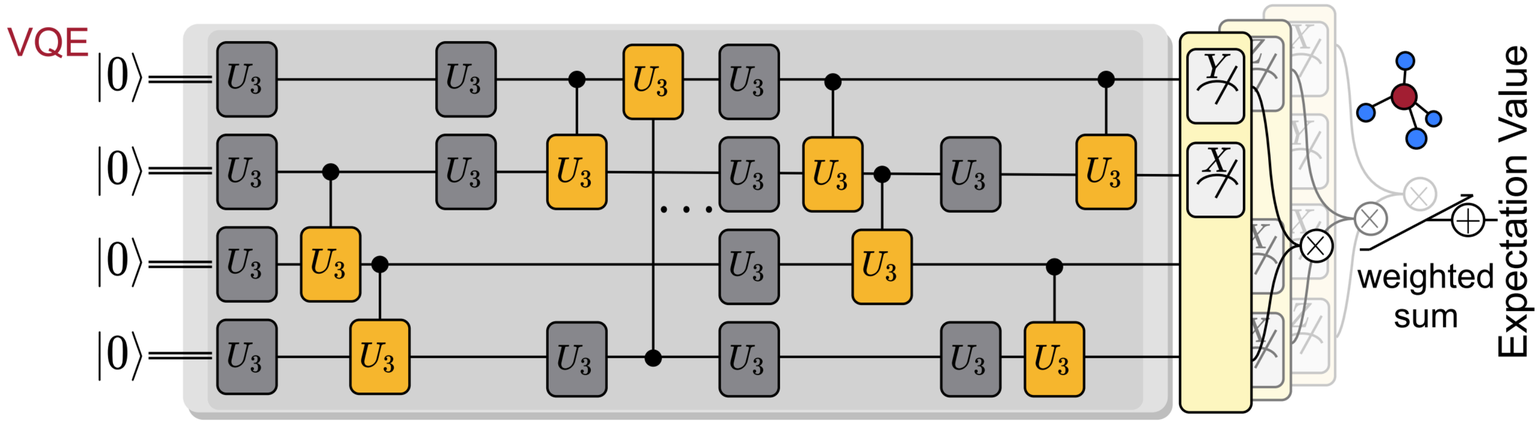


Benchmarks: QNN and VQE

- Quantum Neural Networks: classification

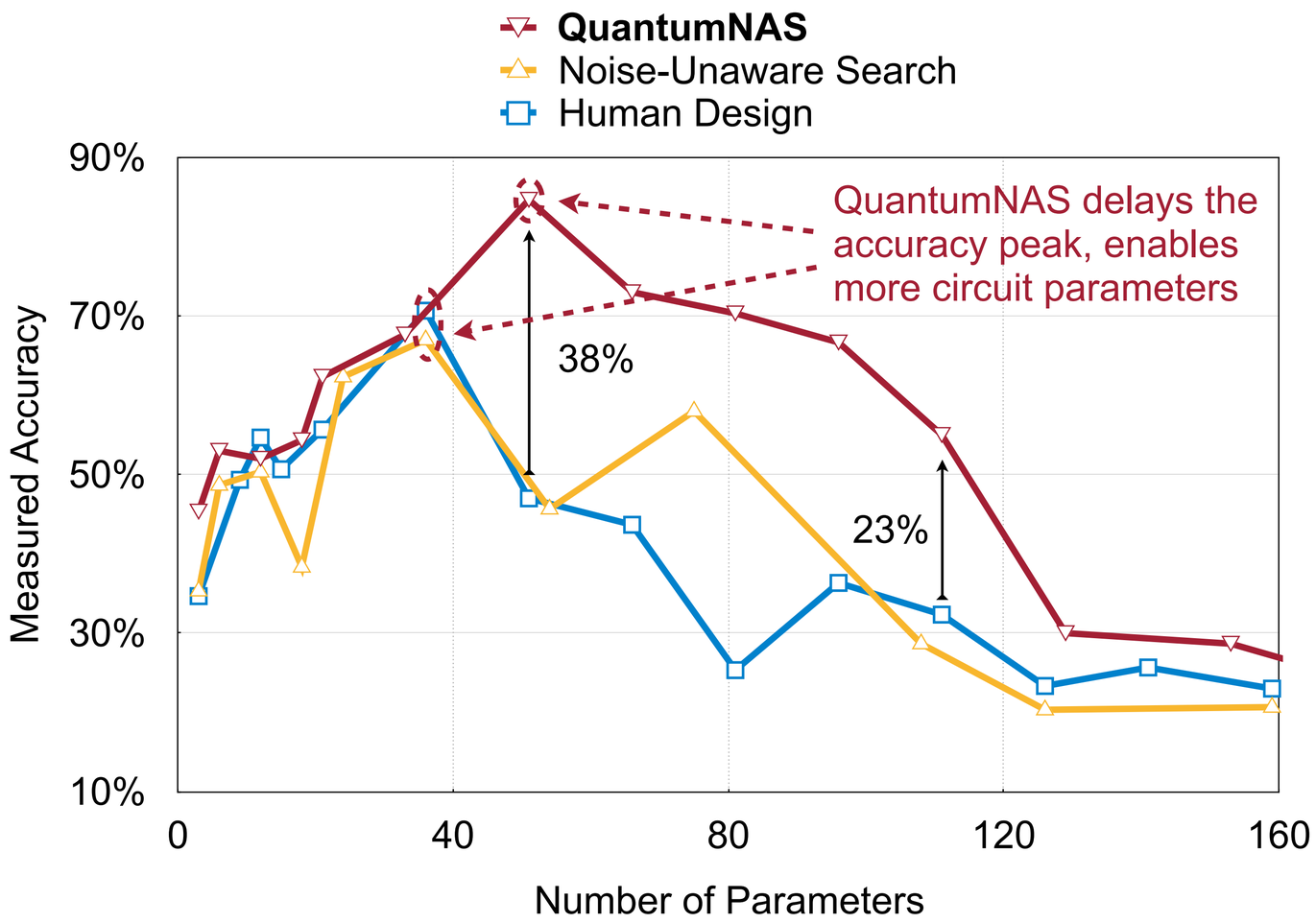


- Variational Quantum Eigensolver: finds the ground state energy of molecule Hamiltonian



QML Results

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



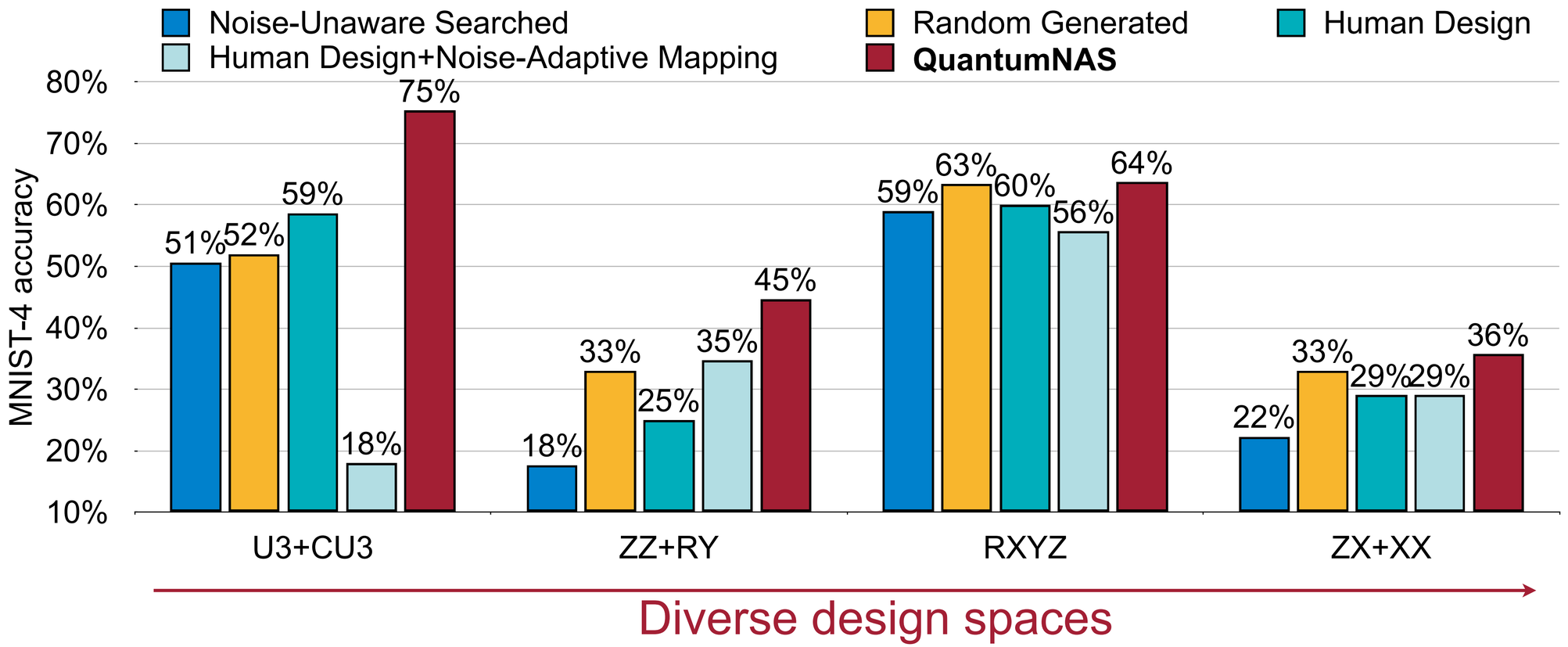
Analysis of Searched Circuit

- QuantumNAS searched circuit has fewer #gates and #params but higher accuracy

MNIST-2	Depth	#Gates	#Params	**QuantumNAS**
Human Design	64	135	36	**88%**
QuantumNAS	70	116	22	**92%**

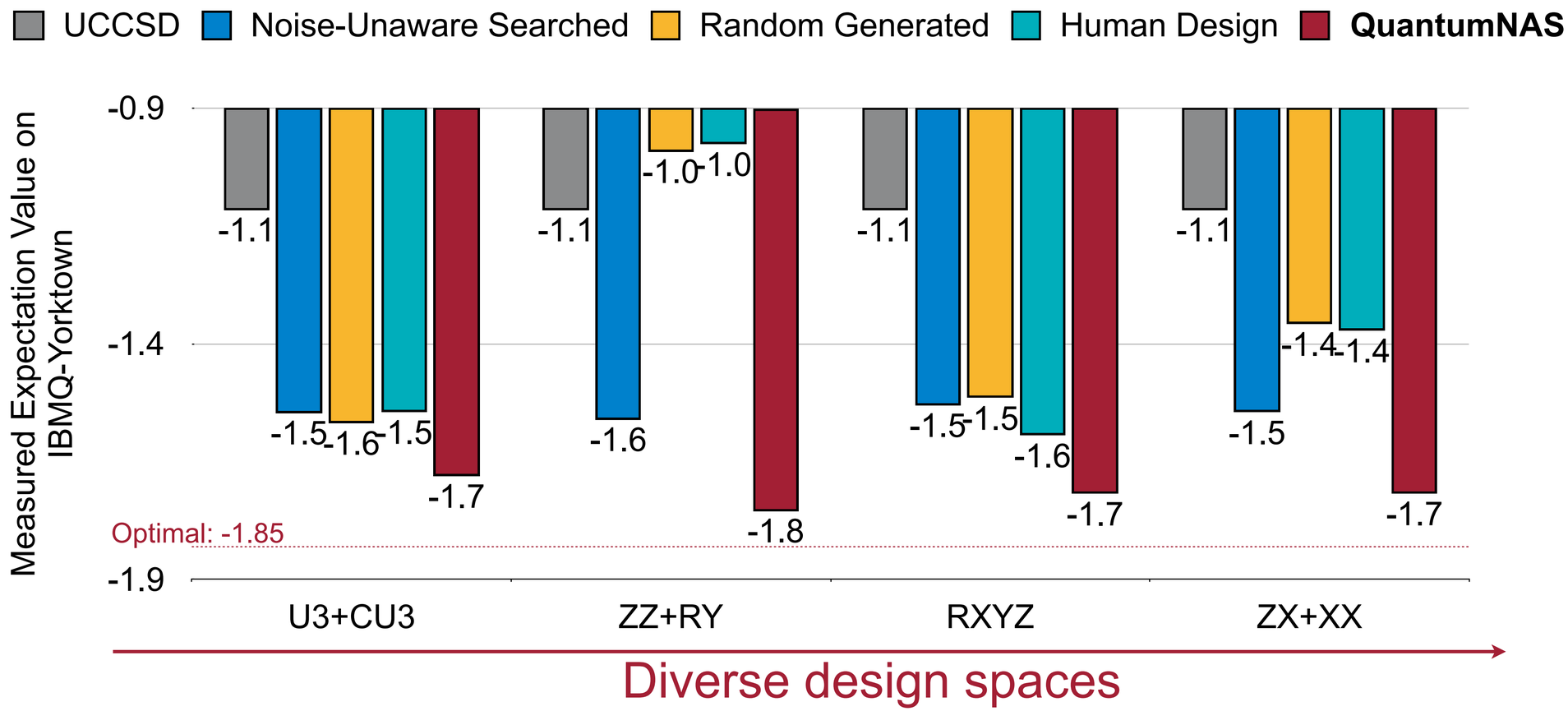
Consistent Improvements on Diverse Design Spaces

- ‘U3+CU3’, ’ZZ+RY’, ‘RXYZ’, ‘ZX+XX’ spaces with different gates



Consistent Improvements on Diverse Design Spaces

- H2 in different design spaces on IBMQ-Yorktown



Consistent Improvements on Diverse Devices

- QuantumNAS is effective for different real quantum devices
- On different 5-Qubit devices
- MNIST-4, Fashion-4, Vowel-4, MNIST-2, Fashion-2 averaged accuracy

Diverse devices

Method	Noise-Unaware Searched	Random	Human	**QuantumNAS**
Belem (5Q, 16QV)	47%	50%	67%	**76%**
Quito (5Q, 16QV)	73%	68%	74%	**79%**
Athens (5Q, 32QV)	50%	63%	68%	**77%**
Santiago (5Q, 32QV)	74%	73%	75%	**80%**

Scalable to Large #Qubits

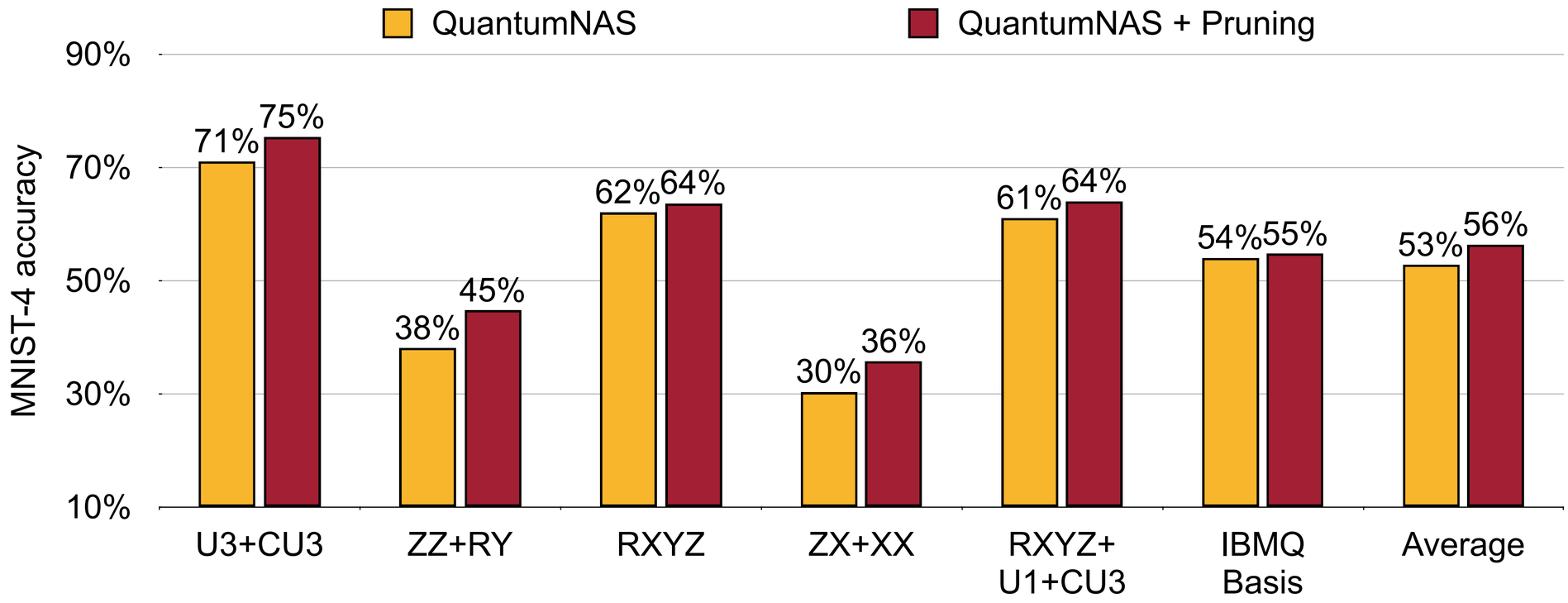
- On large devices
- MNIST-10 accuracy

More Qubits

Method	Noise-Unaware Searched	Random	Human	**QuantumNAS**
Melbourne (15Q, 8QV, use 15Q)	11%	10%	15%	**32%**
Guadalupe (16Q, 32QV, use 16Q)	14%	12%	10%	**15%**
Montreal (27Q, 128QV, use 21Q)	13%	7%	14%	**16%**
Manhattan (65Q, 32QV, use 21Q)	11%	11%	15%	**18%**

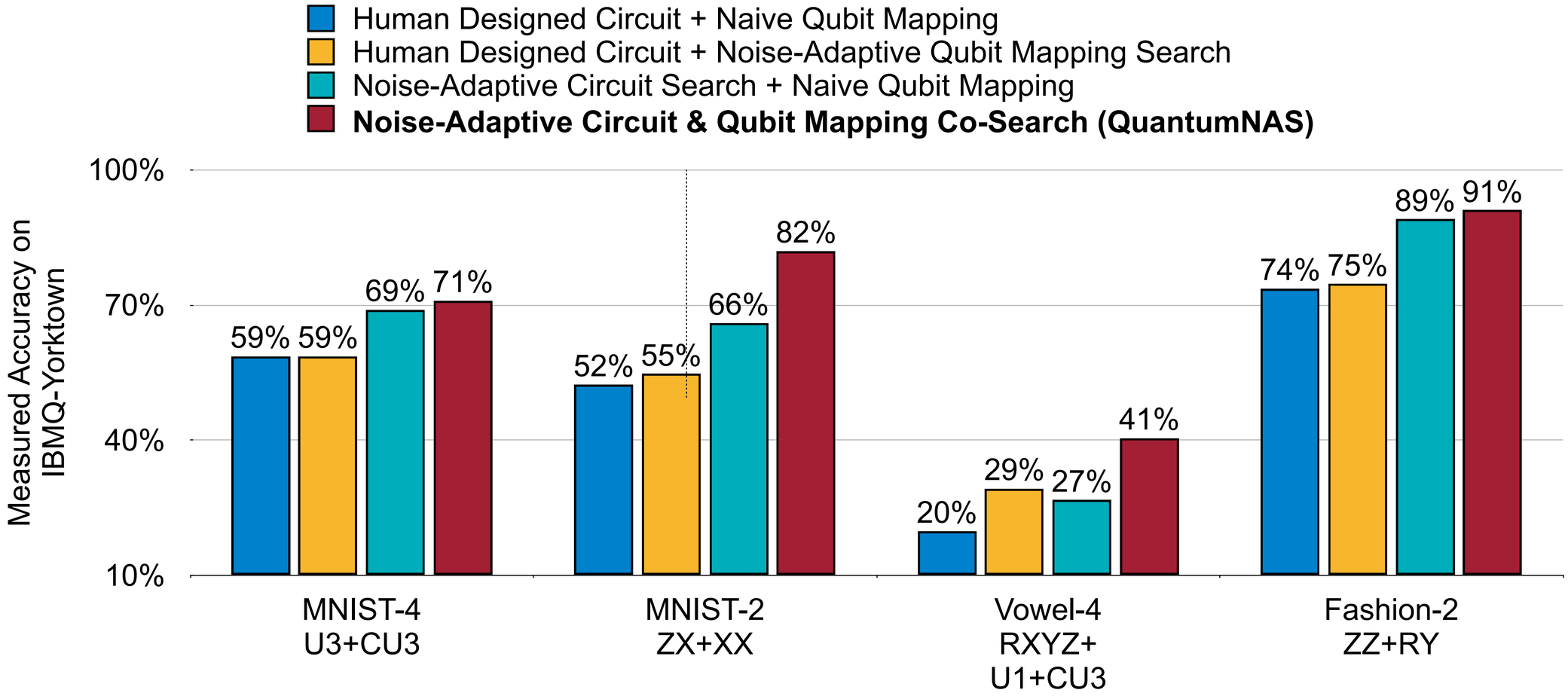
Effective of Quantum Gate Pruning

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



Effect of Circuit & Qubit Mapping Co-Search

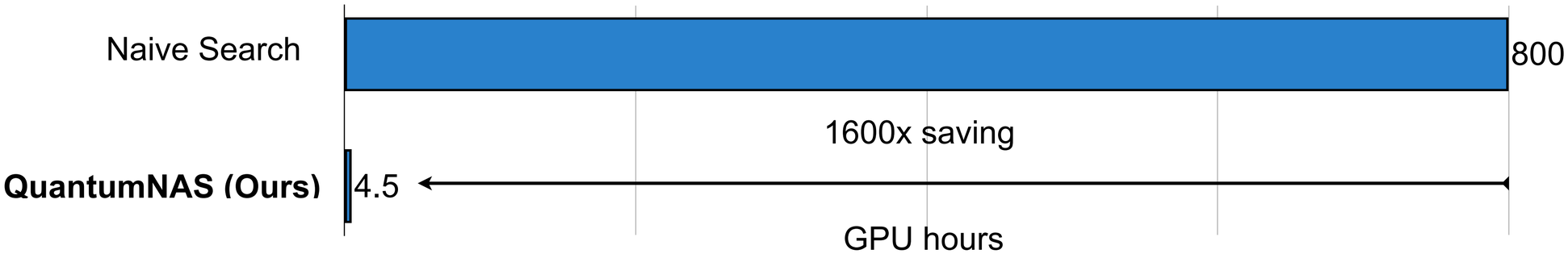
- Co-search is better than only search architecture/mapping



Time Cost

- 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

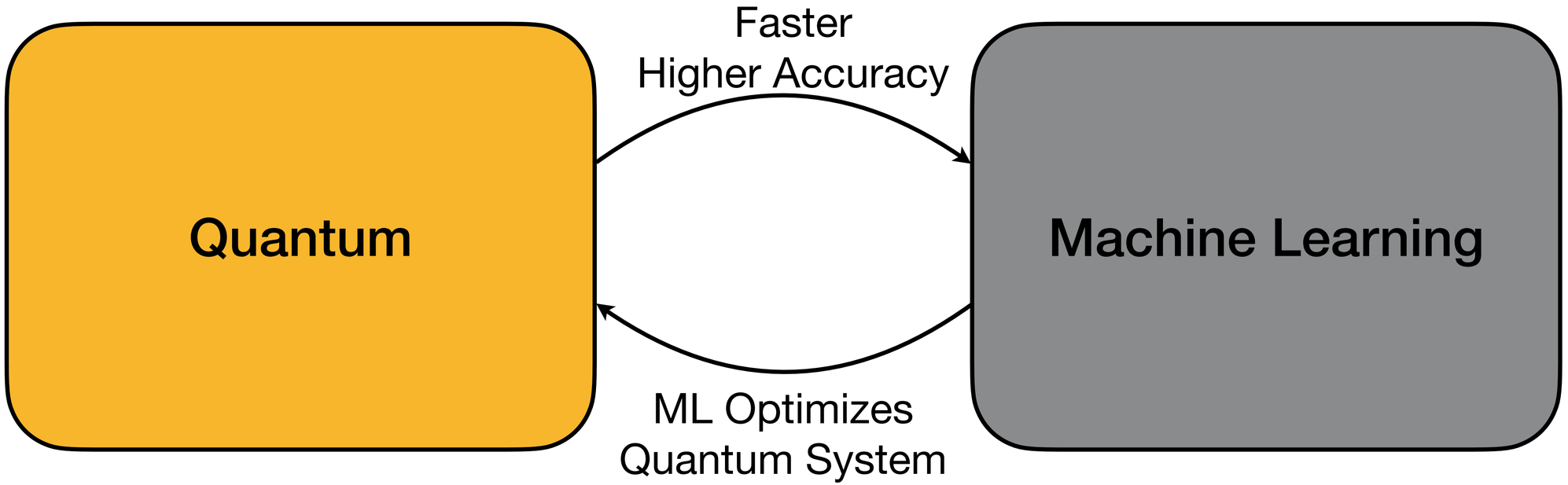


Outline

- Quick Overview
- Background
- QuantumNAS
- Evaluation
- TorchQuantum Open-source
- Conclusion

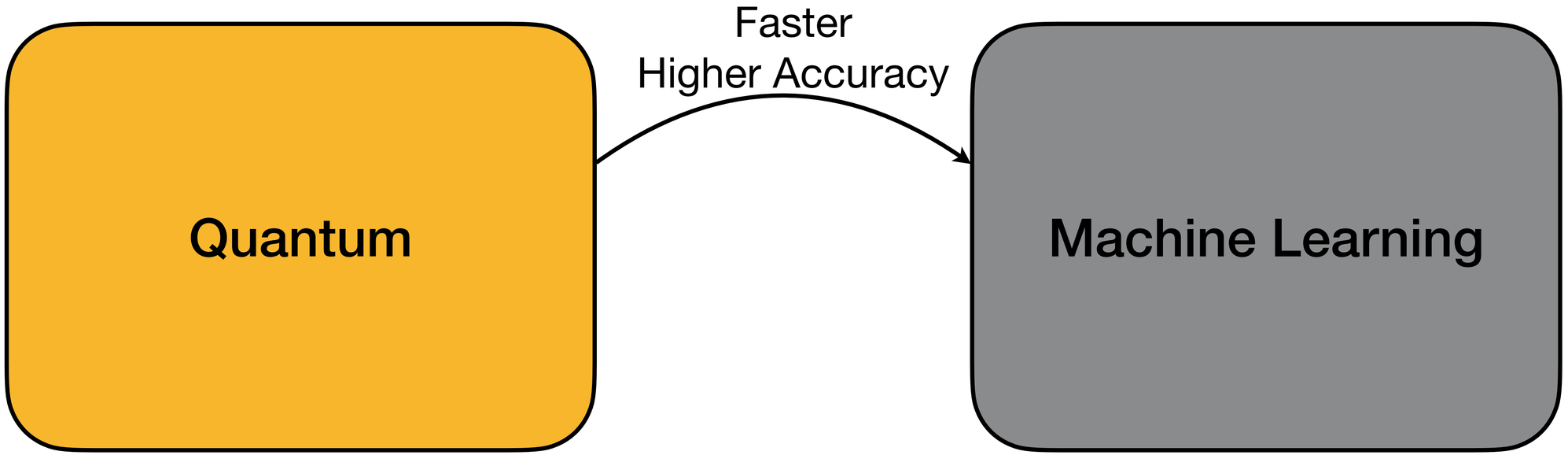
Open-source: TorchQuantum

- TorchQuantum — An open-source library for interdisciplinary research of quantum computing and machine learning
- https://github.com/mit-han-lab/torchquantum



Open-source: TorchQuantum

- TorchQuantum — An open-source library for interdisciplinary research of quantum computing and machine learning
- https://github.com/mit-han-lab/torchquantum

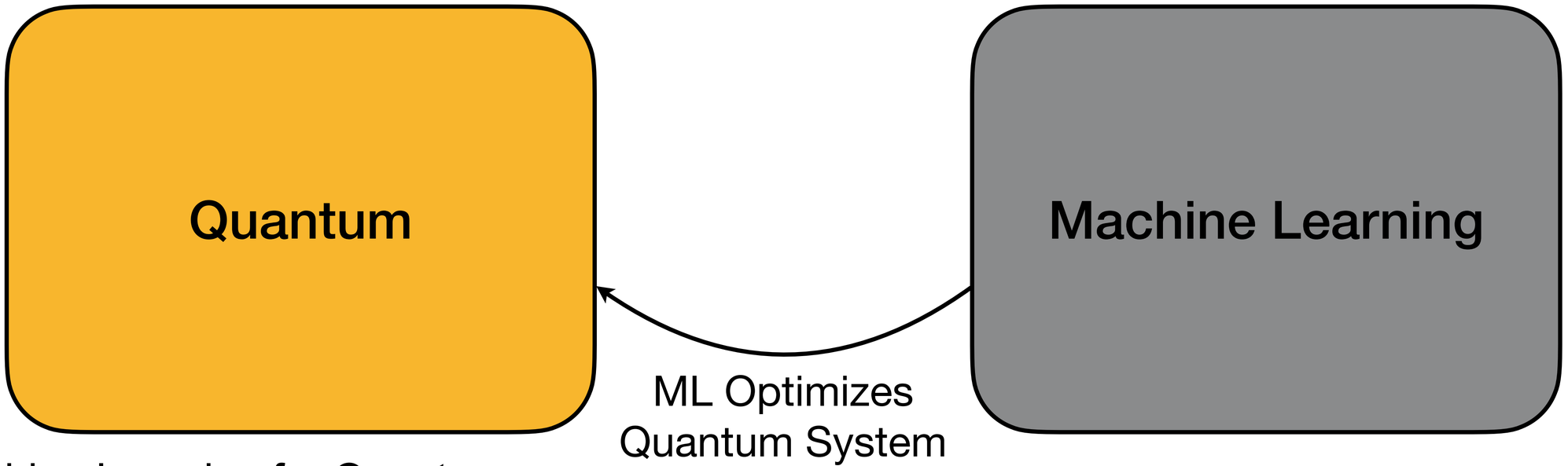


- Quantum for Machine learning
  - Quantum neural networks
  - Quantum kernel methods

Torch Quantum

Open-source: TorchQuantum

- TorchQuantum — An open-source library for interdisciplinary research of quantum computing and machine learning
- https://github.com/mit-han-lab/torchquantum



- Machine Learning for Quantum
  - ML for quantum compilation (qubit mapping, unitary synthesis)
  - …

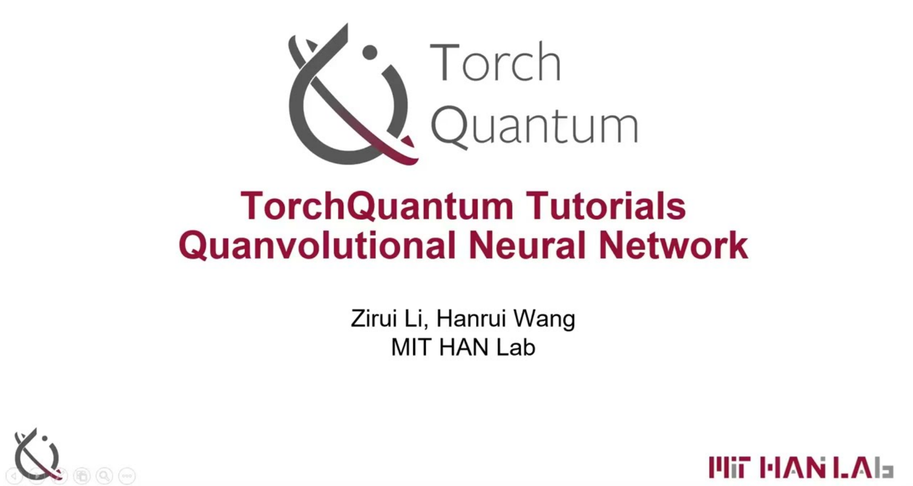
Torch Quantum

TorchQuantum

- Features
  - Easy construction of **parameterized quantum circuits** such as Quantum Neural Networks in PyTorch
  - Support **batch mode inference and training** on GPU/CPU, supports highly-parallelized training
  - Support **easy deployment** on real quantum devices such as IBMQ
  - Provide tutorials, videos and example projects of QML and using ML to optimize quantum computer system problems

Examples and tutorials

- Tutorial Colab and videos



Outline

- Overview
- Background
- QuantumNAS
- Evaluation
- TorchQuantum Library
- Conclusion

Conclusion

- **QuantumNAS** exploits **SuperCircuit-**based co-search for most **noise-robust** circuit architecture and qubit mapping
- Iterative **quantum gate pruning** to further remove redundant gates
- Improves MNIST 2-class accuracy from 88% to **95%**,10-class from 15% to **32%**
- Save search cost by over **1,000 times**
- Open-sourced **TorchQuantum** library for Quantum + ML research

Torch Quantum

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

qmlsys.mit.edu