Appendix A Classical simulation task

It should be needed to sample one million samples, achieving the required XEB.

Fig. 3 shows a corresponding circuit example. It has $n = 2$ qubits for the initial state $|\psi_0\rangle$ and $m = 2$ cycles. The sampling process using the quantum circuit is computed as follows,

1. In the first cycle, random selected single-qubit quantum gates are first applied to all the four qubits of $|\psi_0\rangle$; Then, the double-qubit quantum gates are applied to $\boldsymbol{R}_1^0 |\psi_0\rangle_{0,1}$ and $\boldsymbol{R}_1^2 |\psi_0\rangle_{2,3}$, respectively, and obtain $|\psi_1\rangle$;
2. In the second cycle, random selected single-qubit quantum gates are first applied to all the four qubits of $|\psi_1\rangle$; Then, the double-qubit quantum gates are applied to $\boldsymbol{R}_2^1 |\psi_1\rangle_{1,2}$, and obtain $|\psi_2\rangle$;
3. Next, random selected single-qubit quantum gates are first applied to all the four qubits of $|\psi_2\rangle$, and obtain $\boldsymbol{R}_2^j |\psi_1\rangle_j$, $j = 0, 1, 2, 3$;
4. Last, we have a measurement $\alpha_{i_1 i_2 i_3 i_4} = \langle i_1 i_2 i_3 i_4 |\psi_3\rangle$. The sampled output is a bit-string $\alpha_{i_1 i_2 i_3 i_4}$.

Algorithm 1 Random circuit sampling

Input: initial state $|\psi_0\rangle$, number of cycles m, single-qubit quantum gate $\{\sqrt{\boldsymbol{X}}, \sqrt{\boldsymbol{Y}}, \sqrt{\boldsymbol{W}}\}$, double-qubit quantum gate $\boldsymbol{U}_i$,
for $i = 1, ..., m$ **do**
  for $j = 0, ..., n-1$
    $\boldsymbol{R}_i^j \leftarrow$ randomly select from $\{\sqrt{\boldsymbol{X}}, \sqrt{\boldsymbol{Y}}, \sqrt{\boldsymbol{W}}\}$,
    $|\psi_i\rangle_j = \boldsymbol{R}_i^j |\psi_{i-1}\rangle_j$,
  end for
  for $j = (i+1)\%2, ..., n/2 - 1$
    $p \leftarrow$ compute the manipulate quantum bit index $2j + (i+1)\%2$
    $|\psi_i\rangle_{p,p+1} = \boldsymbol{U}_i^p |\psi_{i-1}\rangle_{p,p+1}$,
  end for
end for
for $j = 0, ..., n-1$
  $\boldsymbol{R}_{m+1}^j \leftarrow$ randomly select from $\{\sqrt{\boldsymbol{X}}, \sqrt{\boldsymbol{Y}}, \sqrt{\boldsymbol{W}}\}$,
  $|\psi_{m+1}\rangle_j = \boldsymbol{R}_{m+1}^j |\psi_m\rangle_j$,
end for
Obtain a measurement by $\alpha_{i_1 i_2 ... i_n} = \langle i_1 i_2 ... i_n |\psi_{m+1}\rangle$,
Output: a bit-string $\alpha_{i_1 i_2 ... i_n}$.

Quantum circuits: There have been many quantum circuits proposed as follows,

- Sycamore quantum [3]: It consists of 53 qubits and 20 cycles. For the Boson sampling problem, it only needs 200 seconds to finish this task, while it needs $10,000$ years for classical simulation.
- Jiu Zhang [60]: It consists of 74 qubits. For the Gaussian Boson Sampling problem, it can use 200s to finish up to a million times compared with classical simulations.
- Zuchongzhi [61]: It has 60 qubits with the number of 24. The achieved sampling task is about 6 orders of magnitude more difficult than that of Sycamore in the classic simulation.

Appendix B Data Generators and Reinforcement Learning Environments.

B.1 Tensor Network Representations

We use the unidirectional graph to represent the tensor network. Specifically, for a given tensor network with n tensors, we use a symmetric $n \times n$ matrix $\boldsymbol{M}$ to represent the dimensional relationships between tensors, which is named the dimension matrix. Specifically, if the tensors with index i and j are connected with a shared dimension as d_{ij}, then we set $\boldsymbol{M}_{ij} = \boldsymbol{M}_{ji} = d_{ij}$, otherwise, it is

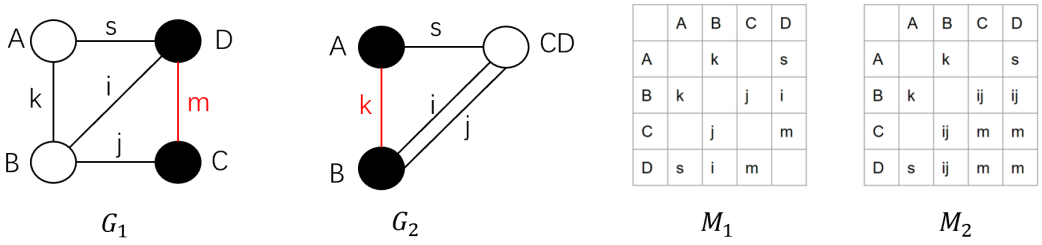


Figure 8: Example of the tensor network contraction environment.

set as 0. We also denote a connection matrix $\boldsymbol{C}$, where $\boldsymbol{C}_{ij} = \boldsymbol{C}_{ji} = 1$ indicates the connection between the i-th tensor and j-th tensor.

B.2 Tensor Network Contraction

We employ tensor network contraction as the underlying computational framework in our quantum circuit simulation. Given a specific tensor network structure and a specified contraction order, the environment is designed to provide the number of multiplications required during the tensor contraction process.

Specifically, as illustrated in Figure 8, we consider a tensor network comprising 4 tensors connected in a grid structure. When performing a tensor contraction between tensors C and D, it is necessary to compute the number of multiplications involved. From the unidirectional graph representation, tensor C has dimensions $M \times J$, tensor D has dimensions $S \times I \times M$, and a shared edge with dimension M exists.

To contract tensors C and D along the edge with dimension M, the number of multiplications can be computed as $\frac{(MJ)\times(SIM)}{M}$. This expression takes into account the multiplication of the dimensions MJ from tensor C and SIM from tensor D, divided by the shared dimension M.

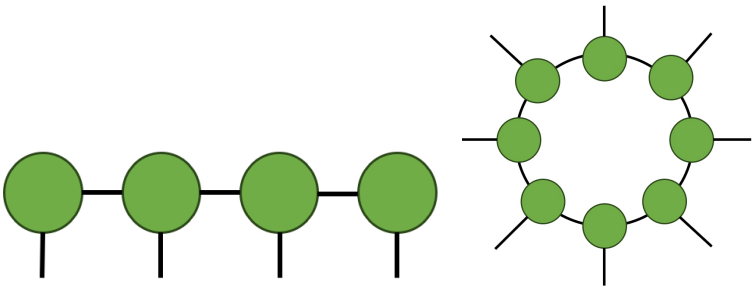
Upon completing the contraction, it is essential to update the unidirectional graph representation. The contracted tensor formed by the contraction of C and D retains an independent edge with dimension M ($\boldsymbol{M}_{33} = \boldsymbol{M}_{34} = \boldsymbol{M}_{43} = \boldsymbol{M}_{44} = M$). The connected edges between the contracted tensor (C or D) and other tensors are determined by multiplying the original edge dimensions between them ($\boldsymbol{M}_{23} = \boldsymbol{M}_{24} = \boldsymbol{M}_{32} = \boldsymbol{M}_{42} = IJ$). This update to the unidirectional graph ensures accuracy and reflects the changes resulting from the contraction step.

By incorporating the update of the unidirectional graph and computing the number of multiplications, we complete the current simulation step.

B.3 Verification

Unified representation: We use a tuple consisting of the contracted tensor indices to represent the tensor contraction order. The resulting tensor is labeled as a new index. As shown for example in Fig. 5, we first write down $(3, 4)$ for the first contraction operation. We label the generated tensor 34 as 5. Next, we use $(1, 2)$ to label the second tensor contraction operation and 6 to label the resulting tensor 12. Last, we contract the tensor 12 and 34, where we use $(5, 6)$ to represent the contraction record. Thus, we have the contraction order $\{(3, 4), (1, 2), (12, 34)\}$

Calculating the number of multiplications: Given the tensor contraction order, we will compute the number of involved multiplications. In Fig. 5, using different optimizers, we can get a specific contraction order, like $\{(3, 4), (1, 2), (12, 34)\}$. We first contract the 3-th and 4-th tensor, of which the number of multiplication is $IJMS$. Then, for the contraction order $(1, 2)$, we have the number of multiplication as $SKIJ$. Last, for the contraction order $(12, 34)$, we have the number of multiplication as SIJ.



(a) Tensor-train network. (b) Tensor-ring network.

Figure 9: Illustration of the tensor train and tensor ring network.

B.4 Datasets for Different Tensor Networks

In this section, we will first introduce the data structure used to describe the relationships between different tensor nodes. Then, we will demonstrate the generation process of the dataset for different tensor networks.

Data Structure: The tensor network can be represented as an undirected graph $\boldsymbol{G} = \{V, E, w\}$ and is amenable to be stored using the adjacent table, which can be defined as follows (C-format),

```
class TensorNetwork {
    int V; // number of tensor nodes
    int E; // number of edges
    Queue[] adj; // adjacent nodes for each node

    TensorNetwork(int V) { // init the graph
        this.V = V; this.E = 0;
        this.adj = new Queue[V];
        for (i = 0; i < V; i++) {
            this.adj[i] = new Queue[];
        }
    }
    void AddEdge(int v, int w) { // add the edges
        if (w > v) v, w = w, v;
        this.adj[v].enqueue(w);
        this.E++;
    }
    void Build(...); //differs in different tensor networks
}
```

Specifically, we use V and E to save the number of tensor nodes and edges, respectively. The variable adj is a queue with flexible length, where $adj[j]$ is also a queue to save the adjacent tensor nodes. Given the total number of tensor nodes V, $TensorNetwork(V)$ is used to initialize the undirected graph, where each tensor node is assigned an empty queue. The $addEdge(\cdot, \cdot)$ is invoked to add the connected relationship between the connected tensor nodes v and w. We make a simplified assumption that only the larger index w can be added to the adjacent queue of v. Different tensor network differs in the implementation of the $Build(...)$ function, which invokes the $addEdge(\cdot, \cdot)$ depending on the specified tensor network structure.

Tensor-Train Network or Matrix Product States: The matrix product state (MPS) [41, 56] or tensor-train (TT) represents a tensor as a chain-like contraction of third-order tensors with the head and tail as matrices. The tensor diagram of MPS/TT is shown in Fig. 9(a).

MPS has been widely used in modeling quantum circuits such as [47, 16, 45]. In the $Build(...)$ function, for each tensor node i, we need to use the $AddEdge(i, i+1)$ to add the $i+1$-th node to the adjacent lists of i-th node, $i < |V| - 1$. We can set up different number of tensor nodes N to generate the dataset for MPS tensor network.

```
    void TensorNetwork::Build(){ // Build for MPS
        for (i = 0; i < this.V-1; i++) this.AddEdge(i, i+1); }
```

The generation code for the tensor-train network is saved in https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits/blob/main/datasets/mps/mps_generate.py.

Tensor ring network: The tensor ring (TR) represents a high-order (or high-dimensional) tensor by a sequence of 3rd-order tensors that are multiplied circularly, whose tensor diagram notation can be represented in Fig. 9(b).

The tensor-ring network has been utilized to simulate the quantum circuit in [44].

The main difference between the tensor ring and the MPS tensor network is that the first and last tensor nodes in tensor ring network are also connected. Thus, only a minor modification of MPS generation can be applied to generate the tensor ring network.

```
    void TensorNetwork::Build(){ // Build for tree tensor
        for (int i = 0; i < this.V; i++)
            this.AddEdge(i, (i+1)%this.V); }
```

The generation code for the tensor-ring network is saved in https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits/blob/main/datasets/tr/tr_generate.py.

Tree Tensor Network: Tree tensor network (TTN) [37, 52, 55] or Hierarchical Tucker (HT) is a generalization of MPS that encodes a tree entanglement structure. The diagram notation of a TTN can be represented in Fig. 10.

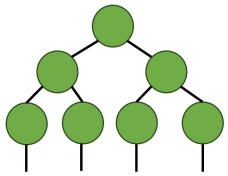


Figure 10: Tree tensor network.

Tree tensor network has been used to model the quantum [24, 50], and quantum chemistry [36, 37].

Tree tensor network is a fully binary tree structure, of which the number of tensor nodes depends on the height. we reload the initialization function and write the build function as follows,

```
    TensorNetwork::TensorNetwork(int H) { // init the graph
        this.H = H; this.V = pow(2,H) - 1; this.E = 0;
        this.adj =  new Queue[V];
        for (i = 0; i < V; i++) {
            this.adj[i] = new Queue[];
        }
    }
    void TensorNetwork::Build(){ // Build for tensor ring
        this.AddEdge(0, 1); this.AddEdge(0, 2);
        for (h = 1; h < this.H; h++){
            for (v = pow(2, h-2); v < pow(2, h-1) + 1; v++){
                this.AddEdge(v, 2v); this.AddEdge(v, 2v+1);
            }
        }
    }

```

The generation code for the tree tensor network is saved in https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits/blob/main/datasets/tree/tree_generate.py.

PEPS Network: The PEPS (projected entangled pair state) tensor network [49, 58] generalizes MPS from a one-dimensional network to a network on an arbitrary graph, whose tensor diagram notation can be represented in Fig. 11.

Some work applies the PEPS network to quantum circuits [43] and quantum systems [40, 31].

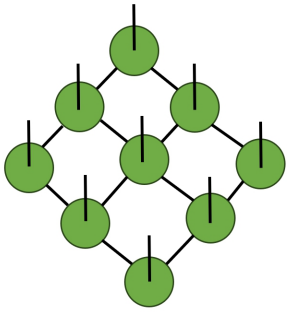


Figure 11: PEPS tensor network.

The generation code for the PEPS network is saved in https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits/blob/main/datasets/peps/peps_generate.py.

MERA Network: The MERA (Multiscale Entanglement Renormalization Ansatz) [14] tensor network colleagues as a refinement of the MPS and PEPS. It has a hierarchical structure, with layers of tensors representing increasingly coarse-grained degrees of freedom, which can be represented in Fig. 12.

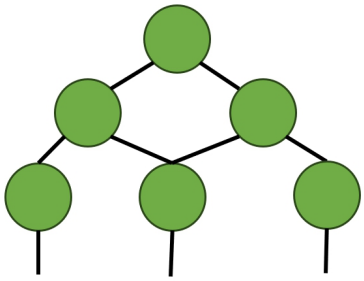


Figure 12: MERA tensor network.

Some work applies the MERA network to quantum circuits [1, 20, 1, 30].

The generation code for the MERA tensor network is saved in https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits/blob/main/datasets/mera/mera_generate.py.

Google's Sycamore Circuits: Some work has put efforts into transforming the Sycamore circuits into tensor representations. The provided data is usually organized into the adjacent graph structure, while the indices of the i-th row are the indices of connected tensors to i-the tensor. The tensor network representation of the Sycamore circuits corresponds to a complicated net structure. To map the Sycamore circuits into the tensor network environment, we need to iteratively read each line of the given Sycamore file and fill it up with corresponding dimension information.

The generation code for the Sycamore circuit is saved in https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits/tree/main/datasets/sycamore.

B.5 Environment

We provide the gym-environment for the tensor network contraction problem in our classical simulation of quantum circuits. The code is provided in https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits/blob/main/rl/mps/env.py for tensor-train tensor network and https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits/blob/main/rl/sycamore/env.py for Sycamore circuits.

Specifically, the gym-environment for the tensor network contraction problem in our classical simulation of quantum circuits is designed as follows,

- Init/Restart: initialize the states of environments, including the number of nodes, the number of edges, *etc.*
- Step: Given the tensor network contraction order, compute the number of involved multiplications as the reward. The rewards are represented in log scale.

The implementations can be found in *init()*, and *get_log10_multiple_times()* functions of our *env.py*.

B.6 High-Performance Reinforcement Learning for TNCO

Mapping onto the K-spin Ising Model: We use the K-spin Ising model to formulate the TNCO problem as follows,

$$\begin{aligned} H(x) &= \sum_{i=1}^{N-1}\left[\left(2-\sum_{u=1}^{N-i}x_{u,i}\right)^2+\sum_{u=1}^{N}\sum_{v=1}^{N}J_{u_i,v_i}x_{u,i}x_{v,i}\right], \\ H(x^1,...,x^K) &= \sum_{k=1}^{K}H(x^k)+\sum_{k=1}^{K}\sum_{i_1\in V^1}...\sum_{i_1\in V^k}J_{i_1\ldots i_k}x^1_{i_1}...x^k_{i_k}, \end{aligned} \tag{12}$$

where we denote the $N(N-1)$ spin as $x_{u,j}$, u as the tensor and j denotes its order in the TNCO path.

We use the variational annealing methods to solve the K-spin Ising model problem of TNCO. Specifically, we minimize the KL divergence between the transition distribution of x to x' with the target Boltzmann distribution as follows,

$$\begin{aligned} \mathcal{D}_{KL}(q_\theta||p) &= \sum_x q_\theta(x\to x')\ln\left(\frac{q_\theta(x\to x')}{p(x\to x')}\right) \\ &= \sum_x q_\theta(x\to x')\ln q_\theta(x\to x') \\ &+ \frac{q_\theta(x\to x')(H(x)-H(x'))}{T}+\ln Z_x, \end{aligned} \tag{13}$$

where T is the temperature, q_θ is the distribution of TNCO path $x\to x'$ parameterized by θ, $p(x'|x)$ is the transition distribution of Boltzmann distribution $p(x'|x)=\frac{\exp(-\frac{H(x')}{T})}{Z}$, $Z_x=\sum_{x'}^{H(x')<H(x)}e^{-\frac{H(x')}{T}}$. During the learning process, we gradually anneal the temperature to optimize (13).

Parallel data sampling: We use RNN parameterized by θ as the policy network to model the transition probability $q_\theta(x\to x')$. We input the K-spin representation of tensor contraction order, $(x_1,..x_K)$, sequentially from left to right, and compute the transition probability as follows,

1. Randomly initialize the hidden variable h_1;
2. Input x_1 and h_1 to the RNN, and output the next hidden variable h_2 and a transition probability $q_\theta(x_1)$;
3. Input x_2 and h_2 to the RNN, and output the next hidden variable h_3 and a transition probability $q_\theta(x_1\to x_2)$;
4. .....
5. Input x_K and h_K to the RNN, and output the next hidden variable h_{K+1} and a transition probability $q_\theta(x_{K-1}\to x_K)$.

Then, by sampling the tensor contraction ordering trajectories, $(x_1,x_2,...,x_K)$, we can optimize (13) to train the RNN. The sampling process can be batched onto multiprocess to achieve high-performance data sampling, thus alleviating the performance bottleneck of reinforcement learning training.

Parallel training: We initialize multiple optimizers in parallel to learn to optimal tensor contraction order. Specifically, at the t-th iteration

1. We parallelly sample N tensor contraction ordering from the TNCO environment and store them in the reply buffer.
2. For the optimizer i -th, we first freeze $\rho_{t,i}$ parameters, $0 < \rho_i < 1$, then sample data from replay buffer, and compute the loss function (13) with the temperature $T_{t,i}$, independently.
3. For i-th optimizer, we compute the gradient and use LSTM to learn to optimize the parameter.

We vary the parameter masked ratio ρ and temperature T by increasing the number of iterations and initialized all the parameters differently to achieve swarm intelligence integrated with the curriculum learning.

Appendix C Accessibility, Usage, License, and Maintenance

Accessibility: All the code, dataset, and tensor network contraction orderings, including the Sycamore circuits, can be found in our open source project `https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits` without personal request.

Dataset generation: We generate the classical simulation of quantum circuits synthetically using `https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits/tree/main/datasets`.

For example, we run `https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits/blob/main/datasets/mps/mps_generate.py` to generate quantum circuits based on the tensor train, where V in code controls the number of nodes in generated data.

Code organization: We implement the baseline methods using Opt-einsum and Cotengra to search for the tensor network contraction orderings. The codes for calling these solvers are provided in `https://github.com/XiaoYangLiu-FinRL/RL4QuantumCircuits/tree/main/baseline`. We provide two methods to solve each type of quantum circuit, like tensor train-based quantum circuits.

Usage: To run the baseline methods, execute "python cotengra.py" or "python opt_einsum.py", where the variable n is the number of nodes. To generate the dataset, please execute "python generate.py", where the variable V in the central part is the number of tensor nodes.

License: MIT License.

Maintenance: On GitHub, we keep updating our codes, merging pull requests, and fixing bugs and issues. We welcome contributions from community members and researchers.