Optimizing ZX-Diagrams with Deep Reinforcement Learning

Maximilian Nägele[1,2] and Florian Marquardt[1,2]

[1]Max Planck Institute for the Science of Light, Staudtstraße 2, 91058 Erlangen, Germany
[2]Physics Department, Friedrich-Alexander-Universität Erlangen-Nürnberg, 91058 Erlangen, Germany

ZX-diagrams are a powerful graphical language for the description of quantum processes with applications in fundamental quantum mechanics, quantum circuit optimization, tensor network simulation, and many more. The utility of ZX-diagrams relies on a set of local transformation rules that can be applied to them without changing the underlying quantum process they describe. These rules can be exploited to optimize the structure of ZX-diagrams for a range of applications. However, finding an optimal sequence of transformation rules is generally an open problem. In this work, we bring together ZX-diagrams with reinforcement learning, a machine learning technique designed to discover an optimal sequence of actions in a decision-making problem and show that a trained reinforcement learning agent can significantly outperform other optimization techniques like a greedy strategy or simulated annealing. The use of graph neural networks to encode the policy of the agent enables generalization to diagrams much bigger than seen during the training phase.

1 Introduction

ZX-calculus is a diagrammatic language for the representation of quantum processes as graphs equipped with a set of local transformation rules. Due to the utility of these transformation rules, ZX-calculus has been applied to a wide range of problems ranging from fundamental quantum mechanics [1] over the description of measurement-based quantum computing [2] and analyzing variational quantum circuits [3] to quantum error correction [4, 5]. In particular, ZX-calculus has proven a promising candidate for speeding up tensor network simulations [6] and quantum circuit optimization [7–11]. However, finding the optimal sequence of transformation rules to achieve a given task is often a non-trivial task. Therefore, we bring together ZX-diagrams with reinforcement learning (RL), a machine learning technique where an agent iteratively interacts with an environment to learn a policy predicting an optimal sequence of actions. RL has been successfully applied to various domains such as game-playing [12, 13], robotics [14, 15], quantum chemistry [16, 17], and problems in quantum computing like quantum error correction [18–20], quantum control [21, 22], and circuit optimization [23, 24]. To capitalize on the graph structure of ZX-diagrams, we encode the policy of our reinforcement learning agent as a graph neural network (GNN) [25]. We train the agent to reduce the node number of random ZX-diagrams and show that the agent learns a non-trivial strategy outperforming a greedy strategy and simulated annealing. Moreover, the agent's policy generalizes well to diagrams much larger than seen during training. Our work lays the foundation for applying the combination of RL and ZX-calculus to a broad range of tasks like minimizing the gate count of quantum circuits or speeding up tensor network simulations by changing the optimization goal of the agent in future work.

Maximilian Nägele: maximilian.naegele@mpl.mpg.de

2 ZX-diagrams

A ZX-diagram is a graph representation of a quantum process defined by an arbitrary complex matrix of size $2^k \times 2^j$, where j is the number of ingoing and k the number of outgoing edges of the diagram. For example, we can represent the following matrix either as a quantum circuit consisting of single qubit X- and Z-gates and a

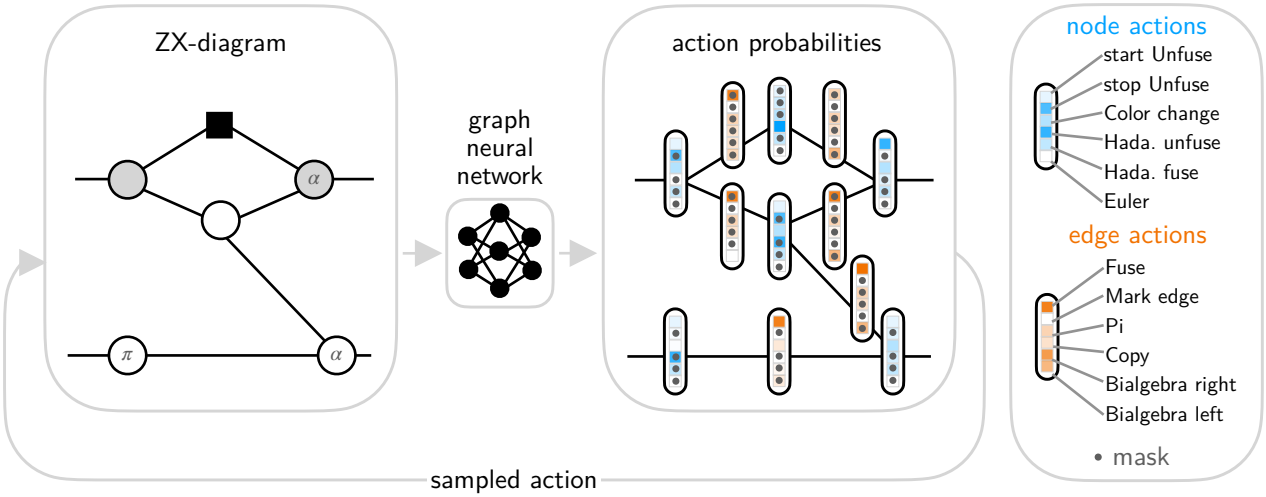

Figure 1: Schematic of the optimization loop. At each step, the reinforcement learning agent is provided with a ZX-diagram in the form of a graph. The agent then uses a graph neural network to suggest action probabilities of local graph transformations (color-coded), which act on either a unique edge (orange) or node (blue). Finally, an action is sampled from this probability distribution and applied to the diagram. In total, there are 6 separate actions per node and edge, some of which are not allowed in their local environment and, therefore, masked (grey dots). For a definition of the graph transformations see Figure 2.

CNOT gate or as a ZX-diagram according to

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} \leftrightarrow \quad \leftrightarrow \quad . \tag{1}$$

The central building blocks of ZX-diagrams are Z-spiders (white) and X-spiders (grey) defined as

$$= |\underbrace{0\ldots0}_{m}\rangle\langle\underbrace{0\ldots0}_{n}| + \mathrm{e}^{\mathrm{i}\alpha}|\underbrace{1\ldots1}_{m}\rangle\langle\underbrace{1\ldots1}_{n}|$$

$$= |\underbrace{+\ldots+}_{m}\rangle\langle\underbrace{+\ldots+}_{n}| + \mathrm{e}^{\mathrm{i}\alpha}|\underbrace{-\ldots-}_{m}\rangle\langle\underbrace{-\ldots-}_{n}|, \tag{2}$$

where $|1/0\rangle$ ($|+/-\rangle$) are the eigenvectors of the Pauli-Z (Pauli-X) matrix, α is an angle, and n and m are non-negative integers specifying the amount of input and output edges of the spider. While multiple different ZX-diagrams can describe the same underlying matrix, they can be transformed into each other by the set of local transformation rules depicted in Figure 2, which are correct up to a non-zero scalar factor [26]. These rules also imply that multiple edges connecting spiders of the same color can be reduced to just one edge and multiple edges between spiders of differing colors can be taken modulo two. Therefore, and due to the inherent symmetries of the Z- and X-spiders, ZX-diagrams can be regarded as simple graphs [27].

3 Optimization of ZX-diagrams as a reinforcement learning problem

Reinforcement learning (RL) is a machine learning technique where an agent recursively interacts with an environment during a trajectory comprising multiple steps. At each step t, the agent uses its policy to select an action (in our case a graph transformation) based on an observation describing the environment's state (in our case a ZX-diagram) as depicted in Figure 1. This action then modifies the state of the environment and a new observation and a numerical value, the reward r_t (in our case the difference in node number between the old and new diagram), is supplied to the agent. This scheme continues until the environment terminates the trajectory after a fixed amount of steps or the agent chooses a special *Stop* action. The agent is trained by repeating two phases: During the sampling phase the agent interacts with the environment for a fixed amount of steps. Then, during the training phase, the agent's policy is updated to maximize the expected cumulative reward over a complete trajectory $\langle\sum_t \gamma^t r_t\rangle$, where γ is the discount factor [28]. To enable the use of graph neural networks to encode the agent's policy, we use a custom implementation of a state-of-the-art reinforcement learning algorithm named Proximal Policy Optimization (PPO) [29] to train the agent (for details

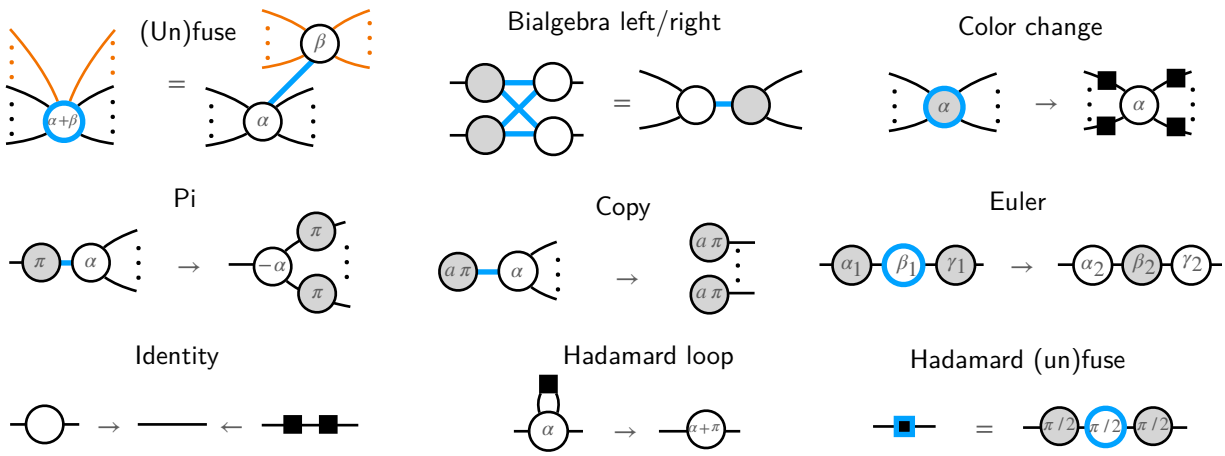

Figure 2: Encoding of the local transformation rules of ZX-diagrams as actions of a reinforcement learning agent. Blue colors indicate the encoding as an action of the agent acting on either an edge or a node. Some transformations are implemented in both directions as separate actions of the reinforcement learning agent (equal signs), while some are only implemented in one direction (arrows). Three dots stand for zero or more edges. Each rule also holds with the spiders' colors inverted and in both directions. Black squares represent a Hadamard gate as defined by the *Hadamard fuse* transformation. During the *Unfuse* transformation, a spider is split into two by arbitrarily splitting up its angle between the two resulting spiders, connecting them with a new edge, and transferring a subset of the originally connected edges (orange) to the new spider. In the *Copy* transformation, $a \in 0, 1$. In the *Euler* transformation, $\alpha_1/\beta_1/\gamma_1$ are related to $\alpha_2/\beta_2/\gamma_2$ by trigonometric functions as defined in [26].

on the algorithm and ablation studies of its features see Appendix B).

Each of the transformation rules of ZX-diagrams acts only on the local neighborhood of an edge or node. We can, therefore, identify each possible action of the RL agent with either a unique node or a unique edge as indicated by the blue lines in Figure 2. The agent's policy then predicts the unnormalized log-likelihood for each of the possible actions. By normalizing over the whole diagram, we build a probability distribution from which we sample an action that is applied to the diagram (see Figure 1).

Some of the transformations are symmetric and only implemented in one direction (arrows) resulting in one action. For example, the *Color change* transformation changes the color of a spider by inserting Hadamards on all connected edges. Because of the *Identity* transformation, an implementation in the other direction would be redundant. Other transformations need to be implemented in both directions (equal signs). For example, the *Hadamard (un)fuse* transformation either fuses three spiders into a single Hadamard or splits up a Hadamard into three spiders and needs to be implemented as two separate actions.

The *Unfuse* transformation is especially challenging to implement since it requires choosing a subset of the edges connected to the selected node. As the number of edges connected to a spider is in principle unbounded, defining multiple *Unfuse* actions, each corresponding to separating the spider with a specific division of its edge set is not feasible. As a solution, we split up the *Unfuse* transformation into multiple consecutive actions of three types. First, the *start Unfuse* action selects a spider. After that, the selected spider is marked by a new node feature and the agent can only select one of two actions at each step: It can iteratively either use the *Mark edge* action on an edge connected to the selected spider or select the *stop Unfuse* action. Once the *stop Unfuse* action is selected, the spider is split and all previously marked edges (orange edges in Figure 2) are moved to the newly created spider. The angle remains fully at the original spider. To also split the angle between both spiders, as an extension, multiple different *stop Unfuse* actions could be defined, each standing for a different angle of the newly created spider. Due to the symmetry of the *Bialgebra right* transformation, it can not be identified with a single unique edge. Instead, it is applied if the agent selects one of the corresponding 4 colored edges. Due to its potentially global properties, the *Copy* transformation is only implemented in one direction. In principle, the other direction could be implemented similarly as the *Unfuse* action by first iteratively

marking all participating nodes.

In total, the agent can choose from 6 different actions for each node and each edge of the considered diagram. Additionally, the agent can always select a global *Stop* action to end a trajectory if it expects that it can't optimize the diagram any further. To enable more efficient training, we mask actions that are not allowed in their local environment by setting their probability to 0. Finally, after each step, possible *Identity* and *Hadamard loop* transformations are applied automatically and redundant edges are removed. We also delete parts of the ZX-diagram that are disconnected from all ingoing and outgoing edges, as they correspond to simple scalar factors.

To encode ZX-diagrams as observations supplied to the agent, we represent them as undirected graphs with one-hot encoded node features. Each node has a color feature that can either be Z-spider, X-spider, Hadamard, Input, or Output and is, therefore, represented as a 5 dimensional vector. For example, the color feature of an X-spider would be $[0, 1, 0, 0, 0]$. The Input and Output nodes are used to define the ingoing and outgoing edges of the diagram by being connected to their otherwise open end. Additionally, each node has an angle feature that can either be an unspecified placeholder angle α, multiples of $\pi/2$, or specify that the node is not a spider and doesn't have an angle. The angle feature, therefore, is a 6 dimensional vector. The discrete multiples of $\pi/2$ are necessary to evaluate the possibility of the transformation rules depicted in Figure 2. Finally, each node has a binary feature indicating whether the node has been marked by the *start Unfuse* action. The complete feature vector of each node n, x_0^n, given to the agent's policy is then all of its features concatenated into a single vector, resulting in a 12 dimensional vector. The feature vector $e_0^{(n,m)}$ of the edge connecting node n and m contains just a single number that is 0 if the edge has not been marked by the *Mark edge* action and 1 otherwise.

Finally, a vital part of every RL algorithm is the definition of the reward of the agent as it determines the optimization goal. To demonstrate the utility of our algorithm we choose a reward that is computationally cheap to evaluate and intuitive to understand for humans but still requires non-trivial strategies to maximize: The difference in node number of the diagram before and after action application. As the sum of those differences corresponds to the total change in node number, the agent tries to minimize the total amount of nodes in the diagram at the end of the trajectory.

4 Neural network architecture

The use of a graph neural network (GNN) to encode the agent's policy has several advantages: As we suppose the ideal policy depends only on the local structure of the ZX-diagram, we expect the GNN to train more efficiently and generalize better to unseen diagrams than other neural network architectures. Also, unlike a dense neural network, the GNN can handle any size of input data. Therefore, it can be efficiently trained on relatively small diagrams and later straightforwardly applied to much bigger diagrams.

As an input, the GNN directly takes the graph representation of a ZX-diagram. First, 6 message-passing layers [30] are applied to the graph. At each layer i, the node feature vectors x_i^n are updated according to

$$x_{i+1}^n = \phi_i \left(x_i^n, \sum_{x_i^m \in \mathcal{N}_n} [\psi_i(x_i^n, x_i^m, e_i^{(n,m)})] \right), \tag{3}$$

where $\mathcal{N}_n$ are the nearest neighbors of node n, and ϕ_i and ψ_i are single dense neural network layers. We also update the edge feature vectors at each layer according to

$$e_{i+1}^{(n,m)} = \theta_i \left(e_i^{(n,m)}, x_i^n, x_i^m \right), \tag{4}$$

where θ_i is also a single dense neural network layer. After the message-passing layers, we apply the multi-layer perceptron $\chi_{\text{node}}\left(x_{\text{f}}^n\right)$ to the final features of each node x_{f}^n and the multi-layer perceptron $\chi_{\text{edge}}\left(e_{\text{f}}^{(n,m)}\right)$ to the final features of each edge $e_{\text{f}}^{(n,m)}$. The networks χ_{node} and χ_{edge} have 6 output neurons each which are interpreted as the unnormalized log-probabilities of the possible actions (see Figure 1).

As the *Stop* action of the agent depends not only on the local structure of the graph but also on global features, we treat it differently from the other actions by computing its unnormalized log-

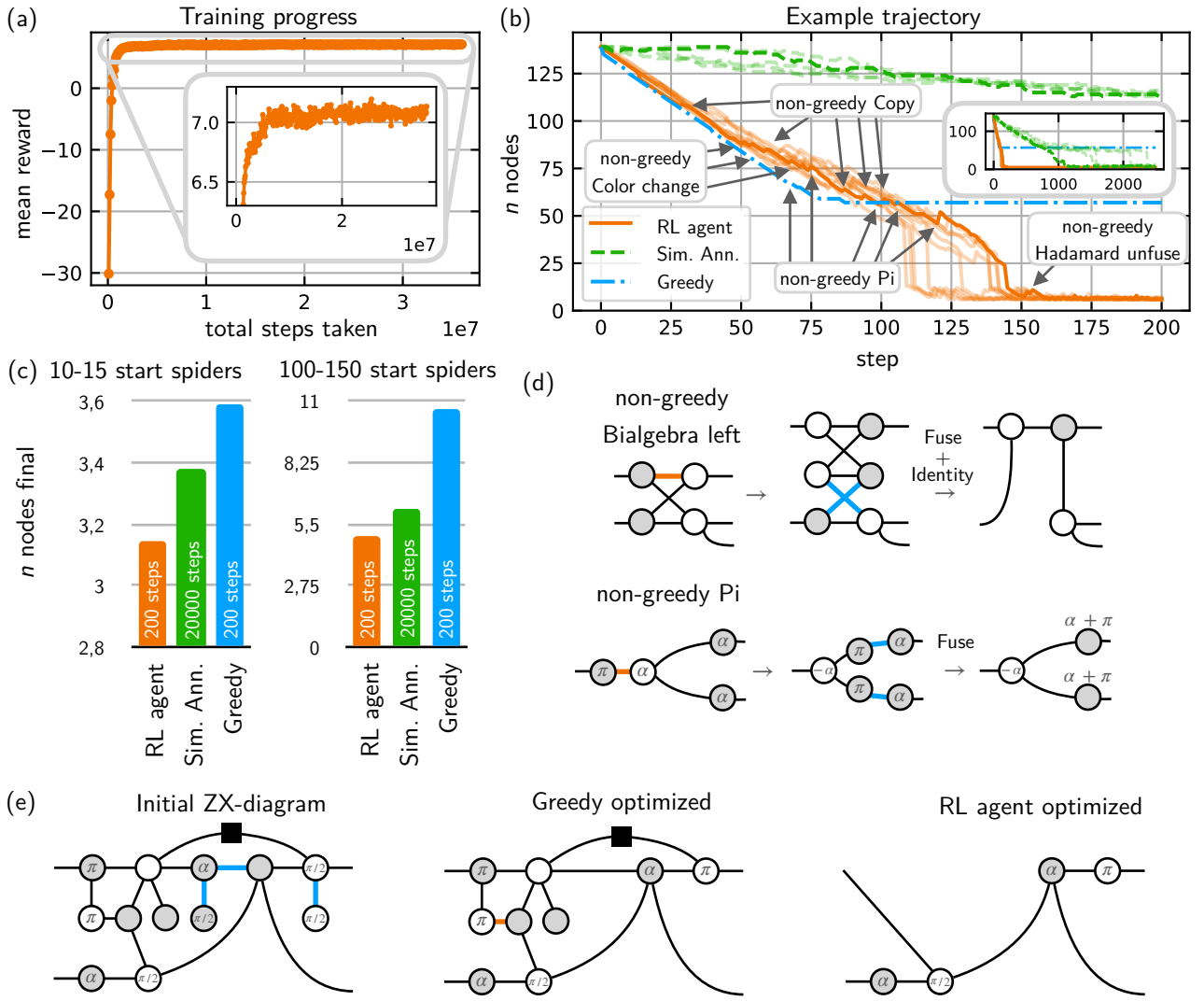

Figure 3: Results. (a) Training progress as the agent is trained to reduce the node number in random ZX-diagrams. Mean cumulative reward of the agent per trajectory against total steps taken in the environment. (b) Optimization of an example ZX-diagram ten times larger than the RL agent's training diagrams. Number of nodes in the ZX-diagram against each action taken for the RL agent (orange), a greedy strategy (blue), and simulated annealing (green). For the RL agent and simulated annealing, multiple trajectories are plotted (transparent). The RL agent and simulated annealing significantly outperform the greedy strategy in terms of cumulative reward with the RL agent requiring an order of magnitude less steps than simulated annealing (inlay). Actions taken by the RL agent that intermittently increase the node number (i.e. non-greedy actions) are indicated by arrows. (c) Average number of nodes after optimization of 1000 ZX-diagrams with 10-15 initial spiders (left), which is the size the agent was trained on, and 100-150 initial spiders (right). Hyperparameters for simulated annealing are optimized to give good performance on two example diagrams and then kept fixed for all diagrams. The agent outperforms both simulated annealing and the greedy strategy on average, while also requiring much fewer steps than simulated annealing. (d) Two examples of non-greedy actions learned by the agent (orange lines), that lead to a positive cumulative reward by consecutive *Fuse* actions (blue lines). (e) Example ZX-diagram sampled from the agent's training set. The greedy strategy can reduce the node number by applying 3 *Fuse* actions (blue lines) while the agent further optimizes the diagram beginning with a non-greedy *Pi* action (orange line).

probability according to

$$p_{\mathrm{s}} = \chi_{\mathrm{stop}}\left(C, \underset{n}{\mathrm{MEAN}}\left(x_{\mathrm{f}}^{n}\right), \underset{(n,m)}{\mathrm{MEAN}}\left(e_{\mathrm{f}}^{(n,m)}\right)\right), \tag{5}$$

where χ_{stop} is a multi-layer perceptron, the MEAN functions are taken over the final node/edge features, and C is a vector containing global information about the amount of each node type, edges and allowed actions.

For an efficient implementation of the GNN, we use the TensorFlow-GNN software package [31]

with custom layers to handle undirected edges. For further details on the network architecture and implementation see Appendix C.

5 Results

5.1 Training

We train the agent to reduce the node number in randomly sampled ZX-diagrams with 10-15 initial spiders (for details on the diagram sampling see Appendix A). The agent is trained for a total of $36 * 10^6$ total actions. However, it already reaches its optimal performance around $9 * 10^6$ actions as shown in Figure 3 (a). To evaluate the trained agent, we sample 1000 new ZX-diagrams of the same size as the training set and optimize them for 200 steps. We then calculate the average of the minimum number of nodes found during optimization which is significantly lower than the number of initial nodes. Next, we want to answer the question of whether the learned policy can straightforwardly be applied to larger diagrams by repeating the same evaluation on ZX-diagrams with 100-150 initial spiders. Even though the agent was only trained on diagrams an order of magnitude smaller, it can reduce the number of nodes in the diagram substantially, thereby highlighting the powerful generalization ability of GNNs [see Figure 3 (c)]. To demonstrate the need for non-trivial strategies of the trained agent to achieve these results, we show two selected actions that initially increase the spider number but later lead to an overall positive cumulative reward in Figure 3 (d).

Training the agent takes around 41 hours on a single compute node with 32 CPUs and 2 GPUs. We run multiple environments in parallel on the CPUs during the sampling phase and train the agent distributed on both GPUs. The implementation of the algorithm could directly take advantage of larger compute nodes to speed up training time.

5.2 Comparison with other techniques

To better estimate the agent's performance, we compare it with a greedy strategy and simulated annealing. The greedy strategy always selects the action with the highest possible reward as long as there are actions with a non-negative reward available. If there are multiple actions leading to the highest possible reward, the greedy strategy chooses randomly out of them. Simulated annealing is a probabilistic strategy for non-convex global optimization problems [32]. We optimize its hyperparameters, i.e. the start temperature and temperature annealing schedule, by hand on two example diagrams [used for Figure 3 (b) and (e)] and then keep them fixed. To compare the different strategies, we evaluate them on the same sets of 1000 diagrams as the RL agent. The RL agent on average outperforms both simulated annealing and the greedy strategy on diagrams the size of the training set as well as on diagrams a magnitude of order larger while requiring much fewer steps than simulated annealing [see Figure 3 (c)]. Moreover, the RL agent needs on average less than 4 s to simplify a diagram with 100-150 initial spiders running on a single GPU and single CPU while simulated annealing with 20000 steps needs over 40 s and the greedy strategy over 100 s, albeit running on only a CPU. For more details on the simulated annealing algorithm and its hyperparameters see Appendix D.

5.3 Analysis of learned policy

While deep neural networks have been successfully employed to solve a wide range of problems, they are often regarded as a 'black box method' due to difficulties in interpreting their learned strategies. However, it is in principle highly desirable to gain some insight into how the neural networks arrive at their predictions [33]. For graph neural networks, an interesting quantity is how local their learned strategy is, i.e. how far away predictions on nodes or edges are influenced by the node and edge features of the diagram. Therefore, we evaluate how far away from a chosen action the ZX-diagram still influences the agent's decision.

To this end, we optimize ZX-diagrams with the agent until 1000 actions of each type are sampled. For each sampled action and the corresponding ZX-diagram, we then build up the diagram in layers around the node/edge identified with the action. Layer n is defined as all nodes that can be reached in n steps by traversing the diagram from the starting point. For each layer n and the corresponding sub-diagram spanning only nodes up to this layer, we compute the agent's unnormalized probability of sampling the original action P_{layer}. We deliberately choose not to normal-

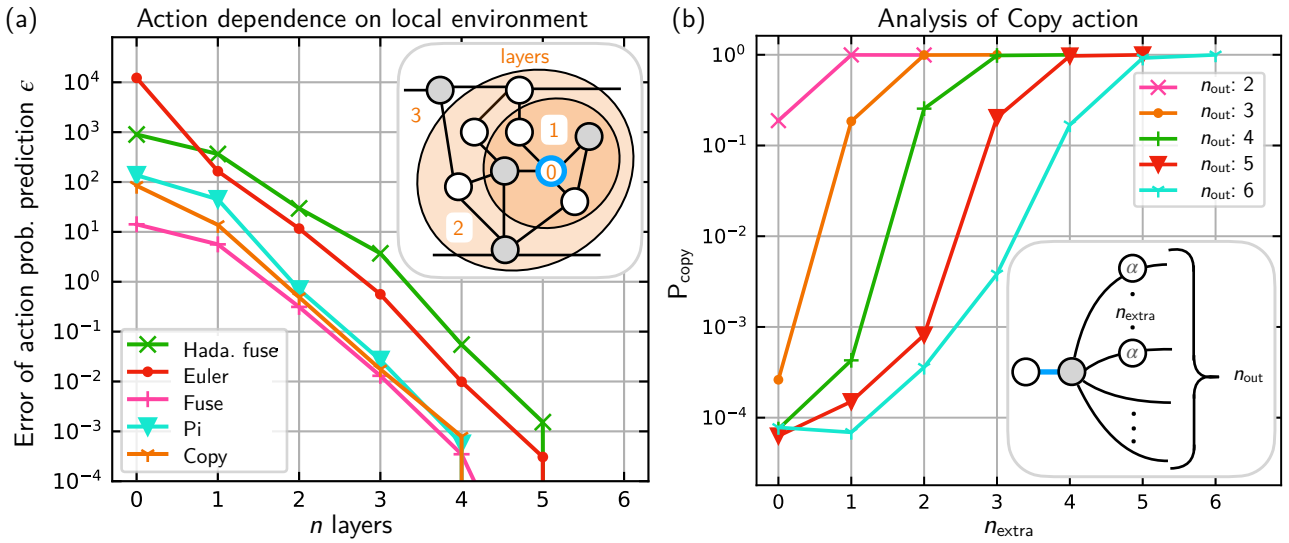

Figure 4: Analysis of learned policy. (a) Action dependence on the local environment. 1000 actions of each type are sampled by the agent. Then, for each action and the diagram in which it was chosen, sub-diagrams are built up in layers around the node/edge identified with the action (see inlay). For each sub-diagram spanning only the nodes in a specific layer, we compute the agent's unnormalized probability of sampling the chosen action P_{layer} and compute the difference ϵ to its probability P_{complete} in the full diagram, where we define ϵ in Equation (6). We plot the average of this difference against the number of layers for 5 action types. (b) Probability of sampling the *Copy* action on the blue edge in the diagram depicted in the inlay for multiple outputs of the diagram n_{out} and multiple additionally inserted spiders on the outputs n_{extra}. The ideal strategy is to select the *Copy* action for $n_{\text{out}} - n_{\text{extra}} \leq 2$. The agent approximately learns the ideal policy.

ize the probabilities, as otherwise far away action probabilities would influence our results through the normalization constant even though no actual information traveled through the GNN.

In Figure 4 (a) we plot the average over the 1000 sampled actions of the quantity ϵ which captures how different P_{layer} is from the unnormalized probability in the full diagram P_{complete}. We define ϵ as

$$\epsilon = \max\left(\frac{P_{\text{layer}}}{P_{\text{complete}}}, \frac{P_{\text{complete}}}{P_{\text{layer}}}\right) - 1, \qquad (6)$$

where $\epsilon = 0$ indicates that P_{layer} and P_{complete} are equal. The max function is necessary to give meaningful values when averaging this quantity. We find that to predict the agent's policy with an accuracy of 1%, information of 3-5 layers is required. Results for all action types are shown in Figure 7.

Finally, we compare the agent's policy in a simple scenario to the, in this case, known optimal policy. Specifically, we take a closer look at the *Copy* action by evaluating its probability in a class of example diagrams as shown in Figure 4 (b). A phaseless Z-spider is connected to a phaseless X-spider with n_{out} additional edges. On n_{extra} of those edges, Z-spiders with arbitrary phase are inserted (see inlay). We plot the probability P_{copy} of applying the *Copy* action to the edge connecting the phaseless spiders against n_{extra} for several n_{out}. The ideal strategy in this diagram is to apply the *Copy* action if $n_{\text{out}} - n_{\text{extra}} \leq 2$ as then multiple *Fuse* actions are enabled, leading to a cumulative positive reward. The agent learns this ideal strategy to good approximation even though it was only trained on random ZX-diagrams and never specifically on diagrams of the type considered here.

6 Outlook

In this work, we have introduced a general scheme for optimizing ZX-diagrams using reinforcement learning with graph neural networks. We showed that the reinforcement learning agent learns non-trivial strategies and generalizes well to diagrams much larger than included in the training set. The presented scheme could be applied to a wide range of problems currently tackled by heuristic and approximate algorithms or simulated annealing.

For example, in [6] the authors speed up tensor

network simulations of quantum circuits by optimizing the graph property treewidth of the corresponding ZX-diagram using simulated annealing, which could straightforwardly be replaced by a reinforcement learning agent.

In [7], a deterministic algorithm for simplification of quantum circuits using ZX-calculus is introduced. The used transformation set is restricted to just two kinds of actions to preserve a special graph property of the ZX-diagram called gFlow, guaranteeing an efficient extraction of a quantum circuit from the optimized diagrams. Later, a heuristic modification was proposed to reduce the number of two-qubit gates in the resulting circuits [8]. Meanwhile, also other gFlow preserving rules have been found [34]. However, it is currently unclear when these rules should be applied for the goal of circuit optimization. In future work, a reinforcement learning agent could be trained including all gFlow preserving rules with a reward dependent on the efficiently extracted quantum circuit corresponding to the diagram, thereby taking advantage of new rules and replacing human heuristics with a learned strategy. The agent's reward could, for example, be the total gate, two-qubit gate, or T-gate count.

During the final preparations of this manuscript, a master thesis using reinforcement learning for quantum circuit compilation with ZX-calculus, albeit using convolutional neural networks, was released [35].

7 Data availability

Python code of the custom reinforcement learning algorithm using graph neural networks and neural network weights of the trained agents are publicly available on GitHub [36].

Acknowledgements

We thank Jonas Landgraf, Jan Olle, and Remmy Zen for fruitful discussions. This research is part of the Munich Quantum Valley, which is supported by the Bavarian state government with funds from the Hightech Agenda Bayern Plus.

A Sampled diagrams

To enable the agent to simplify a wide range of ZX-diagrams, we sample a diverse set of diagrams during training. A typical example is shown in Figure 3 (e). Each new ZX-diagram is constructed with the following steps: The number of inputs and outputs is sampled uniformly between 1 and 3, while the number of initial spiders n_{init} is sampled uniformly between 10 and 15. The amount of Hadamards is then sampled between 0 and $\lfloor 0.2 n_{\text{init}} \rfloor$. The angles of the initial spiders can be one of 0, π, $\pi/2$, and α. To determine the angles of the spiders, we uniformly sample a number between 0 and 1 for each angle type, reduce the number for π, $\pi/2$, α by a factor 0.4 and then normalize the result to a probability distribution from which we sample the angle of each spider. We then uniformly sample the expected number of neighbors n_{neigh} per spider between 2 and 4. From this, we compute the edge probability p_{edge} such that when we create each possible edge in the diagram with p_{edge} we will have an expected amount of n_{neigh} neighbors per spider. We then add each possible edge between all pairs of spiders with probability p_{edge} to the diagram. Finally, we apply the automatic actions that we also apply after each action by the RL agent, i.e. removing redundant edges, removing parts of the diagram not connected to any input or output, and applying all possible *Identity* and *Hadamard loop* transformations. For the performance evaluation of the agent on bigger diagrams we instead sample the number of initial spiders n_{init} between 100 and 150.

B Details on custom PPO algorithm

PPO is an actor-critic RL method with a policy network predicting action probabilities and a critic network predicting the so-called advantage of a specific action [29]. The critic network is only used during training to reduce the variance in gradient update steps. Due to the variable size of our observations and action space, we use a custom implementation of PPO. During the sampling phase of the training, we run n_{env} environments in parallel for n_{max} steps each. Then, the agent's experiences are randomly split into minibatches of size $n_{\text{minibatch}}$ which the agent's policy and critic network is then trained on for one gradient step. After the agent is trained on all minibatches, they are reshuffled and another round of training starts for a maximum of n_{train} steps. However, if the Kullback-Leibler diver-

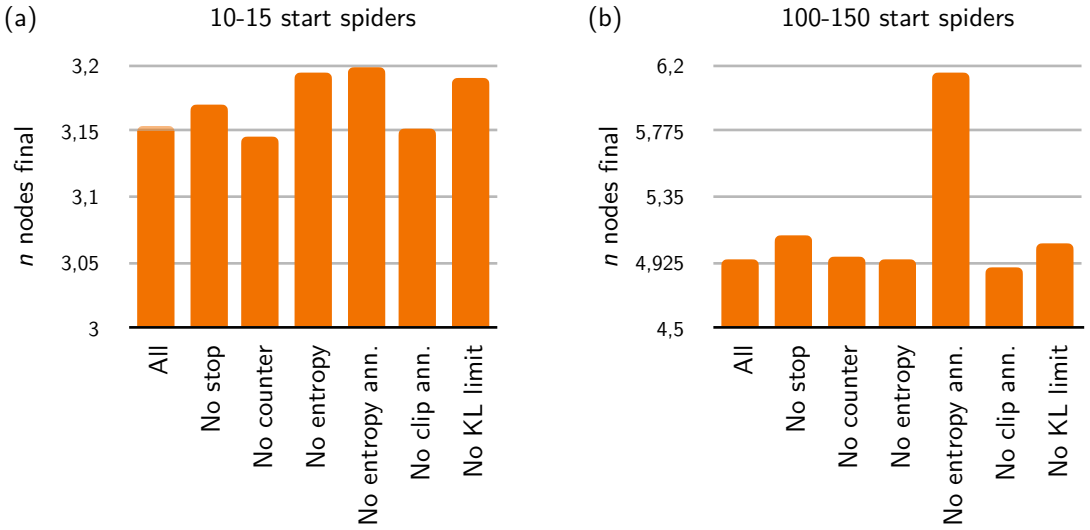

Figure 5: Ablation studies. Average number of nodes left after optimization through an agent trained without a certain feature of the PPO algorithm evaluated over 1000 ZX-diagrams with 10-15 initial spiders (a) and 100-150 initial spiders (b). Two agents with all features were trained resulting in similar performance (leftmost bars in each diagram). The features that were switched off are the *Stop* action of the agent, the stop counter (see Appendix C), the entropy bonus ϵ, the annealing of ϵ, the annealing of the clip range c, and the early stopping of gradient updates if a Kullback-Leibler divergence of c_{KL} is exceeded.

gence, estimated as in [37], between the agent's newly trained policy and the policy used in the last sampling phase gets larger than the constant c_{KL} we stop the training early and start a new sampling phase. This is not a standard feature of PPO algorithms but has e.g. been implemented in [38]. We linearly anneal both the clip range c of the PPO algorithm (as defined in [29]) and the entropy coefficient ϵ, which rewards higher entropy of the policy during training leading to more exploration. During training, we clip all gradients to a maximum of c_{absgrad} and also clip the norm of the gradients of a minibatch to c_{normgrad}. For the gradient updates, we use the ADAM optimizer [39] with a learning rate η and exponential moment decay rates β_1 and β_2. All parameter values are summarized in Table 1, chosen as suggested in [29, 40], and not further optimized.

We perform ablation studies on some features of the PPO algorithm by switching them off and training a new agent without them. The results are summarized in Figure 5. Entropy annealing has a significant positive impact on the agent's performance when simplifying large diagrams. As a policy with high entropy is more probabilistic, it might need more than the 200 given steps to fully simplify a large diagram. All other features don't impact performance significantly. However, we did not optimize the hyperparameters of any of the features which might further increase the performance of the agent.

parameter	value
n_{env}	90
n_{max}	1000
$n_{\mathrm{minibatch}}$	3000
n_{train}	10
c_{KL}	0.01
c	0.2
ϵ	0.1
c_{absgrad}	100
c_{normgrad}	0.5
PPO policy loss γ	0.99
PPO policy loss λ	0.9
η	$3 * 10^{-4}$
β_1	0.9
β_2	0.999

Table 1: Parameter values used in the PPO algorithm. For the definition of γ and λ see [29].

C Details on network architecture

In the policy network, we use 6 message-passing layers. The message functions ψ_i, node features computed by ϕ_i, edge features computed by θ_i, as

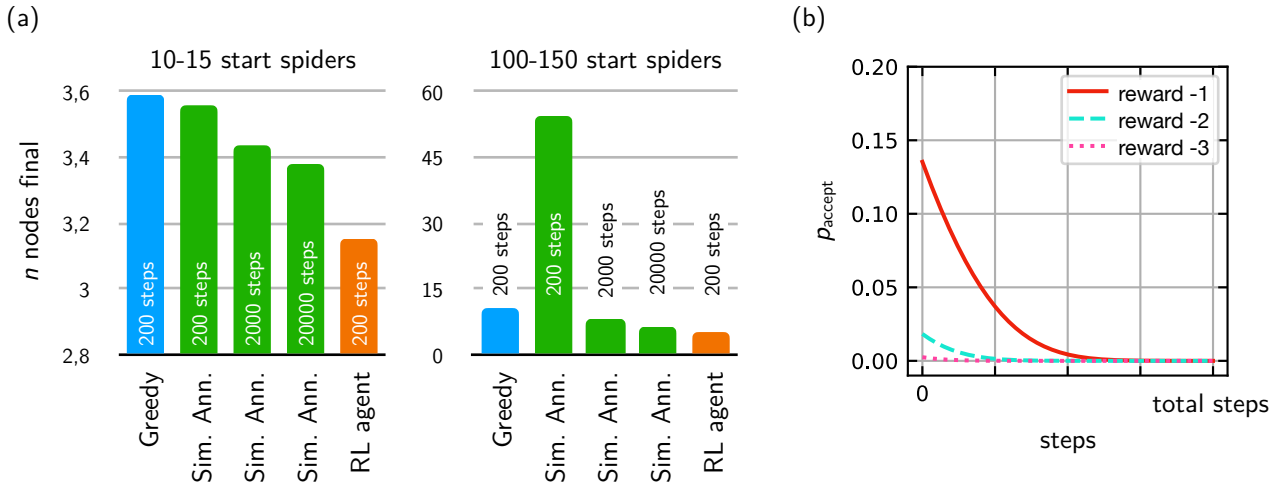

Figure 6: Simulated annealing. (a) Average number of nodes left after optimization through simulated annealing with start temperature $T_{\text{start}} = 0.5$ evaluated over 1000 ZX-diagrams with 10-15 starting spiders (left) and 100-150 starting spiders (right). The temperature decay factor c_{ann} is chosen as $0.01/0.001/0.0001$ for $200/2000/20000$ total steps taken respectively, which results in an acceptance probability of non-greedy actions as shown in (b) for different values of the instantaneous reward of the action.

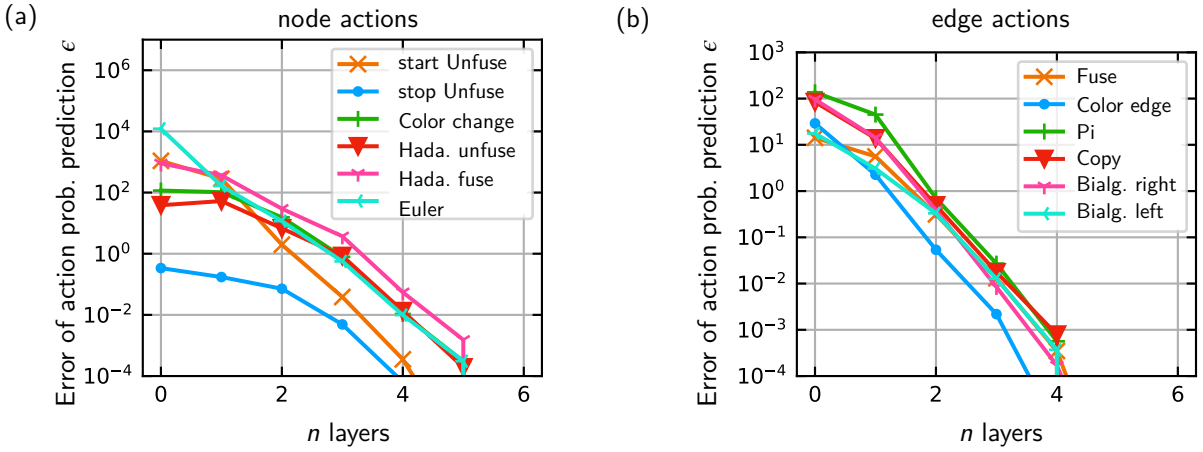

Figure 7: Action probability prediction error ϵ as defined in Equation (6) against the number of layers n as defined in Section 5.3 for all node actions (a) and edge actions (b).

well as the hidden layers of the final action prediction networks χ_{node}, χ_{edge} and χ_{stop}, all contain 128 neurons and use the Tangens hyperbolicus as an activation function. The χ_{node} and χ_{edge} multi-layer perceptrons both have only a single hidden layer, while χ_{stop} has two hidden layers to better learn the more complex global *Stop* action. In addition to the final node/edge states $x_{\text{f}}^{i}/e_{\text{f}}^{(n,m)}$, $\chi_{\text{node}}/\chi_{\text{edge}}$ also get as input an integer number, the stop counter. The stop counter is defined as $\min(20, \text{Steps left in trajectory})$ and tells the agent when a trajectory is about to finish due to the maximum amount of allowed steps being reached.

The global vector C, which is used as part of the input of χ_{stop} contains the number of nodes and the number of edges. Additionally, it holds the number of Z-spiders, X-spiders, Hadamards, spiders with zero/pi/arbitrary angle, and the amount of allowed *Hadamard fuse* and *Euler* actions all normalized by the total spider number and the amount of allowed *Fuse*, *Pi*, *Copy*, *Bialgebra right*, and *Bialgebra left* actions all normalized by the total edge number. Finally, it contains the stop counter and a binary flag, whether the agent has currently selected the *start Unfuse* action. We find that providing the agent global information for predicting the *Stop* action and for predicting the advantage through the critic network is critical to achieving stable training and avoiding exploding gradients as the GNN can otherwise only learn local quantities of the graph.

The critic network has the same network architecture as the network predicting the probability of the *Stop* action but shares no weights with the policy network.

We initialize all trainable parameters of the neural network layers as recommended in [40] using an orthogonal initializer with gain $\sqrt{2}$ for all hidden layers, gain 0.01 for the action prediction networks χ_{node}, χ_{edge} and χ_{stop}, and gain 1 for the final layer of the critic network.

No optimization over the network size or parameters is performed suggesting further possibilities for improving the performance of the RL agent.

D Details on simulated annealing

Simulated annealing is a probabilistic algorithm iteratively transforming the ZX-diagrams. At each step, it randomly selects one of all allowed actions. If the immediate reward r of the action is non-negative, the action is applied. If r is negative, the action is only accepted with probability

$$p_{\text{accept}} = \exp(r/T), \tag{7}$$

where T is the so-called temperature. T is typically continuously decreased during the optimization process. We choose to exponentially anneal T with the start temperature T_{start} at optimization step n_{step} according to

$$T = T_{\text{start}} \exp(-c_{\text{ann}} n_{\text{step}}), \tag{8}$$

where c_{ann} determines the speed of the temperature decay, as it performs better on the example diagrams than linearly annealing T. This may be because the exponential temperature decay leads to a longer nearly greedy phase of the algorithm in the later stages of the optimization.

We further improve the performance of the simulated annealing algorithm by changing the reward structure of the *Unfuse* transformation. Instead of giving 0 reward when the *start Unfuse* action is selected and -1 rewards when the *stop Unfuse* is selected we switch the order of the two rewards. This helps the algorithm to avoid selecting *start Unfuse* in the later, nearly greedy stages of optimization and then getting stuck since it never accepts the negative reward of the *stop Unfuse* action.

We optimize T_{start} and c_{ann} on two diagrams which the greedy strategy can not optimize well [the diagrams used for Figure 3 (b) and (e)] and then keep them fixed while evaluating the performance of simulated annealing on the same set of diagrams, we evaluated the RL agent on. We find that $T_{\text{start}} = 0.5$ performs well with $c_{\text{ann}} = 0.01/0.001/0.0001$ for a maximum of 200/2000/20000 optimization steps. We also tried $T_{\text{start}} = 1$ which performed similar to $T_{\text{start}} = 0.5$ on the example diagrams but considerably worse on average and even higher starting temperatures which even failed to optimize the example diagrams. As shown in Figure 6, simulated annealing performs slightly worse on average while needing a lot more optimization steps than the RL agent.

References

[1] Bob Coecke and Aleks Kissinger. "Picturing quantum processes: A first course in

quantum theory and diagrammatic reasoning". Cambridge University Press. (2017).

[2] Ross Duncan. "A graphical approach to measurement-based quantum computing". In Quantum Physics and Linguistics: A Compositional, Diagrammatic Discourse. Oxford University Press (2013).

[3] Enrique Cervero Martín, Kirill Plekhanov, and Michael Lubasch. "Barren plateaus in quantum tensor network optimization". Quantum **7**, 974 (2023).

[4] Nicholas Chancellor, Aleks Kissinger, Stefan Zohren, Joschka Roffe, and Dominic Horsman. "Graphical structures for design and verification of quantum error correction". Quantum Science and Technology **8**, 045028 (2023).

[5] Liam Garvie and Ross Duncan. "Verifying the smallest interesting colour code with quantomatic.". In Proceedings 14th International Conference on Quantum Physics and Logic. (2107). arXiv:1706.02717.

[6] Tristan Cam and Simon Martiel. "Speeding up quantum circuits simulation using ZX-calculus" (2023). arXiv:2305.02669.

[7] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering. "Graph-theoretic simplification of quantum circuits with the ZX-calculus". Quantum **4**, 279 (2020).

[8] Korbinian Staudacher, Tobias Guggemos, and Sophia Grundner-Culemann. "Reducing 2-qubit gate count for ZX-calculus based quantum circuit optimization". In Proceedings 19th International Conference on Quantum Physics and Logic. (2022). arXiv:2311.08881.

[9] Stefano Gogioso and Richie Yeung. "Annealing optimisation of mixed ZX phase circuits". In Proceedings 19th International Conference on Quantum Physics and Logic. (2023). arXiv:2206.11839.

[10] Aleks Kissinger and John van de Wetering. "Reducing the number of non-clifford gates in quantum circuits". Phys. Rev. A **102**, 022406 (2020).

[11] David Winderl, Qunsheng Huang, and Christian B Mendl. "A recursively partitioned approach to architecture-aware ZX polynomial synthesis and optimization" (2023). arXiv:2303.17366.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning" (2013). arXiv:1312.5602.

[13] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play". Science **362**, 1140–1144 (2018).

[14] Jens Kober, J. Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey". The International Journal of Robotics Research **32**, 1238–1274 (2013).

[15] Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. "Learning dexterous in-hand manipulation". The International Journal of Robotics Research **39**, 3–20 (2020).

[16] Jiaxuan You, Bowen Liu, Zhitao Ying, Vijay Pande, and Jure Leskovec. "Graph convolutional policy network for goal-directed molecular graph generation". Advances in neural information processing systems**31** (2018).

[17] Chence Shi, Minkai Xu, Zhaocheng Zhu, Weinan Zhang, Ming Zhang, and Jian Tang. "Graphaf: a flow-based autoregressive model for molecular graph generation". In International Conference on Learning Representations. (2020). arXiv:2001.09382.

[18] Thomas Fösel, Petru Tighineanu, Talitha Weiss, and Florian Marquardt. "Reinforcement learning with neural networks for quantum feedback". Phys. Rev. X **8**, 031084 (2018).

[19] Jan Olle, Remmy Zen, Matteo Puviani, and Florian Marquardt. "Simultaneous discovery of quantum error correction codes and encoders with a noise-aware reinforcement learning agent" (2023). arXiv:2311.04750.

[20] Ryan Sweke, Markus S Kesselring, Ev-

ert P L van Nieuwenburg, and Jens Eisert. "Reinforcement learning decoders for fault-tolerant quantum computation". Machine Learning: Science and Technology **2**, 025005 (2020).

[21] Yuval Baum, Mirko Amico, Sean Howell, Michael Hush, Maggie Liuzzi, Pranav Mundada, Thomas Merkh, Andre R.R. Carvalho, and Michael J. Biercuk. "Experimental deep reinforcement learning for error-robust gate-set design on a superconducting quantum computer". PRX Quantum **2**, 040324 (2021).

[22] Kevin Reuer, Jonas Landgraf, Thomas Fösel, James O'Sullivan, Liberto Beltrán, Abdulkadir Akin, Graham J Norris, Ants Remm, Michael Kerschbaum, Jean-Claude Besse, et al. "Realizing a deep reinforcement learning agent discovering real-time feedback control strategies for a quantum system". Nat. Comm. **14**, 7138 (2023).

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

[24] Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. "Quarl: A learning-based quantum circuit optimizer" (2023). arXiv:2307.10120.

[25] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. "Graph neural networks: A review of methods and applications". AI Open **1**, 57–81 (2020).

[26] Renaud Vilmart. "A near-minimal axiomatisation of ZX-calculus for pure qubit quantum mechanics". In 2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). Pages 1–10. (2019).

[27] John van de Wetering. "ZX-calculus for the working quantum computer scientist" (2020). arXiv:2012.13966.

[28] Richard S Sutton and Andrew G Barto. "Reinforcement learning: An introduction". MIT press. (2018).

[29] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal policy optimization algorithms" (2017). arXiv:1707.06347.

[30] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. "Neural message passing for quantum chemistry". In International conference on machine learning. Pages 1263–1272. PMLR (2017).

[31] Oleksandr Ferludin, Arno Eigenwillig, Martin Blais, Dustin Zelle, Jan Pfeifer, Alvaro Sanchez-Gonzalez, Wai Lok Sibon Li, Sami Abu-El-Haija, Peter Battaglia, Neslihan Bulut, et al. "Tf-gnn: Graph neural networks in tensorflow" (2022). arXiv:2207.03522.

[32] Darrall Henderson, Sheldon H. Jacobson, and Alan W. Johnson. "The theory and practice of simulated annealing". Pages 287–319. Springer US. Boston, MA (2003).

[33] Lingfei Wu, Peng Cui, Jian Pei, and Liang Zhao. "Graph neural networks: Foundations, frontiers, and applications". Springer Singapore. Singapore (2022). url: graph-neural-networks.github.io.

[34] Tommy McElvanney and Miriam Backens. "Flow-preserving ZX-calculus rewrite rules for optimisation and obfuscation". In Proceedings of the Twentieth International Conference on Quantum Physics and Logic. Volume 384 of Electronic Proceedings in Theoretical Computer Science, pages 203–219. Open Publishing Association (2023).

[35] Jan Nogué Gómez. "Reinforcement learning based circuit compilation via ZX-calculus". Master's thesis. Universitat de Barcelona. (2023).

[36] Maximilian Nägele. "Code for optimizing ZX-diagrams with deep reinforcement learning". GitHub repository (2023). url: github.com/MaxNaeg/ZXreinforce.

[37] Schulmann John. "Approximating KL divergence". personal blog (2020). url: http://joschu.net/blog/kl-approx.html.

[38] Schulmann John. "Modular rl". GitHub repository (2018). url: github.com/joschu/modular_rl.

[39] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization" (2014). arXiv:1412.6980.

[40] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. "What matters for on-policy deep actor-critic methods? a large-

scale study". In International conference on learning representations. (2020). arXiv:2006.05990.