Q-Trainer: *A High-Level API for*

Training Variational Quantum Circuits with Error Mitigation

Min Li

Physics PhD Candidate

Quantum Info, High-Energy

Haoxiang Wang

ECE PhD Candidate

Machine Learning

University of Illinois Urbana-Champaign

Motivations

Variational Quantum Algorithms (VQA)	Quantum Error Mitigation (QEM)
• Software Platforms: Pennylane, Qiskit, TensorFlow Quantum, etc.	• Software Platform: mitiq

mitiq user's feedback: Hope to use mitiq to VQA easily in existing VQA software frameworks.

Example: A user's message in the Discord channel of mitiq 👇

> Hey this is more a question about how to use mitiq. From my understanding I always need an executor and access to the circuit I am using. Is there any easy way to implement mitiq on qiskits algorithm? The code would like the picture I have attached. I don't see any easy way other than implementing qaoa manually or has this already been done?

Q-Trainer

Example 1:

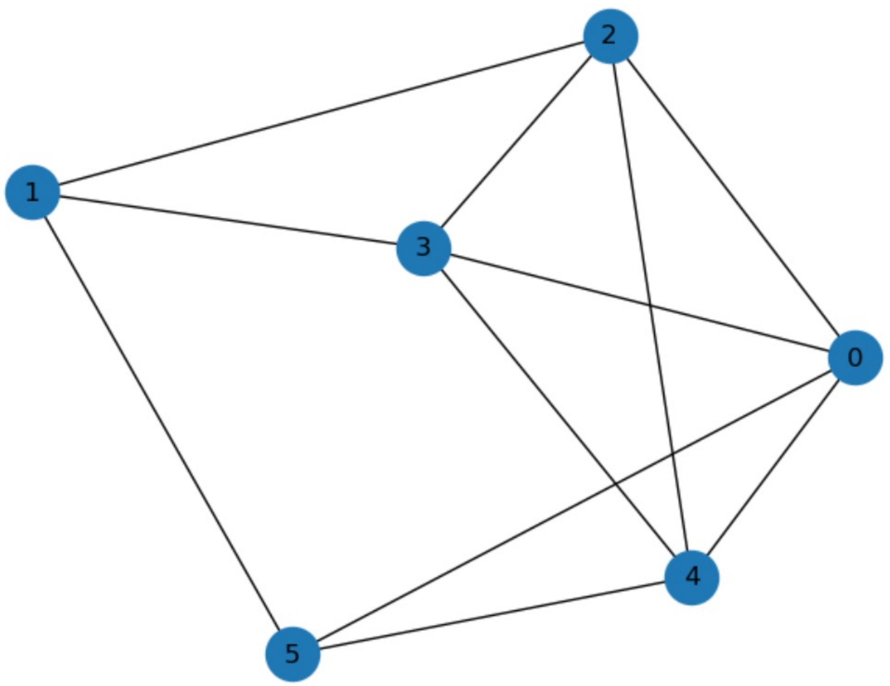
QAOA

Define a graph

```
n_nodes = 6
p = 0.5  # probability of an edge
seed = 1967

g = nx.erdos_renyi_graph(n_nodes, p=p, seed=seed)
positions = nx.spring_layout(g, seed=seed)

nx.draw(g, with_labels=True, pos=positions, node_size=600)
```



Configurations

Q-Trainer's QAOA Circuit class

Available QAOA Tasks:

- `maxcut` : Maximum Cut
- `max_clique` : Maximum Clique
- `max_independent_set` : Maximum Independent Set
- `max_weight_cycle` : Maximum Weighted Cycle
- `min_vertex_cover` : Minimum Vertex Cover

```
task = "maxcut"
depth = 2
circuit = qtrainer.circuits.QAOACircuit(
    graph=g, task=task, depth=depth, seed=0)
```

Noise Model

```
# Depolarization noise on all gates
noise_gate = qml.DepolarizingChannel
noise_strength = 0.1
noise_fn = qml.transforms.insert(
    noise_gate, noise_strength, position="all")
```

Trainer

- `circuit` : Q-Trainer Circuit class
- `device_name` : Pennylane-stype device name.
  - For noised simulation, use `default.mixed`
- `optimizer` : could be `Adam` , `SGD` , `ShotAdaptive` , `SPSA` , etc.
- `noise_fn` : preset noise function
- `error_mitigation_method` : Quantum error mitigation method.
  - e.g., `"zne"` => Zero-Noise Zxtrapolation
- `n_steps` : number of optimization steps

```
train_config = dict(
    device_name = 'default.mixed',
    optimizer = 'Adam',
    optimizer_config={'stepsize': 1},
    n_steps = 100,
    noise_fn = noise_fn,
    eval_freq = 5
)
```

Launch Training in One Line

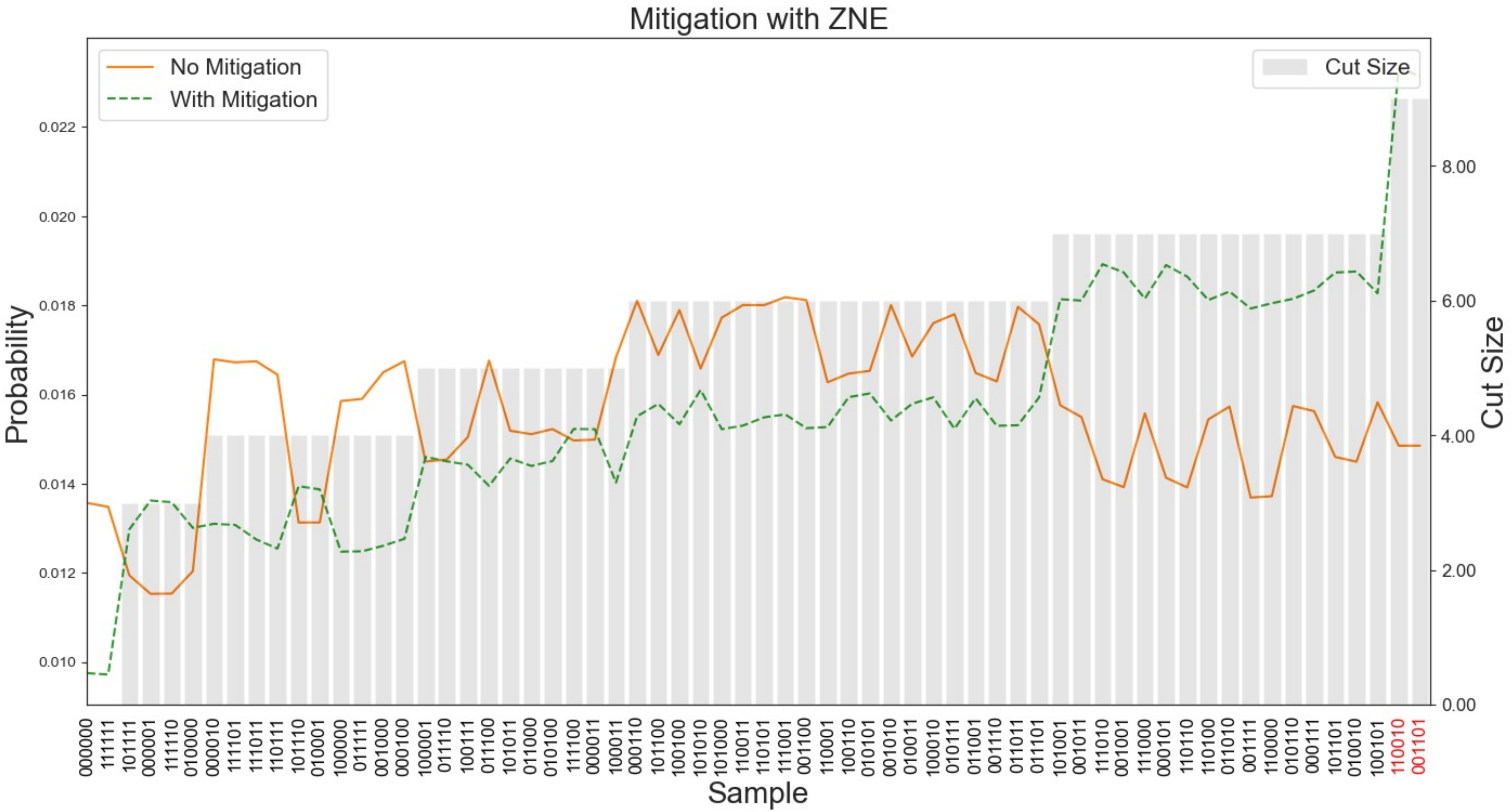
Trainer with Zero-Noise Extrapolation

```
trainer_zne = qtrainer.Trainer(circuit,
                               error_mitigation_method='zne',
                               **train_config)
```

Start training!

```
log = trainer_zne.train()
```

Train: 21% 21/100 [01:03<04:00, 3.04s/it, Cost=-5.56]



Mitigation with ZNE
No Mitigation
With Mitigation
Cut Size
Probability
0.010
0.012
0.014
0.016
0.018
0.020
0.022
Cut Size
0.00
2.00
4.00
6.00
8.00
Sample
000000
111111
101111
000001
111110
010000
000010
111101
111011
110111
101110
010001
100000
011111
001000
000100
100001
011110
100111
011100
101011
011000
010100
111100
000011
100011
000110
101100
100100
101010
101000
110011
110101
111001
001100
110001
100110
010101
001010
010011
100010
010111
011001
001110
011011
011101
101001
001011
111010
001001
111000
000101
110110
110100
011010
001111
110000
010110
000111
101101
010010
100101
110010
001101

Q-Trainer Example 2: VQE

Define a Molecular Hamiltonian

Here, we define a simple Hamiltonian following the [Pennylane tutorial]

$$H = -\sum_i X_i X_{i+1} + 0.5 \sum_i Z_i$$

```
n_qubits = 4
coeffs = [1.0] * (n_qubits - 1) + [0.5] * n_qubits
observables = [qml.PauliX(i) @ qml.PauliX(i + 1)
               for i in range(n_qubits - 1)]
observables += [qml.PauliZ(i) for i in range(n_qubits)]
H = qml.Hamiltonian(coeffs, observables)
```

Configurations

Q-Trainer's VQE Circuit

For VQE, we need to define an ansatz with initial parameters

```
n_layers = 2
init_params = [qnp.ones((n_qubits), requires_grad=True),
               qnp.ones((n_layers, n_qubits-1, 2), requires_grad=True)]
def ansatz(w1, w2):
    qml.SimplifiedTwoDesign(w1, w2, wires=range(n_qubits))
```

Now, we can construct VQE Circuit with the VQE task config (`Hamiltonian` , `n_qubits`) and the ansatz config (`ansatz` , `init_params`)

```
circuit = qtrainer.VQECircuit(Hamiltonian=H, n_qubits=n_qubits,
                              ansatz=ansatz, init_params=init_params)
```

Noise Model

```
# Depolarization noise on all gates
noise_gate = qml.DepolarizingChannel
noise_strength = 0.05
noise_fn = qml.transforms.insert(noise_gate, noise_strength,
                                 position="all")
```

Trainer

- `circuit` : Q-Trainer Circuit class
- `device_name` : Pennylane-stype device name.
  - For noised simulation, use `default.mixed`
- `optimizer` : could be `Adam` , `SGD` , `ShotAdaptive` , `SPSA` , etc.
- `noise_fn` : preset noise function
- `error_mitigation_method` : Quantum error mitigation method.
  - e.g., `"zne"` => Zero-Noise Zxtrapolation
- `n_steps` : number of optimization steps

```
train_config = dict(
    device_name = 'default.mixed',
    optimizer = 'Adam',
    optimizer_config={'stepsize': .1},
    n_steps = 100,
    eval_freq = 5,
    noise_fn = noise_fn,
)
```

Trainer with Zero-Noise Extrapolation

```
trainer = qtrainer.Trainer(circuit,
                           error_mitigation_method='zne',
                            **train_config)
```

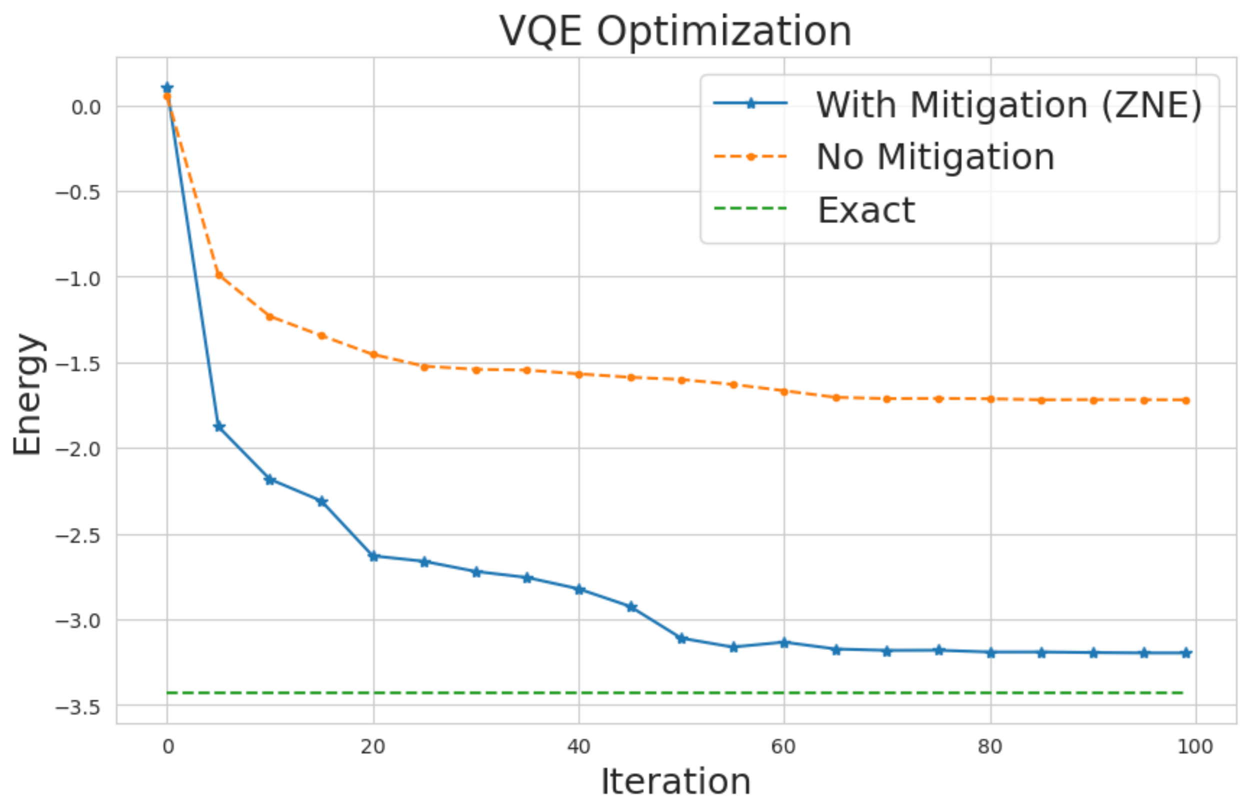
Start training!

```
log = trainer.train()
```

Train: 20% 20/100 [01:11<04:28, 3.36s/it, Cost=-2.31]

Launch Training in One Line

Comparison: *with vs. without* Error Mitigation



Local Simulators for Q-Trainer

With PennyLane as interface of Q-Trainer, various local simulators are available for training variational circuits.

Here we show a list of local simulators.

Simulator	Platform	Package	CPU/GPU	Pure/Mixed State
`default.qubit`	PennyLane	`pennylane`	CPU	Pure
`default.mixed`	PennyLane	`pennylane`	CPU	Mixed
`lightning.qubit`	PennyLane	`pennylane-lightning`	CPU	Pure
`lightning.gpu`	PennyLane	`pennylane-lightning`	GPU	Pure
`braket.local.qubit`	Braket	`amazon-braket-pennylane-plugin`	CPU	Pure/Mixed
`cirq.simulator`	Cirq	`pennylane-cirq`	CPU	Pure
`cirq.qsim`	Cirq	`pennylane-cirq`	CPU	Pure
`qiskit.basicaer`	Qiskit	`pennylane-qiskit`	CPU	Pure
`qiskit.aer`	Qiskit	`pennylane-qiskit`	CPU	Pure

Extension I: Different Local Simulators

Use Simulators/Quantum Computers hosted on AWS Braket

With PennyLane as interface of Q-Trainer, we can use various simulators/quantum computers available on AWS Braket.

Here we show a list of available devices with their Amazon Resource Names (ARNs).

- Simulators (CPUs)
  - State-Vector Simulators: for pure-state simulations
    - SV1: `arn:aws:braket:::device/quantum-simulator/amazon/sv1`
  - Tensor-Network Simulators: for pure-state simulations (faster than SV1 for sparsely-connected qubits)
    - TN1: `arn:aws:braket:::device/quantum-simulator/amazon/tn1`
  - Density-Matrix Simulators: for mixed-state simulations
    - DM1: `arn:aws:braket:::device/quantum-simulator/amazon/dm1`
- Quantum Computers (QPUs)
  - Rigetti
    - Aspen M-2 (80 Qubits): `arn:aws:braket:us-west-1::device/qpu/rigetti/Aspen-M-2`
    - Aspen M-3 (79 Qubits): `arn:aws:braket:us-west-1::device/qpu/rigetti/Aspen-M-3`
  - IonQ
    - IonQ Device (11 Qubits): `arn:aws:braket:::device/qpu/ionq/ionQdevice`

In this tutorial, we use a QAOA problem as an example to demonstrate the use of QPUs from AWS Braket.

Extension II: AWS Braket Devices

Thank you for watching this presentation

Code: https://github.com/Min-Li/qtrainer

Contact:

Min Li (minl2@Illinois.edu)

Haoxiang Wang (hwang264@Illinois.edu)

Acknowledgement: This project is supported by Unitary Fund