Quarl: A Learning-Based Quantum Circuit Optimizer

ZIKUN LI, Carnegie Mellon University, USA
JINJUN PENG, Columbia University, USA
YIXUAN MEI, Carnegie Mellon University, USA
SINA LIN, Microsoft, USA
YI WU, Tsinghua University, China
ODED PADON, VMware Research, USA
ZHIHAO JIA, Carnegie Mellon University, USA

Optimizing quantum circuits is challenging due to the very large search space of functionally equivalent circuits and the necessity of applying transformations that temporarily decrease performance to achieve a final performance improvement. This paper presents Quarl, a learning-based quantum circuit optimizer. Applying reinforcement learning (RL) to quantum circuit optimization raises two main challenges: the large and varying action space and the non-uniform state representation. Quarl addresses these issues with a novel neural architecture and RL-training procedure. Our neural architecture decomposes the action space into two parts and leverages graph neural networks in its state representation, both of which are guided by the intuition that optimization decisions can be mostly guided by local reasoning while allowing global circuit-wide reasoning. Our evaluation shows that Quarl significantly outperforms existing circuit optimizers on almost all benchmark circuits. Surprisingly, Quarl can learn to perform rotation merging—a complex, non-local circuit optimization implemented as a separate pass in existing optimizers.

CCS Concepts: • **Software and its engineering → Compilers**; • **Computing methodologies → Reinforcement learning**; • **Hardware → Quantum computation**.

Additional Key Words and Phrases: Quantum compilers

ACM Reference Format:
Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. 2024. Quarl: A Learning-Based Quantum Circuit Optimizer. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 114 (April 2024), 28 pages. https://doi.org/10.1145/3649831

1 INTRODUCTION

Quantum computing presents a novel paradigm that enables significant acceleration over classical counterparts in a wide range of applications, such as quantum simulation (Cao et al., 2019), integer factorization (Monz et al., 2016), and machine learning (Biamonte et al., 2017). However, programming quantum computers is a challenging task due to the scarcity of qubits and the diverse forms of noise that affect the performance of near-term intermediate-scale quantum (NISQ) devices.

Quantum programs are commonly represented as *quantum circuits*, such as the one shown in Figure 1, where each horizontal wire represents a qubit and boxes on these wires represent quantum

Authors' addresses: Zikun Li, Carnegie Mellon University, Pittsburgh, PA, USA, zikunl@andrew.cmu.edu; Jinjun Peng, Columbia University, New York, NY, USA, mail@co1in.me; Yixuan Mei, Carnegie Mellon University, Pittsburgh, PA, USA, meiyixuan2000@gmail.com; Sina Lin, Microsoft, Mountain View, CA, USA, silin@microsoft.com; Yi Wu, Tsinghua University, Beijing, China, jxwuyi@gmail.com; Oded Padon, VMware Research, Palo Alto, CA, USA, oded.padon@gmail.com; Zhihao Jia, Carnegie Mellon University, Pittsburgh, PA, USA, zhihao@cmu.edu.

ACM 2475-1421/2024/4-ART114
https://doi.org/10.1145/3649831

gates. To enhance the success rate of executing a circuit, a common form of optimization is applying *circuit transformations*, which replace a subcircuit matching a specific pattern with a functionally equivalent subcircuit that has better performance (e.g. fidelity, depth).

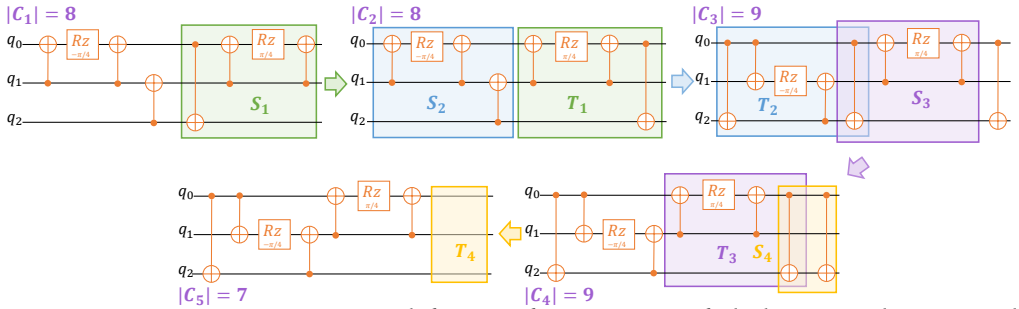


Fig. 1. A quantum circuit optimization with four transformations, one of which temporarily increases the cost. Each arrow indicates a transformation, whose source and target sub-circuits are highlighted by boxes in the same color.

Prior research has proposed two approaches for performing circuit transformations on an input circuit. The first approach is the use of *rule-based* strategies, which are employed by many quantum compilers such as Qiskit (Aleksandrowicz et al., 2019), t|ket⟩ (Sivarajah et al., 2020), and Quilc (Skilbeck et al., 2020). These strategies involve the greedy application of a set of circuit transformations that are manually designed by quantum computing experts to improve the performance of quantum circuits. The second approach, as introduced in recent works (Pointing et al., 2021, Xu et al., 2022c,a), is a *search-based* approach that explores a search space of circuits that are functionally equivalent to the input circuit. For instance, Quartz (Xu et al., 2022a) automatically generates and verifies circuit transformations for a given gate set, which preserves equivalence but may not necessarily improve performance. To optimize an input circuit, Quartz employs a cost-based backtracking search algorithm to apply these transformations and discover an optimized circuit. Although existing approaches improve the performance of quantum circuits, they are limited by the following challenges in transformation-based quantum circuit optimization.

Planar optimization landscape. The set of circuits that can be reached from an input circuit by iteratively applying verified, equivalence-preserving transformations comprises the search space in quantum circuit optimization. However, finding the optimal circuit is challenging due to the size of the space, which makes exhaustive exploration infeasible, and the inability of the cost function (derived from a selected performance metric) to provide enough guidance for a greedy approach. This scenario is referred to as a *planar optimization landscape* since the path from one circuit to another with lower cost often contains many steps in which the cost remains unchanged. (Plateaus of this sort are also present in classic program optimization (Koenig et al., 2021).)

To illustrate the challenge of discovering an optimal circuit in the search space, we analyze the search space of a relatively small circuit `barenco_tof_3` (Nam et al., 2018) which is implemented with 58 gates in the Nam gate set (*CNOT*, *X*, *H*, *Rz*). We consider the 6,206 transformations discovered by Quartz (Xu et al., 2022a) for the Nam gate set and exhaustively find all circuits reachable within seven transformations. Of these roughly 162 million circuits, we randomly sample 200 circuits and analyze the optimization landscape around them. Specifically, for each sampled circuit C, we perform a breadth-first search (BFS) to determine the shortest path from C to a circuit C' with a lower cost (using the total number of gates as the cost function). That is, we determine the radius of the optimization landscape around C. The BFS is performed up to a radius of 6, so we either

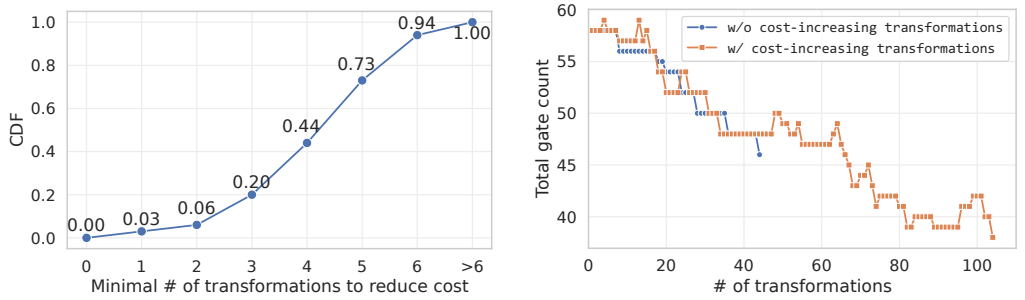


Fig. 2. An analysis on the search space of `barenco_tof_3`. (a) CDF of the minimal number of transformations needed to reduce gate count. (b) The optimization trace of the best discovered circuits when including and excluding cost-increasing transformations.

find the exact radius or conclude that it is greater than 6. Notably, it is guaranteed that all of the 200 circuits can be optimized because their cost is greater then the optimal. The results are summarized in Figure 2. For more than half of the sampled circuits, reducing the total gate count requires more than 4 transformations, and 6% of the sampled circuits require more than 6 transformations to improve performance. While we could exhaustively explore transformation sequences of length 7 for `barenco_tof_3`, this would not be practical for larger circuits with hundreds of gates or more. Thus, in the absence of guidance from the cost function, it is natural to consider a learning-based approach to guide the application of transformations towards a lower-cost circuit.

Cost-increasing transformations. Another challenge in optimizing quantum circuits is the need to use transformations that may temporarily increase the cost. As illustrated in Figure 1, the gate count rises from 8 to 9 before ultimately decreasing to 7 through subsequent transformations. Such cost-increasing transformations are vital for achieving optimal circuits. In Figure 2b, for instance, the best circuit for `barenco_tof_3` without these transformations has 46 gates. In contrast, utilizing them can lower the count to 36. Identifying the appropriate time and place for these transformations is complex, given their value is only realized when paired with cost-decreasing transformations.

Our approach. This paper presents Quarl, a learning-based quantum circuit optimizer which utilizes reinforcement learning (RL) to guide the application of quantum circuit transformations. In the RL task, each circuit is defined as a state, and each application of a transformation is considered an action. By adopting this formulation, Quarl can learn to identify circuit optimization opportunities and perform long sequences of transformations to realize them.

The first challenge we must address in applying RL to circuit optimization is the large action space. For example, when learning to apply the 6,206 transformations discovered by Quartz (Xu et al., 2022a) on a thousand-gate circuit, there may be millions of possible actions (i.e., ways to match a transformation to a subcircuit of the current circuit). Such a large action space would degrade the training efficiency of most RL algorithms. To deal with this issue, we propose a *hierarchical action space* that uses a gate-transformation pair to uniquely identify a potential transformation application, so that the RL agent first chooses a gate and then a transformation to apply. This decomposition allows Quarl to use two much smaller sub-action spaces, enabling effective training. To effectively train the RL agent to select both a gate and a transformation, we propose *hierarchical advantage estimation* (HAE), which allows Quarl to train two policies with a single *actor-critic* architecture. Quarl combines HAE with *proximal policy optimization* (PPO) (Schulman et al., 2017) to jointly train the gate- and transformation-selecting policies.

The second challenge for RL-based quantum circuit optimization is state representation. An RL application typically represents states as vectors in a fixed, high-dimensional space. Unlike most RL tasks whose states have relatively uniform structure, a state in our setting is a quantum circuit whose size and topology depend on the input circuit and change at each step during the optimization process, making it non-trivial to design a fixed, uniform state representation. A key insight behind Quarl's approach is to leverage the *locality* of quantum circuit transformations, that is, the decision of applying a transformation at a certain location is largely guided by the local environment at that location. Based on this insight, Quarl uses a *graph neural network* (GNN)-based approach to represent the local environment of each gate. Although each representation encodes the local environment of only one gate, combining all gate representations allows Quarl to identify optimization opportunities across the entire circuit. A key advantage of our learned gate-level representations is that they are independent to the circuit size and generalize well to unseen circuits. By combining this local decision making with global circuit-wide fine-tuning, we aim to achieve a balance of both local and global guidance for the optimization process.

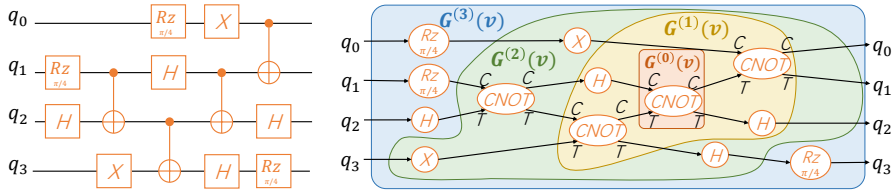
The evaluation results show the superior performance of Quarl over existing optimizers on almost all circuits. On the Nam gate set, Quarl achieves an average reduction of 35.2% and 32.5% in total gate count and `CNOT` gate count, respectively (geometric mean), while the best existing optimizers achieve reductions of only 31.0% and 25.4%. On the IBM gate set, Quarl achieves an average reduction of 36.6% and 21.3% in total gate count and `CNOT` gate count, respectively, while the best existing optimizers achieve reductions of 20.1% and 7.7%. Furthermore, Quarl improves circuit fidelity by up to 4.84× (with an average fidelity improvement of 1.37×) on the IBM gate set, while the best existing optimizer only improves circuit fidelity by 1.07×. Notably, the evaluation also reveals that Quarl can automatically discover rotation merging, a non-local circuit optimization, from its own exploration, while existing optimizers require manual implementation of this technique.

The remainder of this paper is organized as follows. Section 2 provides background information on transformation-based quantum circuit optimization and RL. Section 3 outlines the key challenges associated with applying RL to quantum circuit optimization and describe our approach for addressing them. The main technical contributions of this paper are presented in Section 4 that presents Quarl's neural architecture, and Section 5 that presents Quarl's training methodology. Empirical evaluations are presented in Section 6, and related work are discussed Section 7.

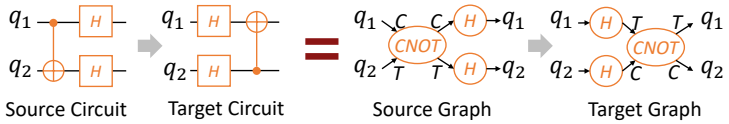
2 BACKGROUND

Graph representation of quantum circuits. We adopt the graph representation of quantum circuits from prior work (Xu et al., 2022a). As illustrated in Figures 3a and 3b, a circuit C is represented as a directed acyclic labeled graph $G = (V, E)$. Each gate with q qubits is represented as a node $v \in V$ with q in- and out-edges, and each edge $e \in E$ represents a connection of two adjacent gates on a qubit. Nodes are labeled with gate types, and edges are labeled to distinguish different qubits of a multi-qubit gate (e.g., the control and target qubits of a `CNOT` gate).

Transformation-based circuit optimization. A common form of optimization of quantum circuits is *circuit transformation* which replaces a subcircuit matching a specific pattern with a distinct equivalent subcircuit. In the graph representation, a subcircuit corresponds to a convex subgraph (Xu et al., 2022a, p.8). A convex subgraph of a directed graph refers to a subset of the graph's vertices along with the directed edges that connect these vertices, possessing a specific property: for any pair of vertices in the subgraph, if there is a directed path in the original graph that connects these two vertices, then all the vertices along this path are also part of the subgraph. A transformation is represented as a pair of connected graphs (G, G') representing equivalent circuits as illustrated in Figure 3c , and applying it to a circuit C (with corresponding graph representation G_C) amounts to



(a) Circuit representation. (b) Graph representation.



(c) Circuit transformation and its graph representation.

Fig. 3. A quantum circuit (a) and its graph representation (b). For each CNOT, C and T correspond to the control and target qubits. $G^{(k)}(v)$ shows the k-hop neighborhood (see Section 4.1) for the CNOT gate identified by $G^{(0)}(v)$. (c) illustrates a circuit transformation and its graph representation.

finding a convex subgraph of G_C that is isomorphic to G and replacing it by G' to yield a new circuit C'. For a given transformation (G, G') and a circuit C, multiple subgraphs of C may be isomorphic to G. Yet, by adding a constraint that requires a specific gate g_C in C to correspond to a gate g_G in G, the matching subgraph can be determined uniquely. It's essential to note that our design leverages the fact that G_C, G, and G' are all *labeled* graphs (e.g., the control and target qubit of a CNOT gate carry different labels), which allows us to differentiate between the in- and out-edges of a node and guarantees uniqueness of a match.

Reinforcement learning (RL). RL is a class of machine learning algorithms that focus on sequential decision making problems. An RL problem is formalized as a Markov decision process defined by a tuple (S, A_s, P, r, γ), where S is the state space of the environment, A_s is the action space for the agent at state $s \in S$, $P(s'|s, a)$ defines the probability of the state transiting from s to s' if the agent takes the action a, $r(s', s, a)$ defines the immediate reward of the transition from state s to state s' by action a, and $\gamma \in (0, 1)$ is the discounted factor to prioritize immediate rewards over future rewards. Each state-action pair is a *step*, and a sequence of steps is a *trajectory*, denoted as $\tau = (s_0, a_0, s_1, a_1, \dots)$. The *return* of a trajectory is the discounted cumulative reward along the trajectory, given by $R_\tau = \sum_{t=0} \gamma^t r_t$, where $r_t = r(s_{t+1}, s_t, a_t)$. We use R_t for the discounted cumulative reward starting from step t in the trajectory, that is, $R_t = \sum_{t'=t} \gamma^{t'-t} r_t$.

RL agents aim to learn a *policy* to maximize the return through trial and error, by first collecting trajectories and then optimizing the policy based on the actions taken and the returns observed. Formally, a *policy* is a function π that maps each state to a probability distribution over valid actions. A policy induces a probability distribution over trajectories starting at a given state (i.e., trajectories *generated* by the policy), where a_t is sampled according to $\pi(s_t)$ and s_{t+1} is sampled according to $P(s_{t+1}|s_t, a_t)$. A policy parameterized by θ is denoted as $\pi(\cdot|s; \theta)$ or π_θ. The learning objective in RL is to maximize the expected return of trajectories generated by the policy π_θ, i.e., $J(\theta) = E_{\tau\sim\pi_\theta}[R_\tau]$.

Policy gradient methods optimize this objective by gradient ascent over $J(\theta)$ w.r.t. θ. By the policy gradient theorem (Sutton et al., 1999), an equivalent formulation in the form of a loss function can be expressed as $L^{PG}(\theta) = E_{\tau\sim\pi_\theta}[\sum_t log\pi(a_t|s_t; \theta)A(s_t, a_t)]$, whose gradient is equivalent to $\nabla J(\theta)$ and can be estimated by sampling τ. $A(s_t, a_t)$ in $L^{PG}(\theta)$ is the *advantage*, defined as $A(s, a) = Q^\pi(s, a) - V^\pi(s)$, where $Q^\pi(s, a) = E[R_\tau|s_0 = s, a_0 = a]$ is the Q-function (i.e., the expected

return starting from state s and taking action a), and $V^\pi(s) = E[R_\tau | s_0 = s]$ is the expected return starting from state s. Advantage measures how much better the agent can get by taking a specific action at a state than average. By gradient ascent, actions are *reinforced* based on their advantages (i.e. the probability of an action increases if its advantage is positive, and drops otherwise). The empirical advantage $\hat{A}_t$ is often estimated as $\hat{A}(s_t, a_t) = R_t - V^\pi(s_t)$ over a sampled trajectory, where R_t is the return, which is a sampled estimation of $Q^\pi(s_t, a_t)$, and $V^\pi(s_t)$ is typically approximated by training another neural network. Such a framework is also referred to as the *actor-critic* method, where actor denotes the policy and critic denotes the value network (Mnih et al., 2016).

Proximal policy optimization (PPO). PPO is a variant of policy gradient methods that achieve state-of-the-art performance in various applications (Schulman et al., 2017). PPO uses an clipped surrogate objective: $L^{CLIP}(\theta) = \underset{\tau \sim \pi_{\theta_{old}}}{E} \left[\sum_t min(\rho_\theta(s_t, a_t)\hat{A}_t, clip(\rho_\theta(s_t, a_t), \epsilon)\hat{A}_t) \right]$, where $\rho_\theta(s, a) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$, $\hat{A}_t$ denotes the estimated advantage $\hat{A}(s_t, a_t)$ at step t over trajectory τ generated by $\pi_{\theta_{old}}$, and $clip(\rho, \epsilon) = \min(1 + \epsilon, \max(1 - \epsilon, \rho))$ is the clip function used by PPO (Schulman et al., 2017, Eq. 7).

3 CHALLENGES AND HIGH-LEVEL APPROACH

Applying reinforcement learning to optimize quantum circuits presents several challenges unique to this problem setting. This section presents these challenges and the key ideas Quarl uses to overcome them. In the sequel, we assume a fixed gate set, set of equivalence-preserving transformations, and cost function. We formulate quantum circuit optimization as a Markov decision process (S, A_s, P, r, γ) as follows. S is the set of circuits over the given gate set. For a circuit $C \in S$, A_C includes all valid applications of transformations on C. Applying a transformation a to a circuit C deterministically defines a new circuit C'; therefore we let $P(C'|C, a) = 1$ and $P(C''|C, a) = 0$ for $C'' \neq C'$. Finally, the reward function is given by the cost difference between the circuits before and after a transformation: $r(C, a) = \textsc{Cost}(C) - \textsc{Cost}(C')$.

3.1 Challenge 1: Action Space

Directly applying existing policy gradient methods to our setting requires the RL agent to learn a policy that can simultaneously select a transformation and a subcircuit to apply the transformation for a given circuit. However, learning such a policy is challenging due to the very large action space. For example, when learning to apply the 8,664 transformations discovered by Quartz (Xu et al., 2022a) on a thousand-gate circuit, there can be up to millions of actions (i.e., possible applications of the transformations) for a given state (i.e., circuit). The large action space degrades the training efficiency of the RL agent, since a training sample only directly updates the probability of a single action, and exploring a large action space requires a huge amount of training samples. Moreover, the action space of a state depends on its graph structure, which changes at each step.

Solution. For a circuit C, Quarl decomposes its action space A_C into two "subspaces": a position space P_C and a transformation space X. P_C includes all gates in C, and X contains all transformations. (Recall that a subcircuit matching a transformation can be determined by matching a gate, see §2.) Under this decomposition, A_C is a subset of $P_C \times X$. Quarl uses separate policies for P_C and X.

3.2 Challenge 2: State Representation

At the core of most RL algorithms is a representation of states as high-dimensional vectors. However, unlike most RL tasks whose states have a uniform structure, a state in our setting is a quantum circuit whose size and topology depend on the input circuit and may change at each step during

the optimization process. Therefore, designing a uniform state representation for quantum circuits is challenging. A straightforward approach would be to directly represent each quantum circuit as a high-dimensional vector. However, due to the diversity of quantum circuits, this approach leads to learned representations that are highly tailored to the circuits used in training and do not generalize well to unseen circuits.

Solution. A key insight for addressing this challenge is leveraging the *locality* of circuit transformations, that is, while the overall optimization strategy depends on the entire circuit, we hypothesize that the decision of applying a transformation at a gate can be largely guided by the local environment of the gate. Based on that, we design a neural architecture that relies on local decision making when selecting a gate to apply a transformation. In particular, Quarl uses a K-layer graph neural network (GNN) to represent the K-hop neighborhood of each gate (see Definition 1). While each representation only encodes a local subcircuit, combining all gates' representations allows Quarl to collectively represent an entire circuit. A key advantage of our approach is that the representations generated by the GNN is independent of the circuit size and thus can generalize to circuits at different time steps in the optimization process. While our approach localizes the decision making, it still allows global circuit-wide guidance, since we fine-tune the RL agent (including the weights of the GNN) when optimizing a circuit (see Section 5). With this design, we aim to achieve a good balance between local and global decision making in the optimization process.

3.3 Quarl's Approach

Combining our solutions to the two challenges discussed above, we propose a hierarchical approach to optimizing quantum circuits using RL. Quarl's neural architecture is outlined in Figure 4. The first stage in processing the current circuit C_t is the *gate representation generator*, which is a graph neural network (GNN) that computes a learned vector representation for the K-hop neighborhood of each gate in C_t (see Section 4.1). Next, based on these learned representations, Quarl's *gate selector* chooses a gate g_t using a learned *gate-selecting policy*, denoted $\pi_g(\cdot|C_t;\theta_g)$, which is a probability distribution over all gates in C_t (see Section 4.2). Finally, the learned representation of g_t is fed into Quarl's *transformation selector*, which selects a transformation using a learned *transformation-selecting policy*, denoted $\pi_x(\cdot|C_t, g_t;\theta_x)$, which is a probability distribution over all valid transformations at g_t (see Section 4.3).

The gate- and transformation-selecting policies are trained jointly in Quarl with a combined actor-critic architecture. Specifically, the actor network learns the transformation-selecting policy, and the critic network acts both as a value estimator for the transformation-selecting policy and as a predictor for the gate-selecting policy. In other words, the transformation-selecting policy is the *only* policy whose parameters are updated by gradient ascent and whose actions (i.e., transformation selections) are reinforced. The critic network learns to estimate the value of the state derived by applying the transformation-selecting policy, which is the K-hop neighborhood of a gate. The gate-selecting policy evaluates all gates with the value estimator of the transformation-selecting policy and selects (with high probability) a high-value gate. Intuitively, the value estimator is suitable to form the gate-selecting policy, because high value indicates high optimization opportunity.

Compared to a straightforward PPO approach, where the actor learns both π_g and π_x, and the critic learns to estimate the value of an entire circuit, our method has fix-sized state (the K-hop neighborhood) for both the actor and critic, and a fixed and relatively small action space for the transformation-selecting policy. These advantages ultimately make our policies easier to train. However, our specialized architecture requires a different advantage estimator from standard PPO, which we develop in Section 5.

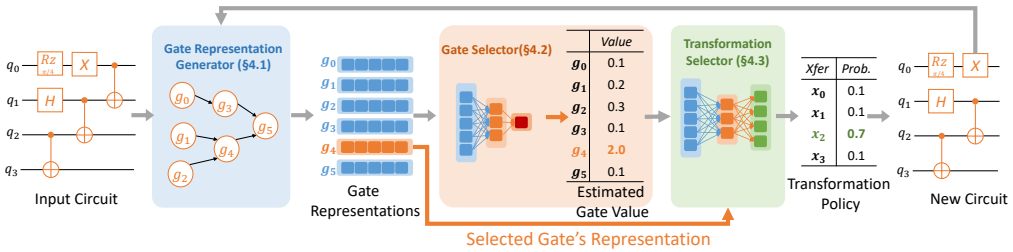


Fig. 4. The neural architecture of Quarl's RL agent. The arrows indicate the control flow.

4 QUARL'S NEURAL ARCHITECTURE

4.1 Gate Representation Generator

For an input circuit C, Quarl uses a graph neural network (GNN) with K layers to learn to represent the *K-hop neighborhood* of each gate in C as a high-dimensional vector. Our GNN architecture follows GraphSAGE (Hamilton et al., 2017).

Definition 1 (k-hop neighborhood). *For a node v in graph G, its* k-hop neighborhood, *denoted $G^{(k)}(v)$, is the subgraph of G that includes all nodes within k (undirected) hops from v.*

Figure 3b shows the k-hop neighborhood for a selected gate (identified by $G^{(0)}(v)$) and different values of $k \in \{1, 2, 3\}$.

The GNN architecture takes as inputs (1) a circuit represented as a graph, (2) gate-level features (i.e., the type of each gate), and (3) edge-level features (i.e., the direction of each edge and the qubits it connects to). Let $h_g^{(k)}, k \in \{1, \ldots, K\}$ denote the representation of the k-hop neighborhood of gate g outputted by the kth GNN layer, let $h_g^{(0)}$ denote the input features of the first GNN layer for g (i.e., an embedding of g's gate type), and let e_{uv} denote the edge features of the edge between node u and v. $h_g^{(k)}$ is computed by taking $h_u^{(k-1)}$ and e_{ug} as inputs, where u is a neighbor gate of g. Each GNN layer includes an *aggregation* phase, which first gathers the representations of each gate's neighbors from the previous GNN layer, and an *update* phase, which computes a new representation for each gate by combining its previous representation and the aggregated neighborhood representations.

Aggregation phase. Let $N(g)$ denote gate g's neighbors in the graph (i.e. a set of gates that share an in- or out-edge with g). For each gate g, the aggregation phase at layer k takes as inputs $h_u^{(k-1)}$ and e_{ug} ($u \in N(g)$) and computes an aggregated representation of g's neighbors $a_g^{(k)}$ with a multi-layer perceptron (MLP) (Gardner and Dorling, 1998). The neural architecture of the aggregate phase is formalized as follows:

$$a_g^{(k)} = \sum_{u \in N(g)} \sigma\left(W_a^{(k)} \cdot \text{concat}(h_u^{(k-1)}, e_{ug}) + b_a^{(k)}\right) \tag{1}$$

where $W_a^{(k)}$ and $b_a^{(k)}$ denote the weights of the MLP in the aggregation phase of the k-th layer, and $\sigma(\cdot)$ is the ReLU function (Agarap, 2018).

Update phase. For each gate g, the update phase computes a new representation $h_g^{(k)}$ by combining g's representation from the previous GNN layer (i.e., $h_g^{(k-1)}$) and the aggregated neighbor representation (i.e., $a_g^{(k)}$). The neural architecture of the update phase is formalized as follows:

$$h_g^{(k)} = \sigma\left(W_u^{(k)} \cdot \text{concat}(h_g^{(k-1)}, a_g^{(k)})\right) \tag{2}$$

where $W_u^{(k)}$ denotes the weight matrix of the MLP in the update phase.

Note that using more GNN layers allows Quarl to represent a larger neighborhood of each gate but introduces more trainable parameters, which requires more time and resource to train (see Section 5). Section 6 analyzes the choice of K. In the text that follows, θ_{rg} represents the parameters used within the representation generator.

4.2 Gate Selector

The gate selector is composed of two parts: a *gate value predictor* and a *gate sampler*. The gate value predictor predicts the on-policy value $V^{\pi_x}(C, g)$ (see Section 5.3) of gate g on circuit C for the transformation selecting policy π_x. The gate value predictor takes as an input $h_g^{(K)}$, which represents the K-hop neighborhood of g on circuit C, and outputs $V^{\pi_x}(C, g; \theta_g, \theta_{rg})$, which approximates $V^{\pi_x}(C, g)$, and θ_g denotes the trainable parameters of the predictor. The predictor uses a multi-layer perceptron (MLP) (Hinton, 1987) in our current implementation.

Our gate selecting policy π_g is formed by applying a temperature softmax (He et al., 2018) to the outputs of the gate value predictor, which is parameterized as follows:

$$\pi_g(g|C; \theta_g, \theta_{rg}) = \frac{exp\big(V^{\pi_x}(C, g; \theta_g, \theta_{rg})/t\big)}{\sum_{g' \in C} exp\big(V^{\pi_x}(C, g'; \theta_g, \theta_{rg})/t\big)} \tag{3}$$

where $\pi_g(g|C; \theta_g, \theta_{rg})$ denotes the probability of choosing g in circuit C, and t is a temperature parameter for the softmax function. The temperature $t \in (0, +\infty)$ balances exploration and exploitation. Specifically, when t is larger, π_g selects gates with increased randomness and becomes more explorative. On the other hand, when t becomes smaller, π_g becomes more exploitative and tends to select the gate with the highest estimated value. Balancing exploration and exploitation across different circuits requires circuit-specific temperatures. For a specific circuit C, we set the temperature t as $t = 1/ln\frac{\lambda(|C|-1)}{1-\lambda}$, $\lambda \in (0, 1)$, where $|C|$ is the number of gates in circuit C and λ is a measure of exploitation. Specifically, we set t such that even if there is only one gate with a value closes to 1 (representing an optimization opportunity to reduce cost by 1), and the values of all other gates are close to 0 (representing no optimization opportunity), the gate selector samples the high-value gate with probability $\sim \lambda$.

4.3 Transformation Selector

Given a circuit C and a gate g, Quarl's *transformation selector* chooses a transformation to apply at g. The transformation selector is an MLP, which takes as an input the representation of the selected gate $h_g^{(K)}$, and outputs a probability distribution $\pi_x(\cdot|C, g; \theta_x, \theta_{rg})$ over the entire set of transformations X, where θ_x denotes the trainable parameters of the transformation selector. The final output of the MLP is followed by a masked softmax layer that filters out invalid transformations. The mask is generated by the circuit transformation engine by checking every transformation in X to figure out which of them can be applied to gate g.

5 TRAINING AND INFERENCE METHODOLOGY

To train Quarl's neural architecture with PPO, Section 5.1 introduces *hierarchical advantage estimator*, a novel approach to estimating the actions' advantages in our problem setting. Algorithm 1 lists Quarl's RL-based optimization algorithm, which optimizes a circuit by training the RL agent. This algorithm is used in both pre-training and fine-tuning of the optimizer. A training iteration of Quarl consists of two phases: *data collection*, which uses the current RL agent to generate trajectories (line 5-19), and *agent update*, which updates the agent's trainable parameters with gradient ascent (line 20-21). The two phases are introduced in Sections 5.2 and 5.3, respectively. Section 5.4 discusses Quarl's combination of pre-training and fine-tuning to optimize an input circuit.

Algorithm 1 Quarl's RL-based circuit optimization algorithm. B and T are hyper-parameters that specify the number of trajectories to collect in each training iteration and the maximum number of transformations in each trajectory. NOP is a special transformation that stops the current trajectory when selected. Quarl uses α to control data collection (see Section 5.2).

Inputs: A circuit C_{input}
Output: An optimized circuit
$C = \{C_{input}\}$ ▷ *C is the initial circuit buffer*
for *iteration = 1, …* ***do***
▷ *Training data collection*
  $\mathcal{R} = \emptyset$ ▷ *Clear the rollout buffer in each iteration*
  for $j = 1, \ldots, B$ ***do***
    Sample an initial circuit $C_0^{(j)}$ *from* C *(see Section 5.2)*
    for $t = 1, \ldots, T$ ***do***
      Compute gate representations $h_g^{(K)}$ *for* $g \in C_{t-1}^{(j)}$
      Selects gate $g_t^{(j)}$ *using* $\pi_g(\cdot|C_{t-1}^{(j)}; \theta_g, \theta_{rg})$
      Selects transformation $x_t^{(j)}$ *using* $\pi_x(\cdot|C_{t-1}^{(j)}, g_t^{(j)}; \theta_x, \theta_{rg})$
      Generate new circuit $C_t^{(j)}$ *by applying* $x_t^{(j)}$ *at* $g_t^{(j)}$
      Compute reward $r_t^{(j)} = \textsc{Cost}(C_{t-1}^{(j)}) - \textsc{Cost}(C_t^{(j)})$
      $\mathcal{R} = \mathcal{R} \cup \left\{\left(C_{t-1}^{(j)}, C_t^{(j)}, g_t^{(j)}, x_t^{(j)}, r_t^{(j)}, V^{\pi_x}(C_{t-1}^{(j)}, g_t^{(j)}; \theta_g, \theta_{rg}),\ \pi_x(x_t^{(j)}|C_{t-1}^{(j)}, g_t^{(j)}; \theta_x, \theta_{rg})\right)\right\}$
      if $\textsc{Cost}(C_t^{(j)}) \le \textsc{Cost}(C_0^{(j)})$ ***then***
        $C = C \cup \{C_t^{(j)}\}$
      if $x_t^{(j)} = \texttt{NOP} \vee \textsc{Cost}(C_t^{(j)}) > \alpha \cdot \textsc{Cost}(C_{input})$ ***then***
        break ▷ *End the trajectory*
▷ *Agent update*
  Update $\theta_g, \theta_x, \theta_{rg}$ *using SGD (loss given by eq. (7)) for M epochs*
return $\operatorname{argmin}_{C \in \mathcal{C}} \textsc{Cost}(C)$

5.1 Hierarchical Advantage Estimator

As described in Sections 3.3 and 4, Quarl uses a combined actor-critic architecture to jointly train two policies: the gate-selecting policy π_g, which is directly approximated using the values $V^{\pi_x}(C, g)$; and the transformation-selecting policy π_x, on which we apply RL training with an adaptation of PPO. For the transformation-selecting policy π_x, its input is the embedding of the K-hop neighborhood of a gate g on circuit C (i.e., $C^{(K)}(g)$), and its output action is an applicable transformation x at gate g. To update π_x using policy gradient, a key challenge is to estimate the advantage of applying transformation x. In the canonical framework of PPO, the advantage $\hat{A}_t$ of applying x at step t over a sampled trajectory τ is estimated by the difference between the return R_t and the value function, namely $\hat{A}_t = R_t - V^{\pi_x}(C, g)$. However, such a straightforward approach can be problematic in our setting. The value function $V^{\pi_x}(C, g)$ is computed using the GNN embedding over the K-hop neighborhood of gate g, so it represents the *local* value for the neighborhood of g rather than the *global* value of the entire circuit C. Note that when generating a trajectory τ during RL training, our hierarchical policy may choose an arbitrary gate g according to π_g to apply a transformation, so the following steps in τ after x is applied to g can involve gates that are arbitrarily far away from g. Accordingly, the trajectory return $R_t = \sum_{i=t}^{T} \gamma^{i-t} r(C_i, g_i, x_i)$ is in fact estimating the *global* return over the circuit C rather than the *local* return over the K-hop neighborhood. Therefore, the *multi-step* trajectory return R_t may not be the most appropriate choice for advantage estimation.

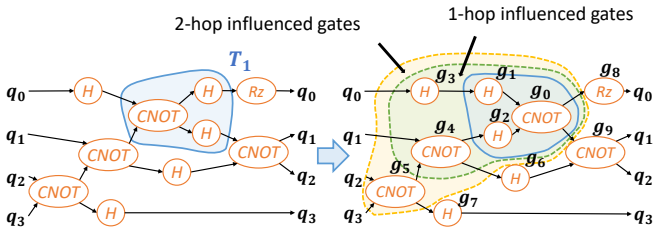


Fig. 5. The ℓ-hop influenced gates of applying the graph transformation in Figure 3c.

In order to obtain an accurate *local* advantage, we propose a *hierarchical advantage estimator*, which is based on a *1-step* return estimation given by

$$\hat{A}(C, g, x) = r(C, g, x) + \gamma \left(\max_{g' \in \mathrm{IG}(\ell, C, g, x)} V^{\pi_x}(C', g') \right) - V^{\pi_x}(C, g), \tag{4}$$

where $r(C, g, x)$ is the reward of applying transformation x to gate g of circuit C, C' denotes the new circuit obtained by applying the transformation, and $\mathrm{IG}(\ell, C, g, x)$ is the ℓ-hop influenced gates of this transformation (defined below) to ensure locality.

Definition 2 (ℓ-hop influenced gates). *For a transformation x applied at gate g of circuit C, its ℓ-hop influenced gates, denoted $IG(\ell, C, g, x)$, is gate set of the new circuit C' that includes (1) all the new gates introduced by the transformation, and (2) all ℓ-hop (directed) predecessors of these gates.*

$\mathrm{IG}(\ell, C, g, x)$ includes all gates in the new circuit C' influenced by transformation x, which can fall into two categories. First, all gates in the target graph of the transformation are in $\mathrm{IG}(\ell, C, g, x)$, since these gates are newly introduced by the transformation. For transformation T_1 in Figure 5, g_0, g_1, and g_2 are added to the new circuit by applying T_1. Second, $\mathrm{IG}(\ell, C, g, x)$ also includes gates whose applicable transformations may change due to the transformation. For this category, we consider all ℓ-hop predecessors of the new gates. Figure 5 shows the 1- and 2-hop influenced gates. Quarl locates a transformation based on a topologically minimal gate in the source graph of the transformation. Therefore for any gate not influenced by a transformation x, its applicable transformations remain the same after applying transformation x, as shown in Theorem 1.

Theorem 1 (Property of influenced gates). *Let C' be the new circuit obtained by applying transformation x to gate g on circuit C, and assume that for any transformation (G, G') the depth of G is at most d (the depth of a directed acyclic graph is the maximal length of a path in the graph). For any gate $g'.g' \in C' \wedge g' \notin IG(d, C, g, x)$, its set of applicable transformations is identical in C and C'.*

Proof. Let $NG(C, g, x)$ refer to the new gates introduced by x in C'. For gates $g' \in C' \wedge g' \notin \mathrm{IG}(d, C, g, x)$, there are three cases: (1) g' is a k-hop predecessor of $NG(C, g, x)$ where $k > d$; (2) g' is a successor of $NG(C, g, x)$; (3) g' is neither a predecessor nor a successor of $NG(C, g, x)$. The availability of any transformation (G, G') where the depth of G is less than or equal to d on gate g, only depends on the d-hop successors of g. For case (1), (2) and (3), the d-hop successors of g' remains unchanged. For case (1), since g' is a k-hop predecessor of $NG(C, g, x)$ where $k > d$, then all gates in $NG(C, g, x)$ are outside g''s d-hop successors, so g''s d-hop successors remain unchanged. For case (2), all successors of g' are successors of $NG(C, g, x)$, remaining unchanged. For case (3), since g' is not a predecessor of $NG(C, g, x)$, none of its successors are in $NG(C, g, x)$. Thus, g''s successors remain unchanged. □

Quarl uses influenced gates to capture dependencies between transformations when estimating their advantages. As per eq. (4), Quarl estimates the local advantage of a transformation based

on the maximum V^{π_x} (calculated by the gate value predictor) across all influenced gates of the transformation. The rationale behind the *max* aggregator is to propagate the reward signal through the dependency path of transformations, so our hierarchical advantage estimator can capture the rewards that truly depend on the current transformation. In our experiments, the maximal depth of circuits used in transformations is 4, but in practice we found that using 1-hop influenced gates leads to better results than using 4-hop influenced gates suggested by Theorem 1. We note that while *some* transformations have a depth of 4, most have a lower depth; including more predecessors introduces more variance into the training process as the value of the maximal-value node in the influenced gates may not always be related to the applied transformation.

5.2 Training Data Collection

Generating Trajectories. As shown in Algorithm 1 line 5-19, in each training iteration, Quarl generates B trajectories according to the current policy, and each trajectory has at most T steps. These generated trajectories are taken as the training dataset to update the gate- and transformation-selecting policies. To generate the j-th trajectory, Quarl randomly selects an initial circuit $C_0^{(j)}$ from the *initial circuit buffer* C (introduced later in this section), and iteratively applies transformations on the selected circuit. Specifically, at the t-th time step, Quarl takes the current circuit $C_{t-1}^{(j)}$ as an input and chooses a gate $g_t^{(j)}$ and a transformation $x_t^{(j)}$ using the gate- and transformation-selecting policies. After that, $x_t^{(j)}$ is applied to $g_t^{(j)}$ on $C_{t-1}^{(j)}$ to generate a new circuit $C_t^{(j)}$. At each time step, Quarl collects the following data: (1) the current and new circuits $C_{t-1}^{(j)}$ and $C_t^{(j)}$, (2) the selected action $(g_t^{(j)}, x_t^{(j)})$, (3) the reward r_t, (4) the value of $g_t^{(j)}$ given by the gate value predictor $V^{\pi_x}(C_{t-1}^{(j)}, g_t^{(j)}; \theta_g, \theta_{rg})$ (see Section 4.2), and (5) the probability of choosing $x_t^{(j)}$ under the current transformation-selecting policy (i.e., $\pi_x(x_t^{(j)} | C_{t-1}^{(j)}, g_t^{(j)}; \theta_x, \theta_{rg})$. The collected data is saved in a *rollout buffer* $\mathcal{R}$, which is initialized to empty at the start of each iteration. According to the PPO algorithm (Schulman et al., 2017), no gradient is generated during the generation of trajectories.

Stop conditions. After a trajectory begins, Quarl keeps applying transformations on the circuit until one of the following stop conditions is met. First, each trajectory can have at most T time steps, where T is a hyper-parameter. Second, Quarl introduces a special transformation named `NOP` into the transformation set. Selecting `NOP` as the transformation (Algorithm 1 line 18) indicates that Quarl chooses to end the current trajectory, which provides Quarl the ability to stop when it finds that moving forward cannot bring benefits or even leads to negative rewards. Third, Quarl stops the current trajectory when the cost of the current circuit is α times greater than the cost of the input circuit (Algorithm 1 line 18), which prevents Quarl from moving toward a wrong direction too far. We set α to be 1.2 in our evaluation.

Initial circuit buffer. Instead of always starting from the input circuit C_{input} in a trajectory, Quarl samples a circuit from an *initial circuit buffer* C (Algorithm 1 line 16-17). C includes all circuits discovered in previous trajectories whose cost is lower than the trajectory's initial circuit. To select a circuit from C, Quarl first samples a cost and then uniformly selects a circuit from the set of circuits with the sampled cost. Users can define customized probability distributions over costs depending on how they expect their agent to behave. Specifically, a greedier distribution where probabilities for sampling lower costs are larger makes the agent more progressive in doing optimization, preferring to extend lower-cost states.

Compared to always starting a trajectory from C_{input}, sampling circuits from the initial circuit buffer enhances the exploration of the search space. In particular, starting from C_{input} restricts Quarl to only explore circuits at most T steps away from C_{input}, where T is the maximum number

of steps in a trajectory. In contrast, starting from a randomly selected circuit allows Quarl to explore previously unknown areas, enabling Quarl to discover more optimized circuits.

5.3 Agent Update

To update Quarl's neural architecture, we first traverse the rollout buffer, and compute the advantage value for each time step according to eq. (4). Next, we train the graph embedding network, the gate value predictor network and the transformation selector network jointly with stochastic gradient descent for M epochs. Following the PPO algorithm in (Schulman et al., 2017), the loss function for the transformation selector can be rewritten as

$$L^{TS}(\theta_x, \theta_{rg}) = E_\tau \left[\sum_t \min(\rho_t(\theta_x, \theta_{rg})\hat{A}_t, \mathit{clip}(\rho_t(\theta_x, \theta_{rg}), \epsilon)\hat{A}_t) \right] \tag{5}$$

where $\rho_t(\theta_x, \theta_{rg}) = \frac{\pi_x(x_t|C_t, g_t; \theta_x, \theta_{rg})}{\pi_x(x_t|C_t, g_t; \theta_{x(old)}, \theta_{rg(old)})}$. Note that $\rho_t(\theta_x, \theta_{rg})$ doesn't depend on θ_g. Intuitively, with eq. (5), the transformation-selecting policy is updated toward making a better choice given C_t and g_t (i.e. increasing the probability of choosing transformation x_t if $\hat{A}_t > 0$, and decreasing the probability otherwise). Since the update is conditioned on fixed C_t and g_t, the gate-selecting policy θ_g isn't involved. To update the value estimator network, which is also the gate selecting policy, Quarl minimizes $L^{VE}(\theta_g, \theta_{rg})$ given by

$$L^{VE}(\theta_g, \theta_{rg}) = E_\tau \left[\sum_t \hat{A}(C_t, g_t, x_t)^2 \right] \tag{6}$$

To train the networks jointly, we combine these loss functions into

$$L(\theta) = L^{TS}(\theta_x, \theta_{rg}) + c_1 L^{VE}(\theta_g, \theta_{rg}) + c_2 H(\pi_x; \theta_x, \theta_{rg}) \tag{7}$$

where θ denotes all the trainable parameters combining θ_g, θ_x and θ_{rg}, $c_1 \geq 0$ and $c_2 \geq 0$ are two coefficients, and $H(\pi_x)$ denotes the entropy of policy π_x, which serves as a regularization term to promote exploration.

We now describe how Quarl updates the trainable parameters of the three networks. Let $\theta_{x(old)}$, $\theta_{g(old)}$ and $\theta_{rg(old)}$ denote the parameters in the gate selector, the transformation selector and the representation generator, used during data collection, respectively. In each epoch, for each data point in the rollout buffer (i.e. C_{t-1}, C_t, g_t, x_t, r_t,$V^{\pi_x}(C_{t-1}, g_t; \theta_{g(old)}, \theta_{rg(old)})$, and $\pi_x(x_t|C_{t-1}, g_t; \theta_{x(old)}, \theta_{rg(old)})$), we compute the loss function $L(\theta)$ w.r.t. the latest parameters $\theta_{x(new)}$, $\theta_{g(new)}$ and $\theta_{rg(new)}$. Quarl uses all three neural networks to compute the loss function. Specifically, in each epoch, to compute $L^{TS}(\theta_x, \theta_{rg})$ and $L^{VE}(\theta_g, \theta_{rg})$, the graph representation generator first generates the representation for the nodes. Second, the transformation selector network uses these representations to compute $\rho(\theta_{x(new)}, \theta_{rg(new)})$ in eq. (5) and $H(\pi_x; \theta_{x(new)}, \theta_{rg(new)})$ in eq. (7). Finally, the representations are also used by the gate value predictor network to compute the regression loss on gate value according to eq. (6). As a result, back-propagating the loss in eq. (7) generates gradients for all three networks. The gradients for the gate selector come from the loss function in eq. (6), the gradients for the transformation selector derive from the loss function in eq. (5) and the entropy term (i.e. $H(\pi_x; \theta_x, \theta_{rg})$ in eq. (7)), and the gradients for the gate representation generator come from all three terms in the loss function eq. (7).

5.4 Pre-training, Fine-tuning, and Policy-guided Search

Circuits implemented in the same gate set share common local topologies, which offers opportunities to transfer the learned optimizations from one circuit to another in the same gate set. This observation motivates our pre-training and fine-tuning approach through which we can

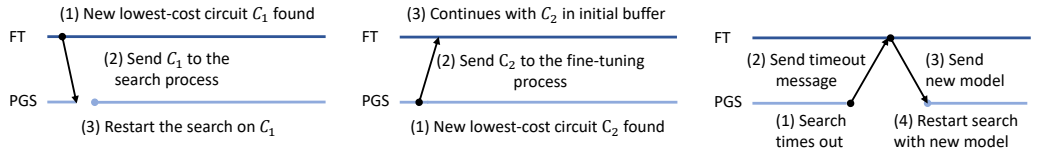
avoid training from scratch on previously unseen circuits and accelerate training. As introduced in Section 4, Quarl's neural architecture can automatically adapt to circuits with different sizes and/or topologies, which enables the pre-training and fine-tuning pipeline.

Pre-training. The goal of the pre-training phase is to help the agent learn how to optimize different local structures. Quarl uses a diversity of circuits for pre-training, which allows the agent to explore various local structures and prevents over-fitting to the structure of a single circuit. Minor changes are needed to support training Quarl on multiple circuits. In the initial buffer, circuits are clustered into equivalent groups. At the beginning of each trajectory, Quarl chooses the initial circuit of a trajectory by first uniformly sampling an equivalent group and then randomly selecting a circuit from the group. All trajectories are used in gradient estimations to update the agent.

Fine-tuning. Quarl optimizes a new circuit by fine-tuning the pre-trained model on the circuit, which allows Quarl to discover optimizations specific to the new circuit. During the fine-tuning, there is a single equivalent group in the initial buffer, which corresponds to the new circuit Quarl fine-tunes on. Optimized circuits may be discovered during the fine-tuning, however, the primary goal of fine-tuning is to make the agent fit to the new circuit, specifically, the model should learn the gate values and policy distribution specific to the new circuit. Once the model has fitted to the new circuit, we should stop the fine-tuning to avoid wasting computation and use the fine-tuned model to apply a more efficient *policy-guided search* to finish the rest of the optimization.

Policy-guided search. In this stage, we employ a circuit optimization model that has been fitted to the circuit at hand. We use this model to guide the further optimization of the circuit using a technique we call *policy-guided search*, which shares some similarities with the trajectory collection stage during training. However, there are a few key differences between policy-guided search and trajectory collection. First, policy-guided search only maintains circuits with lowest cost in the initial circuit buffer, ensuring that Quarl uses one of the best discovered circuits to start a trajectory. Second, after selecting a gate using the gate-selecting policy, instead of sampling a transformation from the transformation-selecting policy, Quarl selects the transformation with highest probability. During the search, if the transformation-selecting policy triggers a stop condition (described in Section 5.2) when selecting a transformation for a gate g on circuit C, Quarl applies a *hard mask* to g, which will prevent Quarl from revisiting g. Furthermore, to prevent Quarl from exploiting the policy without exploration, we also apply *soft masks* to gates that have been visited by Quarl for at least once. The difference between hard and soft masks is that once all gates in a circuit have been masked out, either by hard or soft masks, we remove the soft masks and do not reapply them. The purpose of soft masks is to ensure that every gate in a circuit C is visited at least once if the circuit is visited at least $|C|$ times. Whenever a circuit with lower cost is discovered, Quarl clears the initial circuit buffer, adds the circuit to the initial circuit buffer, and restarts the search.

Optimizing a circuit. To optimize an input circuit, Quarl runs a fine-tuning process and a policy-guided search process simultaneously. Figure 6 shows the interactions between the two processes, which exchange information whenever one process discovers a circuit with new lowest cost. Specifically, if a circuit C_1 with new lowest cost is found during fine-tuning, Quarl restarts the search using C_1, as shown in Figure 13a. Conversely, if a circuit C_2 with new lowest cost is found during the search, the fine-tuning process will include C_2 in its initial circuit buffer and continue fine-tuning, as shown in Figure 13b. Moreover, a timeout is set for the search process in case that its model is obsolete. If no new lowest cost circuit is discovered before timeout, Quarl restarts the policy-guided search using the most recent model from fine-tuning, as shown in Figure 6c.



(a) Fine-tuning discovers a circuit with new lowest cost.

(b) Policy-guided search finds a circuit with new lowest cost.

(c) No processes find a better circuit before search times out.

Fig. 6. Interactions between Quarl's fine-tuning (FT) and policy-guided search (PGS) processes. (a) and (b) show how the two processes exchange information when a circuit with new lowest cost is discovered. (c) shows the case where the policy-guided search times out.

5.5 Scaling to Large Circuits

This section analyzes the scalability of Quarl during the training data collection and agent update phases, and describe how we improve its scalability for large circuits.

Scalability of training data collection. To optimize an input circuit C, Quarl's gate value predictor computes a value for every gate in C at each step in a trajectory. This process consists of a GNN inference to generate the representation of all gates in C, and an MLP inference that calculates the values of C's gates. For each gate, both the GNN and MLP inferences take constant time, therefore the overall time complexity of the gate selector is $O(|C|)$. After a gate g is selected, selecting a transformation and identifying the l-hop influenced gates takes constant time. Overall, the time complexity of Quarl's data collection is $O(|C|)$.

However, as the size of the circuit grows, a significant amount of computation is discarded since all but one gate's representation are not used after gate selection. To address this inefficiency, Quarl partitions an input circuit into sub-circuits by limiting the number of gates in each sub-circuit, and optimize these sub-circuits individually. Specifically, to partition a circuit, Quarl first arranges the circuit's gates in topological order, then divide them into sequential partitions where each contains no more than a constant number of gates. This approach reduces the cost of training data collection to a constant. We evaluate the performance improvement of circuit partitioning in Section 6.6. However, partitioning circuits may lead to missed optimization opportunities that span across partitions, which we plan to explore in future work.

Scalability of agent update. The agent update phase involves recomputing the probability of choosing the same transformation in the data collection phase. Specifically, to calculate the importance sampling ratio $\rho_t(\theta_x, \theta_{rg})$ in the loss function (Equation (5)), we need the probability $\pi_x(x_t|C_t, g_t; \theta_x, \theta_{rg})$ under the updated θ_x and θ_{rg}. Since θ_{rg} is changed due to the update to the parameters of the GNN , Quarl needs to re-calculate the representation for g_t, which takes $O(|C_t|)$. Also, the network update phase involves gradient computation and backward-propagation, whose peak GPU memory usage is $O(|C_t|)$. On GPUs with limited device memory, we have to hand-tune the training batch-size to prevent out-of-memory issues.

Again, only a small number of gates are involved in the gradient generation, while the representation of other gates are compute-to-discard. We deal with this issue based on the fact that since the gate representation only contains information of its k-hop neighborhood, we can obtain exactly the same representation by running GNN only on its k-hop neighborhood. The number of gates in the k-hop neighborhood of a gate has a constant upper bound due to the sparse structure of quantum circuits. With this optimization, the complexity of computing gradient on a single data point becomes a constant.

Note that due to the design of Quarl's neural architecture, the time and space complexity of Quarl only depend the number of gates in a circuit, rather than the number of qubits.

6 EVALUATION

6.1 Experimental Setup

Benchmarks. To evaluate the effectiveness of our framework, we employ a widely-adopted quantum circuit benchmark suite, which has been previously used by several works (Amy et al., 2014, Hietala et al., 2021, Kissinger and van de Wetering, 2020, Nam et al., 2018, Xu et al., 2022c,a,b) for logical quantum circuit optimization. The benchmark suite primarily comprises arithmetic and reversible circuits, and we evaluate them in the Nam gate set (CX, Rz, H, X), following existing studies. Additionally, we transpile these benchmark circuits to the IBM gate set $(\{CX, Rz, X, SX\})$ to demonstrate Quarl's compatibility with different gate sets. Since this benchmark suite contains a limited number of circuit types, we extend our evaluation to include circuits from the MQTBench library (Quetschlich et al., 2022), which encompasses circuits from various categories, such as QAOA (Farhi et al., 2014) and VQE (Peruzzo et al., 2014).

Metrics. We use four metrics in our evaluation: total gate count, CNOT count, circuit depth, and fidelity. Operations on NISQ devices are affected by noise, and the error rates of different gates vary. Specifically, the error rate of two-qubit gates (e.g. CNOT) are typically an order of magnitude larger than single-qubit gates (e.g., 3×10^{-2} and 4.43×10^{-3}, respectively, on IBM Q20 (Li et al., 2018)). In light of this, we evaluate Quarl with not only total gate count but also CNOT count. Moreover, since qubits in NISQ devices have limited coherence time, circuit depth should be within a certain range for successful execution. The optimizing results ultimately translate to the fidelity of executions. However, the end-to-end fidelity involves many factors, such as mapping and routing method, device coupling map and device error rate, which are out of the scope of logical quantum circuit optimization. To rule out the effect of these factors, we follow prior work (Xu et al., 2022c) and report the absolute circuit fidelity, which is defined as the product of the success rate of all gates in the circuit. For a circuit C, its fidelity can be expressed as $\prod_{g \in C}(1 - e(g))$ where $e(g)$ denotes the empirical device error rate for gate g. During evaluation, we use the calibration data of IBM Washington device (IBM, 2023) where CNOT error rate is 1.214×10^{-2}, Rz error rate is 0, and the error rate of other single-qubit gates (i.e. X gate and SX gate) are 2.77×10^{-4}.

Server specification. Our experiments are conducted on the Perlmutter supercomputer (per, 2023). Pre-training is performed on a node equipped with an AMD EPYC 7763 64-core 128-thread processor, 256GB DRAM, and four NVIDIA A100-SXM4-40GB GPUs. For circuit optimization, we use a node with the same hardware specification, but with only one NVIDIA A100-SXM4-40GB GPU.

For each circuit, we allocate a 6-hour time budget for all tools including Nam, VOQC, Qiskit, Tket, Quartz, QUESO, and Quarl in our comparison. Circuit partitioning, as introduced in section 5.5 is only enabled in section 6.6 where we evaluate Quarl's scalability and is not enabled in other comparisons. We also evaluate Quarl against baselines using the same monetary budget, with findings presented in §6.7. We run all Quarl experiments three times with different seeds, and report the mean and standard deviation (std) of the results.

6.2 Implementation Details

We build the Quarl training pipeline on top of PyTorch (Paszke et al., 2019) and DGL (Wang et al., 2019a). We utilize the APIs provided by Quartz (Xu et al., 2022a) to generate graph representations of circuits and transformation rules and perform graph pattern matching. These APIs are encapsulated with Cython (Wang et al., 2019b) to facilitate their invocations by Python processes.

Correctness guarantee. Quarl leverages the verification techniques from Quartz to guarantee the correctness of the optimized circuits. Specifically, all circuit transformations considered by Quarl are first symbolically verified by Quartz's circuit equivalence verifier. In addition, we used

multiple circuit equivalence checkers, including Qiskit's (Aleksandrowicz et al., 2019) statevector equivalence checker and QCEC (Burgholzer and Wille, 2020), to examine the correctness of Quarl's final circuits. Except for some timeouts encountered during the test of very large circuits (i.e., those with many thousands of gates), all the output circuits have passed the equivalence checks.

Data-parallel pre-training. Quarl uses a distributed data-parallel approach during pre-training, leveraging multiple GPUs. Each process occupies a single GPU and contains an agent that independently collects trajectories and computes gradients. After each iteration, gradients are synchronized across GPUs to update model parameters. Each process also keeps its circuit buffer and shares information on the lowest-cost circuits found. In the evaluation, for each gate set, we pre-train Quarl's neural architecture on six circuits (i.e. `barenco_tof_3`, `gf2^4_mult`, `mod5_4`, `mod_mult_55`, `tof_5`, and `vbe_adder_3`) for 8 hours with 4 GPUs. We choose these circuits because they are relatively small and have diverse structures. Since the pre-trained models are reused in circuit optimization, we do not count the pre-training time when reporting Quarl's fine-tuning time.

Table 1. Hyperparameters in Quarl's training.

Hyperparameter	Pre-training	Fine-tuning
Horizon (T)	600	600
Actor learning rate	3e-4	3e-4
Critic learning rate	5e-4	5e-4
GNN learning rate	3e-4	3e-4
Num. actors	128	64
Num. epochs	20	5
Minibatch size	4800	4800
Discount (γ)	0.95	0.95
Clipping parameter ϵ	0.2	0.2
Entropy coeff.	0.02	0.02
GNN layers	6	6
GNN hidden dim.	128	128
GS MLP layers	2	2
GS hidden dim.	128	128
TS MLP layers	2	2
TS hidden dim.	256	256

Fine-tuning and policy-guided search. Within the 6-hour timeframe, we initiate both a fine-tuning process and a policy-guided search, with the fine-tuning's onset marking the beginning of this period. A 5-minute delay before starting the search process serves as a warm-up for the policy it employs. After this initial delay, both processes proceed concurrently for the remainder of the 6-hour window. The search process periodically updates itself with the newest policy and resets if neither process improves the circuit within a 20-minute search timeout. Consequently, throughout the 6-hour window, the search process may experience several timeouts and restarts.

Rotation merging. Aside from circuit transformations, many quantum circuit optimizers (i.e. Nam (Nam et al., 2018), VOQC (Hietala et al., 2021), Quartz (Xu et al., 2022a) and QUESO (Xu et al., 2022c)), incorporate *rotation merging*, a technique that merges R_z gates with identical phase polynomial expressions (Nam et al., 2018), into their optimization pipeline. However, the R_z gates being merged may be arbitrarily far apart, making it difficult to represent rotation merging as combinations of local circuit transformations. Rotation merging is adopted as a pre-processing pass of Quarl, yet we show that Quarl can learn similar behavior by itself.

Cost function. In all our experiments, we use total gate count as the cost function for Quarl. While 2-qubit gate count (e.g., CNOT count) can also be used as the cost function, we find that both of them yield equally good results in 2-qubit gate count. We do not adopt fidelity as the cost function because, at the logical optimization layer, we lack hardware specification, and the circuit is not mapped or routed on a physical device.

Hyperparameters. Table 1 lists the architectural details of the models in Quarl and hyperparameters used in Quarl's learning process. Beside that, we choose $\lambda = 0.9$ in Quarl's gate selector and use 1-hop influenced gates in training. The hyperparameters are chosen by grid search.

Table 2. Comparing Quarl and existing circuit optimizers on reducing total gate count of the benchmark circuits for the Nam gate set. The best result for each circuit is in bold.

Total Gate Count (Nam gate set)											
Circuit	**Orig.**	**Nam**	**VOQC**	**Qiskit**	**Tket**	**Quartz w/ R.M.**	**Quartz w/o R.M.**	**QUESO w/ R.M.**	**QUESO w/o R.M.**	**Quarl w/ R.M.**	**Quarl w/o R.M.**
tof_3	45	35	35	43	39	35	35	35	35	**33 (± 0)**	**33 (± 0)**
barenco_tof_3	58	40	40	54	49	40	46	38	38	**35 (± 0.6)**	**35 (± 0)**
mod5_4	63	51	51	61	57	37	39	25	27	**24 (± 0)**	**24 (± 0)**
tof_4	75	55	55	71	64	55	55	55	55	**51 (± 0)**	**51 (± 0)**
tof_5	105	75	75	99	89	75	75	**68**	75	70 (± 1.2)	71 (± 0)
barenco_tof_4	114	72	72	106	97	72	90	75	68	**64 (± 2.0)**	66 (± 3.5)
mod_mult_55	119	91	92	115	103	90	94	93	100	**84 (± 0)**	85 (± 1.7)
vbe_adder_3	150	89	89	189	130	89	112	79	81	**72 (± 1.2)**	**72 (± 1.2)**
barenco_tof_5	170	104	104	158	145	104	134	98	98	**92 (± 2.0)**	94 (± 2.0)
csla_mux_3	170	155	158	249	149	148	146	148	148	**141 (± 0.0)**	**141 (± 0.6)**
rc_adder_6	200	140	152	226	172	152	170	152	176	**136 (± 1.5)**	151 (± 1.2)
gf2^4_mult	225	187	186	212	205	176	215	176	183	**162 (± 0.6)**	163 (± 2.5)
tof_10	255	175	175	239	214	175	175	175	175	**170 (± 4.2)**	174 (± 1.2)
hwb6	259	-	209	258	217	214	250	202	211	194 (± 1.0)	**192 (± 1.2)**
mod_red_21	278	180	184	256	223	198	268	215	204	**179 (± 0.0)**	182 (± 4.2)
gf2^5_mult	347	296	287	327	319	274	334	273	287	**258 (± 1.7)**	267 (± 3.1)
csum_mux_9	420	266	280	420	365	256	420	252	364	**230 (± 9.5)**	283 (± 6.1)
qcla_com_7	443	284	269	498	377	288	431	**248**	320	257 (± 3.1)	290 (± 27.6)
ham15-low	443	-	348	421	392	360	435	**274**	346	338 (± 21.1)	335 (± 7.8)
barenco_tof_10	450	264	264	418	385	264	450	338	248	**235 (± 1.2)**	270 (± 19.1)
gf2^6_mult	495	403	401	464	453	391	485	381	435	**357 (± 2.5)**	386 (± 0.6)
qcla_adder_10	521	399	416	630	444	404	518	397	464	**381 (± 1.2)**	383 (± 2.0)
gf2^7_mult	669	555	543	627	614	531	657	576	594	**490 (± 5.3)**	557 (± 29.0)
gf2^8_mult	883	712	706	819	806	703	883	680	789	**659 (± 12.1)**	757 (± 19.5)
qcla_mod_7	884	**624**	678	933	759	652	884	645	775	629 (± 2.1)	690 (± 21.5)
adder_8	900	606	596	1001	802	624	874	579	618	**577 (± 13.2)**	598 (± 31.7)
Geo. Mean Reduction	-	28.0%	27.0%	-0.6%	13.1%	28.3%	12.9%	31.0%	26.1%	**35.2 (± 0.2)%**	32.4 (± 0.3)%

6.3 Comparison on the Nam Gate Set

We compare Quarl with existing rule-based [1] (i.e. Nam (Nam et al., 2018), VOQC (Hietala et al., 2021), Qiskit (Aleksandrowicz et al., 2019) and t|ket⟩ (Sivarajah et al., 2020)) and search-based (i.e. Quartz (Xu et al., 2022a) and QUESO (Xu et al., 2022c)) optimizers on total gate count, CNOT count, and circuit depth on the Nam gate set. We do not evaluate on circuit fidelity since the Nam gate set is not hardware native. Both Quarl and Quartz (Xu et al., 2022a) use a set of 6206 transformation rules generated by Quartz. This section (and Section 6.4) focus on circuits with less than 1,500 gates, and Section 6.6 evaluates the performance and scalability of Quarl on larger circuits. Nam et al. (2018) and VOQC (Hietala et al., 2021) have both incorporated rotation merging into their framework while Quartz and QUESO adopt it as a preprocessing procedure. Similarly, Quarl adopts rotation merging as a preprocessing step. However, to demonstrate the effectiveness of Quarl without rotation merging, we report results of Quarl, Quartz and QUESO without rotation merging.

As shown in Table 2, Quarl outperforms existing rule-based circuit optimizers on almost all benchmark circuits. Rule-based optimizers rely on a fixed set of manually designed rules and schedule them in a predetermined, typically greedy, manner. For instance, Nam (Nam et al., 2018) is specifically fine-tuned for the Nam gate set and is among the best-performing rule-based optimizers on that gate set. Nam applies gate-set-specific heuristics and rules, such as rotation merging and floating R_z gates, and also uses 1- and 2-qubit gate cancellation as subroutines to simplify the circuits. To apply these subroutines, Nam et al. (2018) uses two fixed schedules. We report the results of the *heavy* schedule, which is more aggressive and achieves better results. Other rule-based

[1] PyZX (Kissinger and van de Wetering, 2020) is another rule-based optimizer. However, it only minimizes the T gate count and does not explicitly optimize our chosen metrics, namely total gate count, CNOT count, circuit fidelity, and depth. We observe that PyZX achieves worse-than-original performance on these metrics and therefore exclude it in this comparison.

Table 3. Comparing Quarl and existing circuit optimizers on reducing CNOT count of the benchmark circuits for the Nam gate set. The best result for each circuit is in bold.

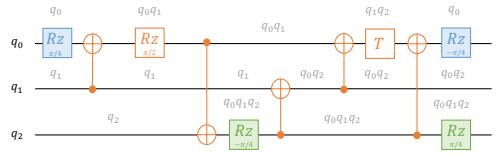
CNOT Count (Nam gate set)											
Circuit	**Orig.**	**Nam**	**VOQC**	**Qiskit**	**Tket**	**Quartz w/ R.M.**	**Quartz w/o R.M.**	**QUESO w/ R.M.**	**QUESO w/o R.M.**	**Quarl w/ R.M.**	**Quarl w/o R.M.**
tof_3	18	14	14	18	18	14	14	14	14	**12 (± 0.0)**	**12 (± 0.0)**
barenco_tof_3	24	18	18	24	24	18	20	16	16	**13 (± 0.6)**	**13 (± 0.0)**
mod5_4	28	28	28	28	28	20	22	14	16	**13 (± 0.0)**	**13 (± 0.0)**
tof_4	30	22	22	30	30	22	22	22	22	**18 (± 0.0)**	**18 (± 0.0)**
tof_5	42	30	30	42	42	30	30	30	30	**25 (± 1.0)**	26 (± 0.0)
barenco_tof_4	48	48	34	48	48	34	40	30	30	26 (± 1.2)	**24 (± 3.5)**
mod_mult_55	48	40	42	48	48	39	41	39	42	37 (± 0.0)	**35 (± 1.7)**
vbe_adder_3	70	50	50	62	62	44	50	38	39	**33 (± 1.0)**	34 (± 1.2)
barenco_tof_5	72	50	50	72	72	50	60	44	44	**38 (± 2.0)**	42 (± 2.0)
csla_mux_3	80	70	74	71	71	70	68	70	70	**63 (± 0.0)**	**63 (± 0.6)**
rc_adder_6	93	71	71	81	81	71	77	71	79	**63 (± 1.5)**	69 (± 1.2)
gf2^4_mult	99	99	99	99	99	95	99	95	96	81 (± 0.6)	**79 (± 1.7)**
tof_10	102	70	70	102	102	70	70	70	70	**65 (± 4.0)**	70 (± 1.2)
hwb6	116	-	104	115	111	99	115	101	99	87 (± 0.6)	**86 (± 1.7)**
mod_red_21	105	81	81	105	104	81	105	81	81	**74 (± 0.0)**	78 (± 2.3)
gf2^5_mult	154	154	154	154	154	149	154	148	154	**133 (± 1.5)**	134 (± 1.0)
csum_mux_9	168	140	140	168	168	140	168	**112**	168	118 (± 8.6)	138 (± 3.1)
qcla_com_7	186	132	132	174	174	127	178	123	139	**116 (± 2.1)**	134 (± 9.1)
ham15-low	236	-	210	236	225	209	236	198	202	**193 (± 13.7)**	**193 (± 3.5)**
barenco_tof_10	192	130	130	192	192	130	192	114	114	**101 (± 0.6)**	134 (± 9.9)
gf2^6_mult	221	221	221	221	221	221	221	211	221	**187 (± 2.5)**	196 (± 1.0)
qcla_adder_10	233	183	199	213	205	187	230	180	208	**166 (± 1.0)**	167 (± 2.0)
gf2^7_mult	300	300	300	300	300	300	300	285	300	**259 (± 4.2)**	282 (± 11.5)
gf2^8_mult	405	405	405	405	402	405	405	382	403	**375 (± 11.6)**	383 (± 7.8)
qcla_mod_7	382	292	328	366	366	307	382	300	356	**286 (± 0.6)**	292 (± 13.9)
adder_8	409	291	301	385	383	305	395	276	295	**275 (± 4.2)**	297 (± 17.2)
Geo. Mean Reduction	-	18.4%	17.8%	2.5%	3.0%	20.6%	10.9%	25.4%	21.1%	**32.5 (± 0.4%)**	30.0 (± 0.2)%

optimizers (i.e., VOQC (Hietala et al., 2021), Qiskit (Aleksandrowicz et al., 2019), t|ket⟩ (Sivarajah et al., 2020)) operate similarly. VOQC (Hietala et al., 2021) also implements specific optimizations for the Nam gate set, which are combined into a pass `optimize_nam`. In contrast, Quarl is equipped with transformation rules automatically generated by Quartz (Xu et al., 2022a).

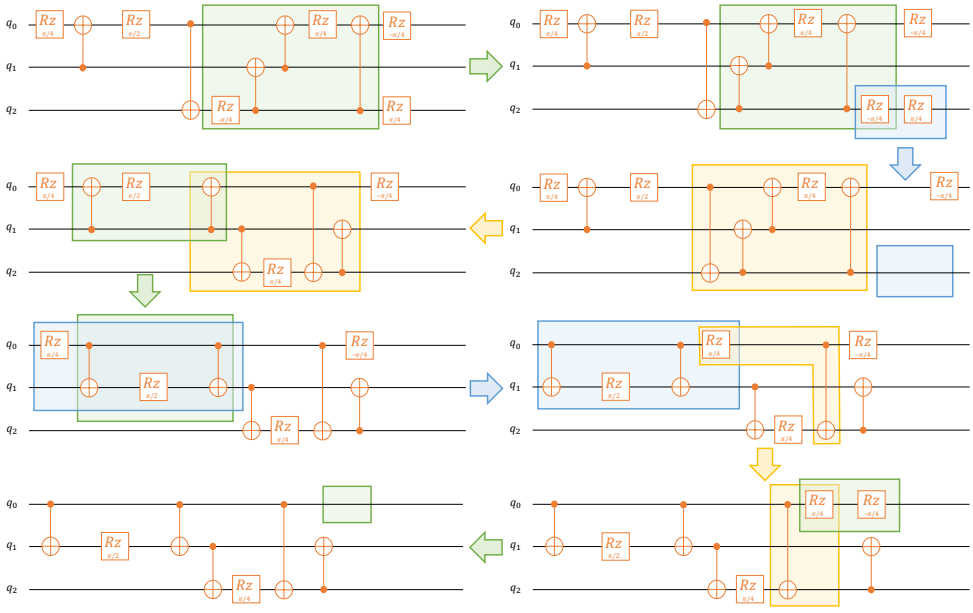
For search-based approaches, Quarl surpasses Quartz (Xu et al., 2022a) and QUESO (Xu et al., 2022c) for most of the benchmark circuits. Both Quartz and Quarl employ the same set of transformations, and therefore the difference in their performance arises from their respective search algorithms. Quartz uses cost-based backtracking search to schedule transformations, while Quarl leverages reinforcement learning to detect optimization opportunities and determine the best combination of transformations to achieve them. QUESO uses a more complex set of transformation rules than Quartz, and employs a beam-search-based optimizer. The reduction in gate count achieved by Quarl translates to an average reduction of 25.7% in circuit depth, which outperforms the best depth reduction achieved by existing optimizers (i.e., 19.6%).

Table 3 shows CNOT count reduction. Quarl reduces CNOT count by 33.9% on average, while the reduction achieves by existing optimizers is at most 25.4%. Reducing CNOT count is a desirable advantage since it significantly increases the fidelity of executing a circuit on NISQ devices with sparse inter-qubit connections.

Even for relatively small circuits, Quarl achieves better performance than Quartz (Xu et al., 2022a) and QUESO (Xu et al., 2022c), as shown in Table 2 and Table 3. This is because exhaustively exploring the search space is prohibitively expensive even for very small circuits. And both Quartz and QUESO greedily prune the search space, resulting in missed optimizations that require cost-increasing transformations. In contrast, Quarl's learning-based approach provides a more efficient and scalable solution, which allows Quarl to better explore the search space and discover more optimization opportunities.



(a) Rotation merging



(b) Rotation merging as transformations

Fig. 7. (a) shows an example circuit with *CNOT* and *Rz* gates. The phase of each *Rz* gate is shown above the gate. Rotations with identical phase can be combined by rotation merging. (b) represents the rotation merging optimization as a sequence of transformations discovered by Quarl. Note that Quarl can also update the final circuit to have the same CNOT topology as the original circuit by applying a few additional transformations, which are omitted in the figure.

As shown in Table 2 and Table 3, the performance gap between Quarl with and without rotation merging is relatively small (i.e., 3.2% in both cases); as a comparison, disabling rotation merging decreases Quartz's performance by 15.4% for total gate count and 9.7% for CNOT count. Though rotation merging can be achieved by multiple applications of local R_z transformation rules, it is challenging to rebuild it with those rules since applying them does not provide immediate rewards until we finally fuse the R_z gates. RL algorithms are efficient in realizing long-term goals. Our experiments show that Quarl can learn to perform optimizations similar to rotation merging through its own exploration. Though it takes some time to learn, once the best schedule is learned, Quarl can apply it everywhere. An example where Quarl merges two pairs of R_z gates with identical phase polynomial using a sequence of local transformations is shown in Figure 7.

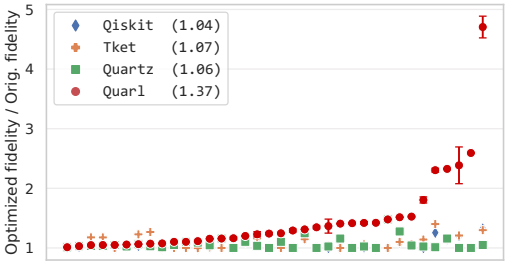


Fig. 8. Fidelity comparison on the IBM gate set. The numbers in parentheses in the legend indicate the average relative fidelity improvement.

Table 4. Comparing Quarl and existing circuit optimizers on reducing total gate count and CNOT count of the benchmark circuits for the IBM gate set. The best result for each circuit is in bold.

Circuit	Orig.		Qiskit		Tket		Quartz		Quarl	
	# gates	# CXs	# gates	# CXs	# gates	# CXs	# gates	# CXs	# gates	# CXs
tof_3	53	18	51	18	51	18	42	**14**	**39 (± 0)**	**14 (± 0)**
barenco_tof_3	65	24	61	24	61	24	52	20	**42 (± 0)**	**16 (± 0)**
mod5_4	71	28	69	28	69	28	62	27	**50 (± 0)**	**22 (± 0)**
tof_4	88	30	84	30	84	30	67	**22**	**62 (± 0)**	**22 (± 0)**
tof_5	123	42	117	42	117	42	92	**30**	**85 (± 0)**	**30 (± 0)**
barenco_tof_4	125	48	117	48	117	48	99	40	**75 (± 0)**	**30 (± 0)**
mod_mult_55	140	48	135	48	145	48	114	41	**101 (± 1.0)**	**39 (± 0.6)**
vbe_adder_3	165	70	186	62	159	62	124	50	**82 (± 0)**	**36 (± 0)**
barenco_tof_5	185	72	173	72	173	72	146	60	**109 (± 1.0)**	**39 (± 0.0)**
csla_mux_3	200	80	235	71	188	71	181	72	**39 (± 0.0)**	**39 (± 0.0)**
rc_adder_6	229	93	250	81	277	81	189	75	**164 (± 0.0)**	**71 (± 0.0)**
gf2^4_mult	246	99	232	99	232	99	229	99	**195 (± 2.0)**	**87 (± 1.5)**
hwb6	287	116	281	115	282	111	272	114	**212 (± 1.5)**	**87 (± 1.2)**
mod_red_21	294	105	279	105	317	104	292	105	**204 (± 2.3)**	**77 (± 0.6)**
tof_10	298	102	282	102	282	102	217	**70**	**201 (± 1.7)**	**70 (± 0.0)**
gf2^5_mult	374	154	353	154	353	154	372	154	**298 (± 1.0)**	**136 (± 1.2)**
csum_mux_9	459	168	453	168	474	168	459	168	**321 (± 0.5)**	**140 (± 0.6)**
barenco_tof_10	485	192	453	192	453	192	482	192	**273 (± 0.6)**	**114 (± 0.0)**
ham15-low	485	236	456	236	455	225	480	234	**362 (± 3.0)**	**188 (± 2.3)**
qcla_com_7	486	186	511	174	486	174	470	174	**304 (± 3.2)**	**117 (± 0.0)**
gf2^6_mult	528	221	496	221	496	221	528	221	**424 (± 3.8)**	**200 (± 1.7)**
qcla_adder_10	587	233	636	213	556	205	586	232	**414 (± 3.1)**	**165 (± 1.2)**
gf2^7_mult	708	300	665	300	665	300	708	300	**576 (± 1.2)**	**276 (± 1.5)**
gf2^8_mult	928	405	864	405	861	402	923	403	**782 (± 16.1)**	**380 (± 7.2)**
qcla_mod_7	982	382	980	366	933	366	978	382	**762 (± 24.1)**	**311 (± 11.0)**
adder_8	1004	409	1010	385	1169	383	997	405	**687 (± 6.4)**	**282 (± 3.2)**
vqe_8	199	14	91	**14**	93	**14**	92	**14**	**85 (± 2.5)**	**14 (± 0.0)**
qgan_8	256	28	114	28	98	28	100	26	**84 (± 4.9)**	**18 (± 0.0)**
qaoa_8	284	32	211	**32**	159	**32**	**157**	**32**	**157 (± 1.7)**	**32 (± 0.0)**
ae_8	502	56	350	56	**283**	56	373	56	289 (±4.6)	**55 (± 0.0)**
qpeexact_8	555	64	373	64	**286**	**55**	371	64	327 (±5.6)	64 (±0.6)
qpeinexact_8	571	65	381	65	**312**	**56**	381	65	339 (±4.6)	65 (±0.6)
qft_8	578	68	392	68	**262**	**56**	326	67	310 (±4.9)	67 (±0.0)
qftentangled_8	647	75	415	75	**295**	**61**	489	75	326(±6.1)	74 (±0.0)
portfoliovqe_8	708	84	288	84	232	84	359	84	**203 (± 4.4)**	**54 (± 0.6)**
portfolioqaoa_8	1352	168	975	168	**712**	168	1207	168	811 (±12.3)	**160 (± 3.2)**
Geo. Mean Reduction	-	-	14.4%	1.8%	20.1%	4.1%	19.9%	7.7%	**36.7 (± 0.1)%**	**21.3 (± 0.1)%**

6.4 Comparison on the IBM Gate Set

We conduct a comparison between Quarl and existing optimizers[2] on the IBM gate set on total gate count, CNOT count, circuit depth, and fidelty. To optimize a circuit, both Quarl and Quartz (Xu et al., 2022a) use a set of 6881 transformation rules generated by Quartz. For parametric quantum gates, Quartz only discovers symbolic transformations applicable to these gates with arbitrary parameter values, and misses transformations that are only valid for specific parameter values. Some of these transformations missed by Quartz are important to optimize circuits on the IBM gate set. To address this limitation, we also include three single-qubit transformations missed by Quartz: (1) $Rz(\pi) = SX\ Rz(\pi)\ SX$, (2) $SX\ Rz(\pi/2)\ SX = Rz(\pi/2)\ SX\ Rz(\pi/2)$, and (3) $SX\ Rz(3\pi/2)\ SX = Rz(3\pi/2)\ SX\ Rz(3\pi/2)$. We verify their correctness by directly computing the concrete matrix representation of the two circuits in each transformation. These three transformations are used by both Quarl and Quartz. Since rotation merging is not native to the IBM gate set, it is not applied during the evaluation.

As shown in Table 4, Quarl greatly outperform existing optimizers for the IBM gate set for both total gate count and CNOT count. Specifically, Quarl reduces total gate count by 36.7% on average, while existing optimizers reduce total gate count by at most 20.1%. Quarl also reduce the CNOT

[2]QUESO (Xu et al., 2022c) is not included in the comparison because they doesn't support the new IBM gate set.

count of the circuits by 21.3%, while existing approaches achieve up to 7.7% CNOT count reduction. The performance gap between Quarl and Quartz (Xu et al., 2022a) is widened on the IBM gate set. We hypothesize that this is because of Quartz's reliance on rotation merging as a preprocessing pass, which is not applicable on the IBM gate set. Quarl achieves 21.1% circuit depth reduction on average, whereas existing optimizers reduce the circuit depth by at most 13.5%.

Figure 8 shows the fidelity improvement achieved by different optimizers. Quarl improves circuit fidelity by up to 4.84× (on `adder_8`) and 1.37 (± 0.007) times on average, while Qiskit, Tket and Quartz improve circuit fidelity by 1.04×, 1.07×, 1.06× on average, respectively. Quarl achieves the best fidelity improvements for most circuits. This is largely because CNOT involves a much higher error rate than other gates, and Quarl performs the best on CNOT reduction.

6.5 Ablation Studies

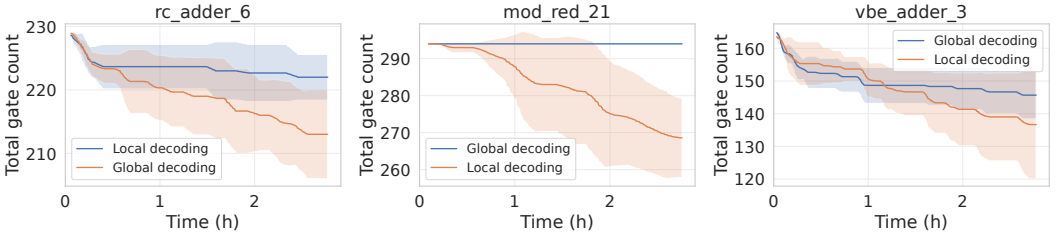


Fig. 9. Comparison on Quarl's local decoding of actions with global decoding.

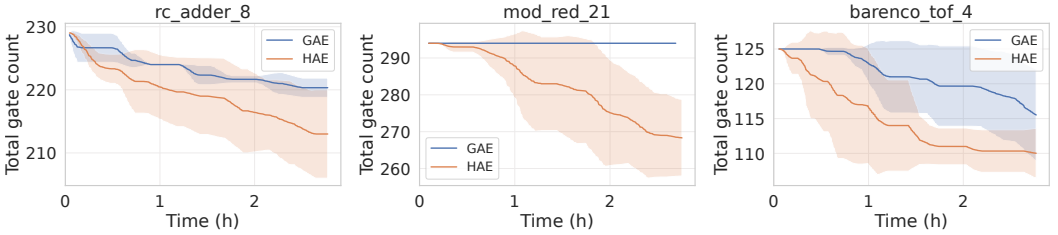


Fig. 10. Comparison on Quarl's hierarchical advantage estimation (HAE) with GAE.

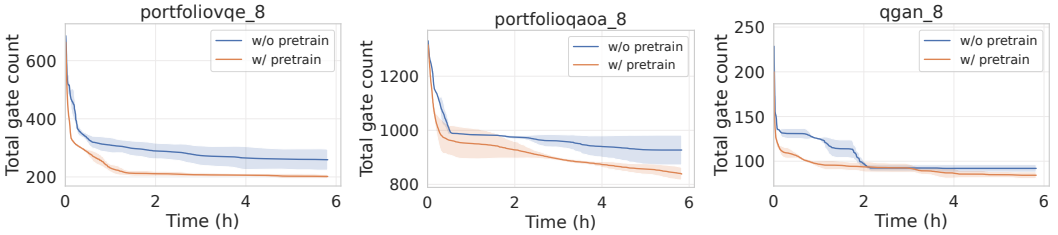


Fig. 11. Comparison on Quarl's optimization with and without pretraining.

Local decoding of actions. We conduct an ablation study on Quarl's local decoding of actions. To conduct a comparison, the baseline uses *global* decoding of actions where both the gate/transformation selectors and the value network use a global representation of an entire circuit to make predictions. The global representation is obtained by a max pooling over all the node representations. As shown in Figure 9, the global decoding approach achieves relatively marginal optimizations compared with Quarl. This is because it is very difficult for the networks to take the entire graph as an input and learn to select one action among millions of candidates. In contrast, Quarl first selects a promising sub-graph and decodes actions locally by leveraging locality of quantum circuit optimization.

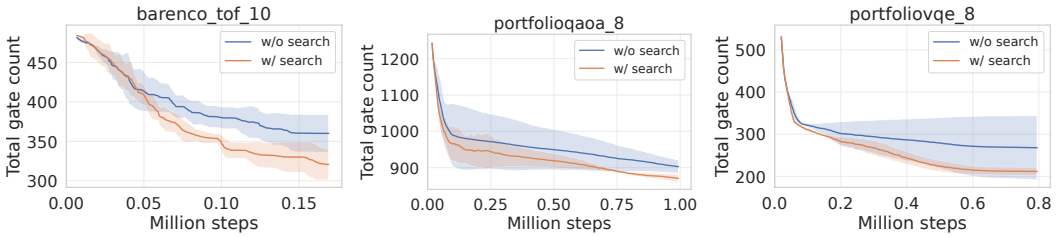
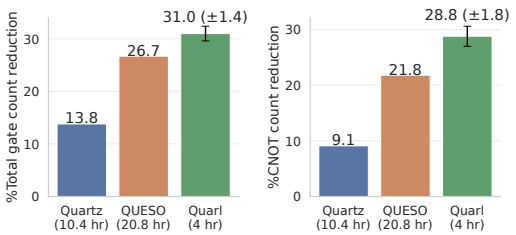


Fig. 12. Comparison on Quarl's optimization with and without policy-guided search.

Hierarchical advantage estimation (HAE). We further evaluate the effect of HAE compared with generalized advantage estimation (GAE). As shown in Figure 10, Quarl outperforms the agent equipped with GAE [3]. Though GAE is different from estimating the advantage with the return R_t, they are similar in that they both reflects global advantage, while Quarl's HAE outputs the *local* advantage which is more appropriate to use in this scenario because we want the advantage to reflect the *local* influence of an action.

Pre-training. We assess the generalizability of Quarl's pre-trained neural architecture to unfamiliar circuits and its influence on performance. We contrast the performance of fine-tuning processes (with search) using the pre-trained model against those initialized with random parameters. Results in Figure 10, derived from the MQT benchmark circuits (distinct from pre-training circuits), reveal that pre-training enables Quarl to quickly identify optimizations across all circuits. This yields superior outcomes, demonstrating the model's efficacy on larger, previously unencountered circuits.

Policy-guided search. To evaluate Quarl's policy-guided search, we start two groups of experiments from the same pre-trained model checkpoint: the first group performs fine-tuning with policy-guided search while the second only conducts fine-tuning. As shown in Figure 12, policy-guided search allows Quarl to discover better solution faster. Compared to fine-tuning, Quarl's policy-guided search allows greedier exploitation (i.e., storing only best cost circuits in the initial buffer and using argmax instead of sampling during transfer selection) and bolder exploration (i.e. using soft mask to force exploration).



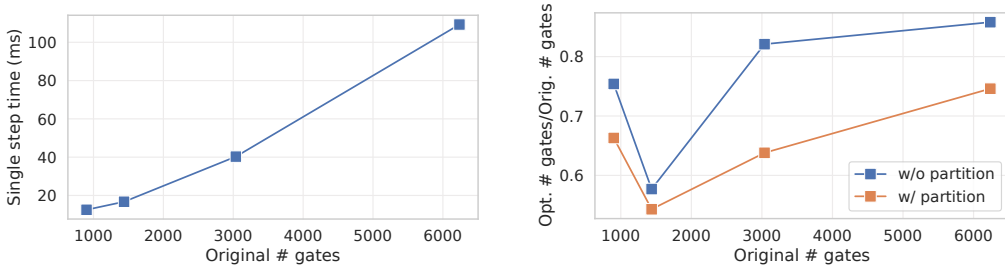
(a) Total gate count. (b) CNOT count.

Fig. 13. Comparison between Quarl, Quartz and QUESO with same amount of computational resources.

6.6 Scalability Analysis

This section investigates the relationship between circuit size and time to collect training data, and evaluates how circuit partitioning improves Quarl's scalability. We evaluate Quarl using four circuits with similar structures, namely `adder_8`, `adder_16`, `adder_32`, and `adder_64`, which have 900, 1437, 3037, and 6237 gates, respectively. Figure 14a shows the time to collect training data for these circuits, demonstrating a linear relationship, which aligns with our analysis in Section 5.5. Additionally, Figure 14b compares the performance of Quarl on the original and partitioned circuits, where we partition the input circuit into subcircuits with at most 512 gates based on a topological

[3]Note that the results on circuit `mod_red_21` are the same in fig. 9 and fig. 10, this is because they are compared to same Quarl baseline and neither the global-decoding method nor GAE find any optimizations.



(a) Relation between gate count and single-step time cost in data collection.

(b) Comparison between Quarl's performance with and without circuit partitioning.

Fig. 14. Scalability analysis.

order. The experiments are conducted with a search time budget of 2 hours to rule out the influence of optimization saturation. The results indicate that circuit partitioning generally increases Quarl's performance in limited search time, but it may lead to performance degradation due to missed optimization opportunities across partitions.

6.7 Discussion on Computational Cost

We also compare the computational requirement of Quarl with other tools, focusing on monetary cost as a unifying metric since direct computational power comparison between CPUs and GPUs is challenging. While the performance of rule-based optimizers does not vary with computational resources, we contrast Quarl against existing search-based optimizers, including Quartz (Xu et al., 2022a) and QUESO (Xu et al., 2022c), both of which run on CPUs. The major monetary cost of Quarl comes from the use of an A100 GPU. The lowest market price of an on-demand instance with a NVIDIA A100-SXM4-40GB GPU is $1.10/hr (Lambda, 2023). For Quartz and QUESO, the computational cost is mainly contributed by the memory usage. As claimed in their papers, running Quartz and QUESO requires 32GB and 16GB of memory, respectively, while the market price of an on-demand instance with 32GB and 16GB of memory are at least $0.384/hr and $0.192/hr, respectively (AWS, 2023). For a comparable budget, Quarl runs for 4 hours, whereas Quartz and QUESO operate for 10.4 and 20.8 hours, respectively. As shown in Figure 13, Quarl outperforms both Quartz and QUESO under the same monetary budget. Running Quartz and QUESO for 10 and 20 hours slightly improves the final performance compared with the 6-hour experiments (shown in table 2 and table 3), which stems from the greedy search algorithms used in these approaches. In contrast, Quarl strikes a better balance that favors both time-efficiency and superior performance.

7 RELATED WORK

Quantum circuit optimization. In recent years, several optimizing compilers for quantum circuits have been introduced, such as Qiskit (Aleksandrowicz et al., 2019), t|ket⟩ (Sivarajah et al., 2020), Quilc (Skilbeck et al., 2020), and voqc (Hietala et al., 2021). These optimizers rely on a rule-based strategy that applies manually designed circuit transformations whenever possible. Quartz (Xu et al., 2022a) uses a search-based approach that generates a comprehensive set of circuit transformations and applies them using a backtracking search algorithm. QUESO (Xu et al., 2022c), on the other hand, uses a path-sum-based approach to synthesize non-local circuit transformations and apply them using beam search. However, the large search space of functionally equivalent circuits and the need for cost-increasing transformations pose challenges for existing approaches to effectively discover circuit optimizations. Quarl addresses this challenge with a novel neural architecture and RL-training procedure, enabling it to automatically identify optimization opportunities and

apply the right transformations through interaction with the environment. Different from the aforementioned compilers that apply transformations, PyZX (Kissinger and van de Wetering, 2020) represents circuits as ZX-diagrams and use ZX-calculus (Hadzihasanovic et al., 2018, Jeandel et al., 2018) to simplify ZX-diagrams, which are eventually converted back to the circuit representation. In contrast, Quarl is designed to learn to perform transformations. Applying Quarl's techniques to optimize circuits in ZX-calculus is a promising avenue for future research.

Reinforcement learning for quantum computing. Prior work has explored the use of reinforcement learning for quantum circuit optimization. For example, Fösel et al. (2021) directly applied the PPO algorithm to this problem, focusing on a limited set of transformations (mainly gate cancellation). In contrast, Quarl tackles additional challenges arising from a more diverse and complex set of transformations. Quarl's solution involves a hierarchical action space, leveraging the locality of circuit transformations, and the introduction of hierarchical advantage estimations. As another example, Ostaszewski et al. (2021) emphasizes the optimization of state preparation circuits for the VQE algorithm. Instead of a transformation-centric approach, Ostaszewski et al. (2021) optimizes circuits by reconstructing them from ground up. Quarl, when compared with this work, seeks to optimize a broader range of quantum circuits and presents a fundamentally different approach.

Reinforcement learning has also been applied to the qubit routing problem, which involves inserting `SWAP` gates into the circuits to enable their execution on near-term intermediate-scale quantum (NISQ) devices (Li et al., 2018). Examples include QRoute (Sinha et al., 2022) and a DQN-based method proposed by Pozzi et al. (2020), both aiming to minimize the depth of circuits after routing with RL. It is worth noting that Quarl's techniques are orthogonal to RL-based qubit router, and combining them may lead to further improvements in circuit optimization on NISQ devices.

8 CONCLUSION

In this paper, we present Quarl, a learning-based quantum circuit optimizer. In Quarl's hierarchical approach, circuits are optimized with a sequence of transformation applications where in each application a gate-selecting policy and a transformation-selecting policy are used to choose a gate and a transformation to apply on the selected gate, respectively. The two policies are trained jointly, with a single actor-critic architecture. To apply PPO in the training of this architecture, we propose hierarchical advantage estimation (HAE) as an novel advantage estimator. Experiment results show that Quarl significantly outperforming existing circuit optimizers.

ACKNOWLEDGEMENT

We thank Shinjae Yoo and Mingkuan Xu for their helpful feedback. This work is supported by NSF awards CNS-2147909, CNS-2211882, and CNS-2239351, and research awards from Amazon, Cisco, Google, Meta, Oracle, Qualcomm, and Samsung. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC award DDR-ERCAP0023403.

DATA-AVAILABILITY STATEMENT

We have deposited a research artifact associated with our study (Li et al., 2024), which includes both code and scripts necessary for replicating the experimental results reported in our paper. Please note that due to non-deterministic factors related to multi-processing, there may be minor discrepancies between the replicated results and those originally published. Despite these potential variations, the provided materials offer a robust framework for accurately understanding and reproducing our experimental methodology.

REFERENCES

2023. The Perlmutter Supercomputer. https://docs.nersc.gov/systems/perlmutter/.

Abien Fred Agarap. 2018. Deep Learning using Rectified Linear Units (ReLU). https://doi.org/10.48550/ARXIV.1803.08375

Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. 2019. *Qiskit: An Open-source Framework for Quantum Computing*. https://doi.org/10.5281/zenodo.2562111

Matthew Amy, Dmitri Maslov, and Michele Mosca. 2014. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 10 (2014), 1476–1489. https://doi.org/10.1109/TCAD.2014.2341953

AWS. 2023. Amazon EC2 M6i Instances. https://aws.amazon.com/ec2/instance-types/m6i/ Accessed: 19th Oct 2023.

Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. 2017. Quantum machine learning. *Nature* 549, 7671 (2017), 195–202.

Lukas Burgholzer and Robert Wille. 2020. Advanced equivalence checking for quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 9 (2020), 1810–1824.

Yudong Cao, Jonathan Romero, Jonathan P Olson, Matthias Degroote, Peter D Johnson, Mária Kieferová, Ian D Kivlichan, Tim Menke, Borja Peropadre, Nicolas PD Sawaya, et al. 2019. Quantum chemistry in the age of quantum computing. *Chemical reviews* 119, 19 (2019), 10856–10915.

Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028* (2014).

Thomas Fösel, Murphy Yuezhen Niu, Florian Marquardt, and Li Li. 2021. Quantum circuit optimization with deep reinforcement learning. *arXiv preprint arXiv:2103.07585* (2021).

Matt W Gardner and SR Dorling. 1998. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment* 32, 14-15 (1998), 2627–2636.

Amar Hadzihasanovic, Kang Feng Ng, and Quanlong Wang. 2018. Two Complete Axiomatisations of Pure-State Qubit Quantum Computing. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 502–511. https://doi.org/10.1145/3209108.3209128

Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

Yu-Lin He, Xiao-Liang Zhang, Wei Ao, and Joshua Zhexue Huang. 2018. Determining the optimal temperature parameter for Softmax function in reinforcement learning. *Applied Soft Computing* 70 (2018), 80–85.

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434318

Geoffrey E Hinton. 1987. Learning translation invariant recognition in a massively parallel networks. In *International Conference on Parallel Architectures and Languages Europe*. Springer, 1–13.

IBM. 2023. *The IBM Washington quantum device*. https://reversiblebenchmarks.github.io/

Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. 2018. A Complete Axiomatisation of the ZX-Calculus for Clifford+T Quantum Mechanics. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 559–568. https://doi.org/10.1145/3209108.3209131

Aleks Kissinger and John van de Wetering. 2020. PyZX: Large Scale Automated Diagrammatic Reasoning. *Electronic Proceedings in Theoretical Computer Science* 318 (may 2020), 229–241. https://doi.org/10.4204/eptcs.318.14

Jason R. Koenig, Oded Padon, and Alex Aiken. 2021. Adaptive restarts for stochastic synthesis. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 696–709. https://doi.org/10.1145/3453483.3454071

Lambda. 2023. On-demand GPU cloud pricing. https://lambdalabs.com/service/gpu-cloud Accessed: 19th Oct 2023.

Gushu Li, Yufei Ding, and Yuan Xie. 2018. Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. https://doi.org/10.48550/ARXIV.1809.02573

Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. 2024. *Artifact for OOPSLA 2024 Paper: Quarl: A learning- based quantum circuit optimizer*. https://doi.org/10.5281/zenodo.10463907

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1928–1937.

Thomas Monz, Daniel Nigg, Esteban A Martinez, Matthias F Brandl, Philipp Schindler, Richard Rines, Shannon X Wang, Isaac L Chuang, and Rainer Blatt. 2016. Realization of a scalable Shor algorithm. *Science* 351, 6277 (2016), 1068–1070.

Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information* 4, 1 (2018), 1–12.

Mateusz Ostaszewski, Lea M Trenkwalder, Wojciech Masarczyk, Eleanor Scerri, and Vedran Dunjko. 2021. Reinforcement learning for optimization of variational quantum circuit architectures. *Advances in Neural Information Processing Systems* 34 (2021), 18182–18194.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature communications* 5, 1 (2014), 4213.

Jessica Pointing, Oded Padon, Zhihao Jia, Henry Ma, Auguste Hirth, Jens Palsberg, and Alex Aiken. 2021. Quanto: Optimizing Quantum Circuits with Automatic Generation of Circuit Identities. *arXiv:2111.11387* (2021). https://doi.org/10.48550/arXiv.2111.11387

Matteo G Pozzi, Steven J Herbert, Akash Sengupta, and Robert D Mullins. 2020. Using reinforcement learning to perform qubit routing in quantum compilers. *arXiv preprint arXiv:2007.15957* (2020).

Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2022. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. arXiv:2204.13719 MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

Animesh Sinha, Utkarsh Azad, and Harjinder Singh. 2022. Qubit routing using graph neural network aided Monte Carlo tree search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 9935–9943.

Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Science and Technology* 6, 1 (Nov 2020), 014003. https://doi.org/10.1088/2058-9565/ab8e92

Mark Skilbeck, Eric Peterson, appleby, Erik Davis, Peter Karalekas, Juan M. Bello-Rivas, Daniel Kochmanski, Zach Beane, Robert Smith, Andrew Shi, Cole Scott, Adam Paszke, Eric Hulburd, Matthew Young, Aaron S. Jackson, BHAVISHYA, M. Sohaib Alam, Wilfredo Velázquez-Rodríguez, c. b. osborn, fengdlm, and jmackeyrigetti. 2020. *rigetti/quilc: v1.21.0*. https://doi.org/10.5281/zenodo.3967926

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems* 12 (1999).

Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019a. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019b. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. 2022c. Synthesizing Quantum-Circuit Optimizers. *arXiv preprint arXiv:2211.09691* (2022).

Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. 2022a. Quartz: Superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA*. ACM. https://doi.org/10.1145/3519939.3523433

Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. 2022b. Quartz: Superoptimization of quantum circuits (extended version). arXiv:2204.09033 [cs.PL]