Scaling Automated Quantum Error Correction Discovery with Reinforcement Learning

Jan Olle [1] **Remmy Zen** [1] **Matteo Puviani** [1] **Florian Marauardt** [1 2]

Abstract

In the ongoing race towards experimental implementations of quantum error correction (QEC), finding ways to automatically discover codes and encoding strategies tailored to the qubit hardware platform is emerging as a critical problem. Reinforcement learning (RL) has been identified as a promising approach, but so far it has been severely restricted in terms of scalability. In this work, we significantly expand the power of RL approaches to QEC code discovery. Explicitly, we train an RL agent that automatically discovers both QEC codes and their encoding circuits from scratch. We show its effectiveness with up to 25 physical qubits and distance 5 codes, while presenting a roadmap to scale up to 100 qubits and distance 10 codes in the near future.

1. Introduction

Quantum error correction (Inguscio et al., 2007; Girvin, 2023) (QEC) protects quantum information by encoding the state of a logical qubit into several physical qubits and is crucial to ensure that quantum technologies such as quantum communication or quantum computing can achieve their groundbreaking potential.

The past few years have witnessed dramatic progress in experimental realizations of QEC on different experimental platforms (Krinner et al., 2022; Ryan-Anderson et al., 2021; Postler et al., 2022; Cong et al., 2022; Acharya et al., 2023; Sivak et al., 2023), including superconducting qubits, ion traps, quantum dots, and neutral atoms. Given the strong differences in all of these platforms, there is a strong need for a flexible and efficient scheme to automatically discover not only codes but also efficient encoding circuits, adapted to the hardware at hand.

At the same time, Artificial Intelligence (AI) is transforming scientific discovery (Wang et al., 2023). Within, Reinforcement Learning (RL) is a promising artificial discovery tool for QEC strategies. The task to solve is encoded in a *reward* function, and the aim of RL training algorithms is to maximize such a reward over time. RL can provide new answers to difficult questions, in particular in fields where optimization in a high-dimensional search space plays a crucial role. For this reason, RL can be an efficient tool to tackle the problem of QEC code construction and encoding.

In our work, we significantly expand the scaling capabilities of RL for QEC code search by introducing two critical components:

1. An efficiently computable and general RL reward based on the Knill-Laflamme error correction conditions (defined further below).

2. A highly parallelized custom-built Clifford circuit simulator (defined further below) that runs entirely on modern AI chip accelerators such as GPUs or TPUs.

The main results that are enabled by this strategy are the following:

1. Effortless discovery of codes and encoders with code distances from 3 (found in tens of seconds) to 5 (found in tens of minutes to a few hours) with up to 25 physical qubits.

2. A scalable platform for artificial scientific discovery of QEC strategies based on RL that potentially allows discovery of distance 8-10 codes on a *single* GPU, while offering further scaling opportunities on distributed machines.

Some of the work presented in this paper can also be found as part of the paper (Olle et al., 2024).

[1]Max Planck Institute for the Science of Light, Staudtstraße 2, 91058 Erlangen, Germany [2]Department of Physics, Friedrich-Alexander Universität Erlangen-Nürnberg, Staudtstraße 5, 91058 Erlangen, Germany. Correspondence to: Jan Olle <jan.olle@mpl.mpg.de>.

AI for science workshop of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

2. Related work

The first example of RL-based automated discovery of QEC strategies (Fösel et al., 2018) did not rely on any human knowledge of QEC concepts. While this allowed exploration without any restrictions, it was limited to only small qubit numbers (at most, 4 or 5). More recent works have moved towards optimizing only certain QEC subtasks, injecting substantial human knowledge (inductive biases). For example, RL has been used for optimization of given QEC codes (Nautrup et al., 2019), and to discover tensor network codes (Mauron et al., 2023) or codes based on "Quantum Lego" parametrizations (Su et al., 2023; Cao & Lackey, 2022). Additionally, RL has been used to find efficient decoding processes (Andreasson et al., 2019; Sweke et al., 2020; Colomer et al., 2020; Fitzek et al., 2020) and self-correcting control protocols (Metz & Bukov, 2023).

In (Cao et al., 2022), the authors also set themselves the task of finding both codes and their encoding circuits. However, this was done using variational quantum circuits involving continuously parametrized gates, which leads to much more costly numerical simulations and eventually only an approximate QEC scheme. In fact, with that approach it was not possible to scale to $d = 5$ codes due to prohibitive computational costs. In contrast, our RL-based approach does not rely on any human-provided circuit ansatz, can use directly any given discrete gate set and is able to scale up to $d = 5$ (and in principle even higher) codes, exploiting highly efficient Clifford simulations.

3. Theoretical background

3.1. Stabilizer codes

The stabilizer formalism (Gottesman, 1997) provides a compact description of quantum states and processes. The central idea is to describe quantum states by listing the set of operators of which that state is an eigenvector with eigenvalue $+1$. Such operators are said to *stabilize* the state. Given n qubits, a state can be described by listing n operators. A particularly useful set of operators are Pauli strings, which are composed of Kronecker products of the Pauli matrices,

$$\begin{aligned} I &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \\ Y &= \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \end{aligned} \tag{1}$$

For example, $IXYIZ$ (which is shorthand for $I \otimes X \otimes Y \otimes I \otimes Z$) is a Pauli string. The *weight* of a Pauli string is the number of non-identity Pauli matrices inside it (e.g. $IXYIZ$ has weight 3). Pauli strings form a group under matrix multiplication (Gottesman, 1997) and can be represented as binary arrays of size $2n$ (Aaronson & Gottesman, 2004). The latter property is where their usefulness resides: instead of 2^n complex-valued coefficients, $2n^2$ binary values suffice to represent quantum states.

Time evolution of stabilizer states are represented in terms of Clifford gates. By definition, these are unitary processes that map Pauli strings to Pauli strings. In contrast, non-Clifford gates (which we don't consider in this work) are also unitary but map Pauli strings to superpositions of Pauli strings. All Clifford gates can be generated by the Hadamard H, the Phase S and the CNOT gates (Aaronson & Gottesman, 2004) (see Appendix A for their definition).

The stabilizer formalism can also describe subspaces, which are called codes. A code that encodes k logical qubits into n physical qubits is a 2^k-dimensional subspace (the *code space* $\mathcal{C}$) of the full 2^n-dimensional Hilbert space. It is completely specified by a set of $n - k$ Pauli strings that *stabilize* it. In fact, these $n - k$ Pauli strings generate a group denoted by $S_{\mathcal{C}} = \langle g_1, g_2, \ldots, g_{n-k} \rangle$, which is called the stabilizer group of $\mathcal{C}$. In order to describe such codes, $2n(n - k)$ bits are needed.

3.2. Error correction conditions

The fundamental theorem in QEC is a set of necessary and sufficient conditions discovered by Knill and Laflamme (KL conditions) in (Knill & Laflamme, 1997) that state that a code $\mathcal{C}$ with associated stabilizer group $S_{\mathcal{C}}$ can *detect* a set of errors $\{E_\mu\}$ (which for our purposes will also be Pauli strings) if and only if they anticommute,

$$\{E_\mu, g_i\} = 0\,, \tag{2}$$

for at least one g_i, or the error itself is harmless, i.e.

$$E_\mu \in S_{\mathcal{C}}\,. \tag{3}$$

The smallest weight in $\{E_\mu\}$ for which any of the above two conditions do not hold is called the *distance* of the code. A quantum code that can correct up to weight-t errors must have a distance of at least $d = 2t + 1$. We follow standard notation and write quantum codes of distance d that encode k logical qubits into n physical qubits as $[[n, k, d]]$.

3.3. Calderbank-Steane-Shor (CSS) codes

CSS codes (Steane, 1996; Calderbank & Shor, 1996), named after A. R. Calderbank, P. Shor, and A. Steane, are a subclass of stabilizer codes with very useful properties. They are defined by their stabilizer group generators being Pauli strings containing either only X's or only Z's (apart from I). Concretely, we write the X-type generators of a CSS code as G_X and the Z-type ones as G_Z. For instance, a well-known example of a CSS code is Steane's $[[7, 1, 3]]$ code, with generators $G_X = \{IIIXXXX,\ IXXIIXX,\ XIXIXIX\}$ and $G_Z = \{IIIZZZZ,\ IZZIIZZ,\ ZIZIZIZ\}$.

An advantage of working with CSS codes is that we can make the binary representation of Pauli strings even more compact. Specifically, we will never encounter a Pauli string with a Y in it, and all Pauli strings will contain either only X's or only Z's. Thus, it suffices to represent Pauli strings with arrays of n bits. Possible ambiguities (e.g. both XX and ZZ would be represented by $(1,1)$) are avoided by labelling which code generators are in G_X and which ones are in G_Z. We can thus represent an $[[n,k]]$ code with $n(n-k)$ bits, getting an improvement of a factor of 2 with respect to generic stabilizer codes.

By construction, detection of X-type and Z-type errors in CSS codes happen *independently*. This implies that Y-type errors are identified when X and Z-type stabilizer measurements fire simultaneously. This conveniently reduces the number of error operators that have to be checked in the KL conditions (2), (3) for an $[[n,k,d]]$ code to

$$\text{num}\left(\{E_\mu\} \mid \text{CSS},\ [[n,k,d]]\right) = \sum_{w=0}^{d-1} \binom{n}{w}, \tag{4}$$

i.e. we have to choose in how many of the n positions of the Pauli strings we place single-qubit errors with a budget of w. In comparison with generic stabilizer codes, an exponential factor 3^w suppression from inside the sum is achieved.

3.4. Reinforcement Learning

Reinforcement Learning (RL) (Sutton et al., 1999; Sutton & Barto, 2018) is designed to discover optimal action sequences in decision-making problems. The goal in any RL task is encoded by choosing a suitable *reward* r, a quantity that measures how well the task has been solved, and consists of an *agent* (the entity making the decisions) interacting with an *environment* (the physical system of interest or a simulation of it). In each time step t, the environment's state s_t is observed. Based on this observation, the agent takes an action a_t which then affects the current state of the environment. For each action, the agent receives a reward r_t, and the goal of RL algorithms is to maximize the expected cumulative reward (return), $\mathbb{E}\left[\sum_t r_t\right]$. A *trajectory* is a sequence of state, action and reward triples that the agent experiences from an initial state to a terminal state. The agent's behavior is defined by the *policy* $\pi_\theta(a_t|s_t)$, which denotes the probability of choosing action a_t given observation s_t, and that we parameterize by a neural network with parameters θ.

Within RL, policy gradient methods (Sutton et al., 1999) optimize the policy by maximizing the expected return with respect to the parameters θ with gradient descent. One of the most successful algorithms within policy gradient methods is the actor-critic algorithm (Konda & Tsitsiklis, 1999). The idea is to have two neural networks: an actor network that acts as the agent and that defines the policy, and a critic network, which measures how good was the action taken by the agent. In this paper, we use a state-of-the-art policy-gradient actor-critic method called Proximal Policy Optimization (PPO) (Schulman et al., 2017), which improves the efficiency and stability of policy gradient methods. We include in Appendix B a review of the PPO algorithm with implementation-specific details used in this work.

4. CSS code discovery with reinforcement learning

The main objective of this work is to explore the scaling limits of the automated discovery of QEC codes and their encoding circuits using RL. We exclusively focus on CSS codes due to their enhanced simulability with classical computers. A scheme of our approach can be found in Fig. 1(a), and the next subsections are dedicated to explaining its constituent parts.

4.1. Encoding circuit

In order to encode the state of k logical qubits on n physical qubits one must find a sequence of quantum gates that will entangle the quantum information in such a way that QEC is possible with respect to a target list of error operators. Initially, we imagine the first k qubits as the original containers of our (yet unencoded) quantum information, which can be in any state $|\psi\rangle \in (\mathbb{C}_2)^{\otimes k}$. The remaining $n-k$ qubits are chosen to each be initialized in the state $|0\rangle$, as shown in Fig. 1(a). These will be turned into the corresponding logical state $|\psi\rangle_L \in (\mathbb{C}_2)^{\otimes n}$ via the application of a sequence of Clifford gates on any of the n qubits. In the stabilizer formalism, this means that initially the stabilizer generators of the code subspace are

$$Z_{k+1},\ Z_{k+2}, \ldots, Z_n\ . \tag{5}$$

The search of stabilizer codes can be restricted to CSS codes by constraining the circuit to be built from an initial layer of Hadamard gates followed by CNOT gates thereafter, as seen in Fig. 1(a). We give a proof of this statement in Appendix C, but the main idea is that X's and Z's never mix. The number of Hadamard gates applied at the beginning is the cardinality of G_X (the set of code generators of the X kind), while the cardinality of G_Z (the set of code generators of the Z kind) is $n-k-\text{num}(H)$.

We could ask the RL agent to provide the full circuit (both the Hadamards and the CNOT gates). However, we choose a mixed *human-AI* strategy where *we* are the ones deciding the content of the Hadamard layer and where the agent has to discover a suitable encoding sequence of CNOT gates for the particular list of errors under consideration. In this way, we simplify the task of the agent as much as possible (see Fig. 1(a) for an example).

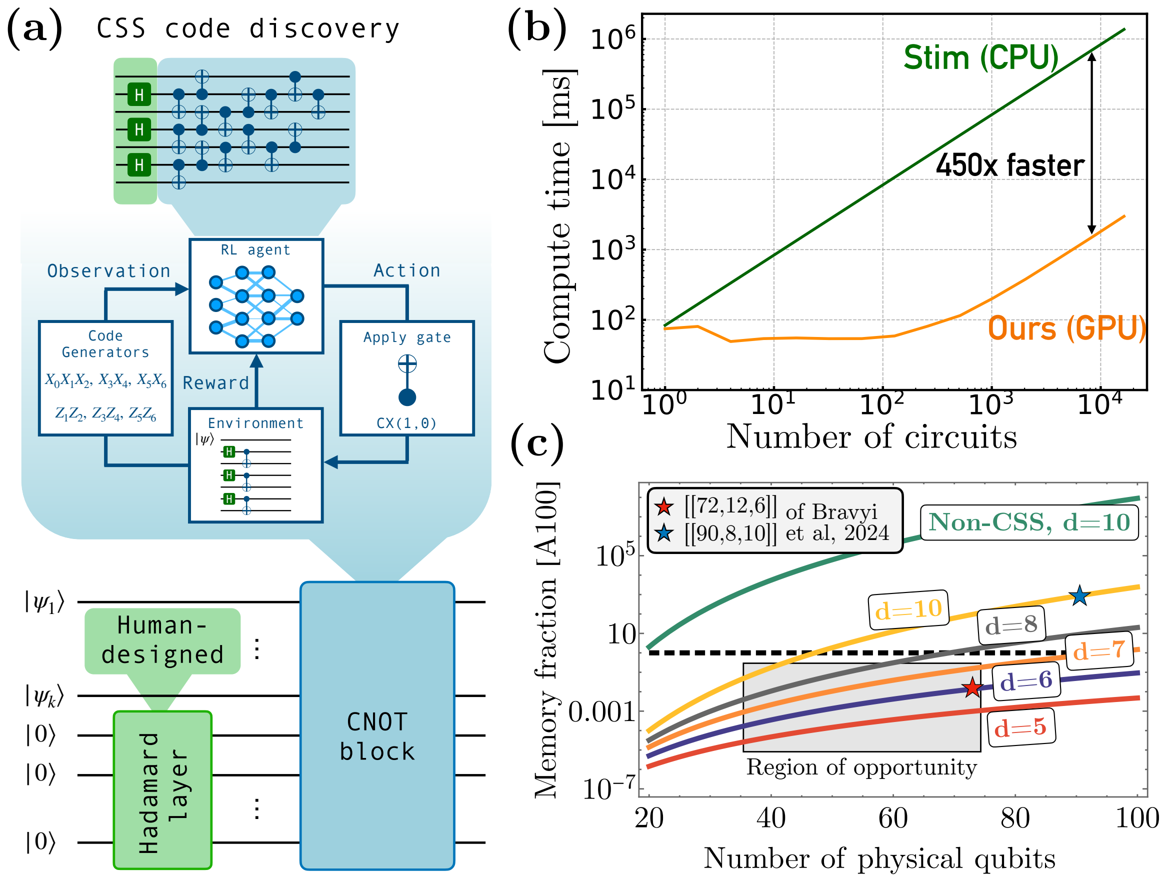


Figure 1. **(a)** Code and encoding discovery with reinforcement learning: scheme of our framework. An initial layer of Hadmard gates is chosen and the RL agent's task is to find a suitable circuit of CNOTs that is able to correct a target list of errors (which enter through the reward). We use PPO, which is an actor-critic methods with two neural networks. They both receive as observation a binary representation of the code generators at that given point in time. The actions are discrete and correspond to applying a single CNOT gate (whose possible control and target qubits are determined by the available qubit connectivity). In the present example, Steane's $[[7,1,3]]$ code is rediscovered using a nearest-neighbor qubit connectivity. **(b)** Fast Clifford circuit simulator. We benchmark our custom-built Clifford circuit simulator against STIM (Gidney, 2021) by simulating random Clifford circuits of 1000 gates on 49 qubits. Thanks to our vectorized implementation, we are able to run many circuit in parallel, achieving a 450 times speedup with respect to STIM in this benchmark. **(c)** Scaling CSS (defined in the text) code and encoding discovery to larger code parameters. We show the fraction of the 80 GB of GPU memory needed (NVIDIA A100 GPU) to store all the error operators that are required to reward the agent. We also show for comparison the memory load of stabilizer (non-CSS) code discovery for code distance $d = 10$. We identify a *region of opportunity* where our RL strategy could outperform some of the qLDPC (defined in the text) codes found in (Bravyi et al., 2024) in the near future.

After applying each gate, the $n-k$ code generators (5) are updated. The agent then receives a representation of these generators as input (as its observation) and suggests the next gate (action) to apply. In this way, an encoding circuit is built up step by step, taking into account the available gate set and connectivity for the particular hardware platform.

4.2. Reward

In this work we use a scheme where the cumulative reward (which RL optimizes) is maximized whenever all the Knill-Laflamme conditions are fulfilled. This way, one can verify that the circuit found by the agent has the desired error-correcting capabilities. One implementation of this idea uses what we call the (negative) weighted Knill-Laflamme sum as an instantaneous reward, which we define as:

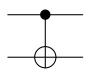
$$r_t = -\sum_{\mu} \lambda_\mu K_\mu \;, \tag{6}$$

where $K_\mu = 0$ if the corresponding error operator E_μ satisfies the KL conditions, and $K_\mu = 1$ otherwise. Here λ_μ are (positive) hyperparameters quantifying how dangerous each error is (i.e. proportional to its occurrence probability p_μ). The reward (6) is zero if and only if all the chosen errors can be detected, at which point we know that the encoding has been successful, and is negative otherwise. The range of μ is determined by (4) upon a choice of target code distance d. Finally, making the reward non-positive favors short gate sequences. An important technical aspect is the following. The second KL condition, Eq. (3), in principle requires computing the entire stabilizer group $S_\mathcal{C}$ of 2^{n-k} elements at every time step, which can be very costly. In practice, almost all errors are always detected through the first KL condition Eq. 2, meaning that generating only a subgroup of $S_\mathcal{C}$ is enough. More detailed aspects of our implementation are explained in Appendix D.

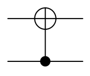
4.3. Accelerated simulator of Clifford circuits

RL algorithms exploit guided trial-and-error loops until a signal of a good strategy is picked up and convergence is reached, so it is of paramount importance that simulations of our RL environment are extremely fast. Thanks to the Gottesman-Knill theorem, the Clifford circuits needed here can be simulated efficiently on classical computers. Optimized numerical implementations of Clifford circuits exist, e.g. STIM (Gidney, 2021). However, in an RL application we want to be able to run multiple circuits in parallel in an efficient, vectorized way that is compatible with modern machine learning frameworks. For that reason, we have implemented our own special-purpose vectorized GPU Clifford simulator. When compared to STIM, we find a $\sim 450\times$ speedup at simulating random Clifford circuits. In particular, we can simulate around 6000 random Clifford circuits of 1000 gates each on 49 qubits in under a second (see Fig. 1(b)).

We achieve such a performance by implementing the action of Clifford gates as binary matrices. In practice, we only need to implement the CNOT gate (H only decides the splitting between G_X and G_Z). Here we show how to implement a simple CNOT gate on a system of two qubits for illustrative purposes. The CNOT transformation rules are the following:



$$\begin{aligned} XI \to XX\ ,\ ZI \to ZI \\ IX \to IX\ ,\ IZ \to ZZ \end{aligned} \tag{7}$$



$$\begin{aligned} XI \to XI\ ,\ ZI \to ZZ \\ IX \to XX\ ,\ IZ \to IZ \end{aligned} \tag{8}$$

Crucially, exchange of `control` and `target` labels turns an X transformation rule into a Z transformation rule. We can thus use a *single* binary matrix per CNOT (we choose the one that implements the X transformation rule) and use the binary matrix representation of the CNOT with exchanged `control` and `target` to transform Z-type stabilizers.

A key aspect is that matrix multiplication happens from the right, i.e. to update a Pauli string with binary representation $(P_1)_{\text{bin}}$ with a gate G_{bin} the order is $(P_1)_{\text{bin}} \cdot G_{\text{bin}}$. Having this in mind, the binary matrix that implements the CNOT rule (7) is

$$\text{CNOT}(0,1)_{\text{bin}} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \ , \tag{9}$$

while the binary matrix representation of (8) is

$$\text{CNOT}(1,0)_{\text{bin}} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \ . \tag{10}$$

In summary, Clifford gates are represented by (n, n) binary matrices and updates in G_X and G_Z are implemented by binary matrix multiplication.

The first KL condition, (2), requires checking anticommutation relations between Pauli strings. Two Pauli strings P_1 and P_2 either commute or anticommute: they anticommute if they overlay with an odd number of different Pauli matrices and commute otherwise. In our case there is only one option: the number of overlaying X and Z single Paulis must be odd. This is computed by the binary dot product between Pauli strings P_X and P_Z. If 1, they anticommute; if 0, they commute.

In the end, we implement all the operations that are required for both simulating the quantum circuits and to check the error correction conditions using binary linear algebra. These are efficiently vectorizable operations using modern machine learning frameworks such as JAX (Bradbury et al., 2018) and are extremely fast on a GPU.

4.4. Scaling limits

Before showing specific circuits and codes found with our approach, we make some estimations on the practical scaling limits of this strategy, which are summarized in Fig. 1(c).

The largest bottleneck is being able to store all error operators needed to compute the reward in the GPU memory (it is always possible to use CPUs with larger memory at the expenses of performance). We thus estimate the amount of memory that would be needed to store all error operators for some code parameters n and d (this calculation is independent of k). We simply count the number of error operators (4), times the amount of bits that have to be specified for each of them. Every error operator is a binary array of size n. Using a standard 8-bit representation for integers, every error operator needs $8n$ bits of memory storage.

We show the results of this estimation in Fig. 1(c), considering what fraction of memory they would occupy in an NVIDIA A100 GPU. The results shown in Fig. 1(c) indicate that this approach can be extended to ~ 100 physical qubits ($d = 6$) and to approximately 40 physical qubits and $d = 10$ in a *single* GPU.

Moreover, we identify a *region of opportunity* that could potentially lead to new codes surpassing the performance of the recent quantum Low Density Parity Check (LDPC) codes found in (Bravyi et al., 2024). Contrary to their strategy, the kind of codes that we can find are not limited by a particular anstaz beyond being CSS. We emphasize that not only would we discover the code, but a hardware-efficient encoding circuit would also be simultaneously discovered, which is something currently lacking.

Exploring this region of opportunity is an equally exciting and challenging endeavor, so we leave it for future work. In this work, we will focus on smaller codes of up to distance-5.

5. Reinforcement Learning Results

In this section we show quantum codes that we have found with our RL approach. We have restricted the search to weakly self-dual codes (meaning the Hadamard layer contains num$(H) = (n-k)/2$ gates) for concreteness. We have considered code distances from 3 to 5 and three sets of qubit connectivities: nearest-neighbor, next-to-nearest-neighbor and all-to-all. For each code distance we have chosen three different target code parameters $[[n, k, d]]$. We have always placed the initial Hadamard gates in alternating qubit indices. Hyperparameters are explained in Appendix B together with a range of hyperparameter values that we have used for this work. All our experiments are open-sourced and reproducible through our github repository (qdx). For compactness, we summarize our results in the following Table 1.

Code parameters	NN-1	NN-2	All-to-All
[[7,1,3]]	3 + 15	3 + 10	3 + 9
[[9,1,3]]	4 + 19	4 + 11	4 + 10
[[11,3,3]]	4 + 38	4 + 23	4 + 16
[[13,1,4]]	6 + 38	6 + 26	6 + 18
[[16,2,4]]	7 + 65	7 + 37	7 + 24
[[20,6,4]]	7 + 141	7 + 75	7 + 37
[[19,1,5]]	9 + 98	9 + 61	9 + 31
[[22,2,5]]	10 + 123	10 + 78	10 + 37
[[25,1,5]]	12 + 117	12 + 70	12 + 37

Table 1. Discovered $[[n, k, d]]$ codes and encoding circuits in different qubit connectivities: nearest-neighbor (NN-1), next-to-nearest-neighbor (NN-2) and all-to-all. We break the encoding circuit into Hadamard gates + CNOT gates. Here we report the minimal circuit size found across 8 independently trained agents (16 for $d = 5$).

5.1. Distance 3 codes

The smallest $d = 3$ CSS code is Steane code (Steane, 1996), with code parameters $[[7, 1, 3]]$. It is a self-dual code (meaning that $G_X = G_Z$) and thus requires an initial Hadamard layer consisting of num$(H) = (7-1)/2 = 4$. With our approach we are able to rediscover Steane's code and provide encoding circuits in the three qubit connectivities that we have considered. Each of them were found in around 20 seconds.

The smallest $d = 3$ surface code (Kitaev, 1997) has parameters $[[9, 1, 3]]$, so we target these code parameters next. The process is again successful using between 20 seconds to 1 minute for each of the three connectivities.

Finally, we show an example with $k > 1$. By increasing the physical qubit count by 2 we can encode 3 logical qubits, i.e. we consider $[[11, 3, 3]]$ last. There is a noticeable increase in the number of gates that are needed in this case with respect to the previous two, yet our approach is again successful. The time needed now is approximately 1 minute, except for the nearest-neighbor connectivity, where finding a solution is more challenging due to the larger circuit sizes needed and which takes close to 5 minutes.

We remark that the runtimes reported throughout the paper correspond to training all 8 agents in parallel on a single Quadro RTX 6000 GPU.

5.2. Distance 4 codes

The code parameters that we consider in this part are $[[13, 1, 4]]$, $[[16, 2, 4]]$ and $[[20, 6, 4]]$. We have chosen these to show variety in different numbers of logical qubits and closeness to the $d = 4$ surface code (with parameters $[[16, 1, 4]]$). In particular, we note how the number of gates needed to encode $[[20, 6, 4]]$ is much larger than the others. This is due to wanting to encode many more logical qubits than in the other examples.

A useful training strategy that we adopt here (and also for larger code distances) is doing an initial pretraining with $d = 3$, keep the weights of the neural networks and then train with $d = 4$. This strategy akin to curricular learning leads to more successful learning runs. In particular, we observe that the first few CNOT gates that the agent uses in the $d = 3$ task are sometimes kept when solving the $d = 4$ task (more prominently in nearest-neighbor connectivity), thus making this transfer learning quite effective. An example of this behavior is shown for $[[13, 1, 4]]$ in Fig. 2(a).

The compute time needed for solving these tasks are now in the tens of minutes timescale rather than the tens of seconds needed for $d = 3$. Further, we also observe that, from the three connectivities considered, the next-to-nearest-neighbor (NN-2 in Table 1) is the most effective at solving this task. The other two connectivities come with two different challenges: nearest-neighbor (NN-1 in Table 1) requires longer sequences of gates (longer trajectories); agents with all-to-all connectivity have access to more actions ($O(n^2)$) and thus require more timesteps to find a solution. In addi-

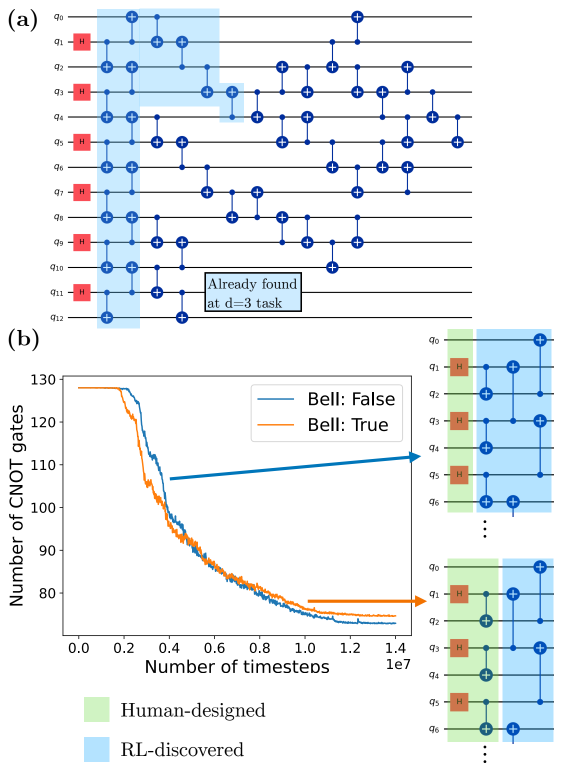


Figure 2. CSS code and encoding discovery. **(a)** Example of curricular learning in $[[13, 1, 4]]$ with nearest-neighbor connectivity. Before targetting $d = 4$, an initial pretraining with $d = 3$ is done. The agent discovers the subcircuit highlighted in blue that is then reused to encode $d = 4$. **(b)** Bell condition in $[[25, 1, 5]]$ with next-to-nearest-neighbor connectivity. We consider starting from just the initial Hadamard layer (Bell: False) or also providing the first layer of CNOTs to start from neighboring Bell pairs (Bell: True). The learning speeds up on average by providing the initial Bell pairs at the expenses of converging to longer encoding sequences than by not using them. The results are averaged over 8 independently trained agents.

tion, transfer learning from a smaller code distance scenario is not as effective for the latter.

5.3. Distance 5 codes

The largest code distance that we have tested with this approach is $d = 5$. Here the challenges encountered in the previous $d = 4$ experiments persist and even accentuate. Nevertheless, these are solved by the same curricular learning strategy presented above and by scaling both the number of parameters of the neural networks and the number of timesteps during which we train. For these reasons, the tasks are now solved in 1-3 hours.

The codes that we target in this last section are $[[19, 1, 5]]$, $[[22, 2, 5]]$ and $[[25, 1, 5]]$. The last one is compatible with the $d = 5$ surface code.

An interesting persistent strategy that we have noticed is that the agent first builds Bell pairs between adjacent qubits (which are $[[2, 0, 2]]$ codes) and then entangles these pairs with each other to gradually build up a $d = 5$ code. We thus consider an additional scenario where *we* initialize the circuit with neighboring Bell pairs and ask the agent to complete the encoding circuit. Correspondingly, in this last part we train 16 agents: 8 with initial Bell pairs provided and 8 agents without. We show a comparison of these two cases on $[[25, 1, 5]]$ with next-to-nearest-neighbor connectivity in Fig. 2(b). The Bell strategy leads to faster convergence on average, and the agents that do not use it still learn to prepare Bell pairs in the first few steps. However, using the Bell strategy seems to lead to slightly longer encoding circuits.

We remark that these code parameters are by no means the upper limit of what is possible with our strategy. However, we defer the exploration of systematic and effective scaling strategies to future work.

6. Conclusions and Outlook

We have presented an efficient RL framework that is able to simultaneously discover QEC codes and their encoding circuits from scratch, given a qubit connectivity, gate set, and error operators. We have shown how to restrict the search of codes to CSS codes, which are of interest for practical applications. We have been able to discover codes and circuits up to 25 physical qubits and code distance 5, while presenting a roadmap to scale this approach much further. This is thanks to our formulation in terms of stabilizers, which serve both as compact input to the agent as well as the basis for rapid Clifford simulations, which we implemented in a vectorized fashion using a modern machine-learning framework.

One of the limits of our approach is GPU memory. However, this could be circumvented through different means. One possibility could be to use a stochastic version of our reward, where only a subset of the error operators are sampled at each timestep. On the other hand, it is always possible to trade performance by memory load by e.g. doing some computations on CPUs. However, the tendency to train very large AI models is thrusting both the development of novel hardware with increased memory capabilities and the integration of distributed computing options in modern machine learning libraries. These developments makes us envision scenarios where the framework presented in this work could be scaled up straightforwardly to multiple GPU machines. This makes us optimistic about the future of AI-discovered QEC in the very near future.

References

QDX: AI-discovery of QEC codes with JAX. https://github.com/jolle-ag/qdx.

Aaronson, S. and Gottesman, D. Improved simulation of stabilizer circuits. *Phys. Rev. A*, 2004. doi: 10.1103/physreva.70.052328. URL https://doi.org/10.1103%2Fphysreva.70.052328.

Acharya, R., Aleiner, I., Allen, R., Andersen, T. I., Ansmann, M., Arute, F., Arya, K., Asfaw, A., Atalaya, J., Babbush, R., et al. Suppressing quantum errors by scaling a surface code logical qubit. *Nature*, 614(7949):676–681, February 2023. doi: 10.1038/s41586-022-05434-1. URL https://doi.org/10.1038/s41586-022-05434-1.

Andreasson, P., Johansson, J., Liljestrand, S., and Granath, M. Quantum error correction for the toric code using deep reinforcement learning. *Quantum*, 3:183, 2019.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., George Necula, A. P., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Bravyi, S., Cross, A. W., Gambetta, J. M., Maslov, D., Rall, P., and Yoder, T. J. High-threshold and low-overhead fault-tolerant quantum memory. *Nature*, 627(8005):778–782, 2024.

Calderbank, A. R. and Shor, P. W. Good quantum error-correcting codes exist. *Phys. Rev. A*, 54:1098–1105, Aug 1996. doi: 10.1103/PhysRevA.54.1098. URL https://link.aps.org/doi/10.1103/PhysRevA.54.1098.

Cao, C. and Lackey, B. Quantum lego: Building quantum error correction codes from tensor networks. *PRX Quantum*, 3:020332, May 2022. doi: 10.1103/PRXQuantum.3.020332. URL https://link.aps.org/doi/10.1103/PRXQuantum.3.020332.

Cao, C., Zhang, C., Wu, Z., Grassl, M., and Zeng, B. Quantum variational learning for quantum error-correcting codes. *Quantum*, 2022. doi: 10.22331/q-2022-10-06-828. URL https://doi.org/10.22331%2Fq-2022-10-06-828.

Colomer, L. D., Skotiniotis, M., and Muñoz-Tapia, R. Reinforcement learning for optimal error correction of toric codes. *Physics Letters A*, 384(17):126353, 2020.

Cong, I., Levine, H., Keesling, A., Bluvstein, D., Wang, S.-T., and Lukin, M. D. Hardware-efficient, fault-tolerant quantum computation with rydberg atoms. *Physical Review X*, 12(2):021049, 2022.

Fitzek, D., Eliasson, M., Kockum, A. F., and Granath, M. Deep q-learning decoder for depolarizing noise on the toric code. *Physical Review Research*, 2(2):023230, 2020.

Fösel, T., Tighineanu, P., Weiss, T., and Marquardt, F. Reinforcement learning with neural networks for quantum feedback. *Phys. Rev. X*, 2018. doi: 10.1103/PhysRevX.8.031084. URL https://link.aps.org/doi/10.1103/PhysRevX.8.031084.

Gidney, C. Stim: a fast stabilizer circuit simulator. *Quantum*, 5:497, July 2021. ISSN 2521-327X. doi: 10.22331/q-2021-07-06-497. URL https://doi.org/10.22331/q-2021-07-06-497.

Girvin, S. M. Introduction to quantum error correction and fault tolerance. *SciPost Physics Lecture Notes*, 2023. doi: 10.21468/scipostphyslectnotes.70. URL https://doi.org/10.21468%2Fscipostphyslectnotes.70.

Gottesman, D. Stabilizer codes and quantum error correction, 1997.

Inguscio, M., Ketterle, W., and Salomon, C. *Proceedings of the International School of Physics" Enrico Fermi."*, volume 164. IOS press, 2007.

Kitaev, A. Y. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys*, 52(6):1191, 1997.

Knill, E. and Laflamme, R. Theory of quantum error-correcting codes. *Phys. Rev. A*, 1997. doi: 10.1103/PhysRevA.55.900. URL https://link.aps.org/doi/10.1103/PhysRevA.55.900.

Konda, V. and Tsitsiklis, J. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

Krinner, S., Lacroix, N., Remm, A., Di Paolo, A., Genois, E., Leroux, C., Hellings, C., Lazar, S., Swiadek, F., Herrmann, J., et al. Realizing repeated quantum error correction in a distance-three surface code. *Nature*, 605(7911):669–674, 2022.

Lu, C., Kuba, J., Letcher, A., Metz, L., de Witt, C. S., and Foerster, J. Discovered policy optimisation. *Advances in Neural Information Processing Systems*, pp. 16455–16468, 2022.

Mauron, C., Farrelly, T., and Stace, T. M. Optimization of tensor network codes with reinforcement learning. *arXiv:2305.11470*, 2023.

Metz, F. and Bukov, M. Self-correcting quantum many-body control using reinforcement learning with tensor networks. *Nature Machine Intelligence*, 5(7):780–791, 2023.

Nautrup, H. P., Delfosse, N., Dunjko, V., Briegel, H. J., and Friis, N. Optimizing quantum error correction codes with reinforcement learning. *Quantum*, 2019. doi: 10.22331/q-2019-12-16-215. URL https://doi.org/10.22331%2Fq-2019-12-16-215.

Olle, J., Zen, R., Puviani, M., and Marquardt, F. Simultaneous discovery of quantum error correction codes and encoders with a noise-aware reinforcement learning agent. *arXiv preprint arXiv:2311.04750*, 2024.

Postler, L., Heuβen, S., Pogorelov, I., Rispler, M., Feldker, T., Meth, M., Marciniak, C. D., Stricker, R., Ringbauer, M., Blatt, R., et al. Demonstration of fault-tolerant universal quantum gate operations. *Nature*, 605(7911):675–680, 2022.

Ryan-Anderson, C., Bohnet, J. G., Lee, K., Gresh, D., Hankin, A., Gaebler, J., Francois, D., Chernoguzov, A., Lucchetti, D., Brown, N. C., et al. Realization of real-time fault-tolerant quantum error correction. *Physical Review X*, 11(4):041058, 2021.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017.

Sivak, V., Eickbusch, A., Royer, B., Singh, S., Tsioutsios, I., Ganjam, S., Miano, A., Brock, B., Ding, A., Frunzio, L., et al. Real-time quantum error correction beyond break-even. *Nature*, 616(7955):50–55, 2023.

Steane, A. M. Simple quantum error-correcting codes. *Phys. Rev. A*, 54:4741–4751, Dec 1996. doi: 10.1103/PhysRevA.54.4741. URL https://link.aps.org/doi/10.1103/PhysRevA.54.4741.

Su, V. P., Cao, C., Hu, H.-Y., Yanay, Y., Tahan, C., and Swingle, B. Discovery of optimal quantum error correcting codes via reinforcement learning, 2023.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

Sweke, R., Kesselring, M. S., van Nieuwenburg, E. P., and Eisert, J. Reinforcement learning decoders for fault-tolerant quantum computation. *Machine Learning: Science and Technology*, 2(2):025005, 2020.

Wang, H., Fu, T., Du, Y., Gao, W., Huang, K., Liu, Z., Chandak, P., Liu, S., Van Katwyk, P., Deac, A., et al. Scientific discovery in the age of artificial intelligence. *Nature*, 620(7972):47–60, 2023.

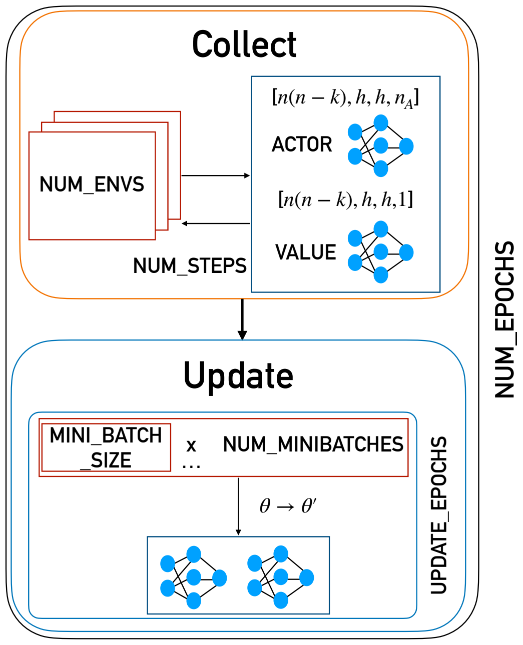


Figure 3. Structure of the PPO algorithm used in this work, focusing on its structural and operational aspects. In the Collect phase, the agent interacts with the environments to extract triples (observation, action, reward) that are then used in the Update phase to update the parameters of the neural networks via gradient descent.

A. Clifford Gates

In this Appendix we explicitly define the Clifford gate generators: the Hadamard gate H, the phase gate S and the CNOT gate. There are different equivalent ways to define them, but here we choose to give their definitions in terms of how they transform Pauli strings (this corresponds to the Heisenberg picture definition).

Explicitly, the Hadamard gate rule is,

$$H \cdot X \cdot H^\dagger = Z\,, \quad H \cdot Z \cdot H^\dagger = X\,, \tag{11}$$

where $\cdot$ denotes matrix multiplication and $\dagger$ denotes the self-adjoint operation.

The phase gate rule is,

$$S \cdot X \cdot S^\dagger = Y\,, \quad S \cdot Z \cdot S^\dagger = Z\,. \tag{12}$$

Finally, the CNOT rule is,

$$\mathrm{CNOT}(0,1) \cdot XI \cdot \mathrm{CNOT}(0,1)^\dagger = XX\,, \quad \mathrm{CNOT}(0,1) \cdot IX \cdot \mathrm{CNOT}(0,1)^\dagger = IX\,, \tag{13}$$

$$\mathrm{CNOT}(0,1) \cdot ZI \cdot \mathrm{CNOT}(0,1)^\dagger = ZI\,, \quad \mathrm{CNOT}(0,1) \cdot IZ \cdot \mathrm{CNOT}(0,1)^\dagger = ZZ\,, \tag{14}$$

$$\mathrm{CNOT}(1,0) \cdot XI \cdot \mathrm{CNOT}(1,0)^\dagger = XI\,, \quad \mathrm{CNOT}(1,0) \cdot IX \cdot \mathrm{CNOT}(1,0)^\dagger = XX\,, \tag{15}$$

$$\mathrm{CNOT}(1,0) \cdot ZI \cdot \mathrm{CNOT}(1,0)^\dagger = ZZ\,, \quad \mathrm{CNOT}(1,0) \cdot IZ \cdot \mathrm{CNOT}(1,0)^\dagger = IZ\,. \tag{16}$$

B. Proximal Policy Optimization

We use the PPO implementation of (Lu et al., 2022), which we break down in more detail here (see also Fig. 3 and Table 2 for a list of hyperparameters). In our implementation, the RL environment is vectorized, meaning that the agent interacts with multiple different quantum circuits at the same time. The hyperparameter that determines this number of RL environments is called `NUM_ENVS`. The learning algorithm consists of two processes: collect and update. During collection, the agent interacts with the environments and a total of `NUM_STEPS` sequences of (observation, action, reward) are collected per environment. Following the collection, the update process begins. Here, we have a total of `NUM_ENVS * NUM_STEPS` individual steps that are shuffled and reshaped into `NUM_MINIBATCHES` minibatches (each of size `NUM_ENVS * NUM_STEPS // NUM_MINIBATCHES`). These are used for updating the weights of the neural networks through gradient descent, which happens a number `UPDATE_EPOCHS` times during every update process. The whole collection-update cycle gets repeated `NUM_EPOCHS` times.

The neural networks that we have chosen are standard feedforward fully-connected neural networks with ReLU activation functions and with identical architectures for both the actor and value networks, except for the output layer. In particular, they both consist of an input layer of size $n(n-k)$ given by the observation from the environment and consisting of all the flattened code generators. This is followed by two hidden layers of size h (we have experimented with sizes 64 to 400) and an output layer of size n_A (number of actions) in the case of the actor network and of size 1 for the value network (see Fig. 3). The number of actions n_A is determined by the number of physical qubits and qubit connectivity.

Other hyperparameters that participate in the PPO implementation which we include for completeness (but that we refer to (Schulman et al., 2017) for further explanations) are the discount factor γ, the generalized advantage estimator (GAE) parameter λ, the actor loss clipping parameter ε, the entropy coefficient and the value function (VF) coefficient (see Table 2 for typical values that we have found to work well).

Regarding the optimizer itself, we use ADAM with a clipping in the norm of the gradient (`MAX_GRAD_NORM`) and some initial learning rate (`LR`) that gets annealed (`ANNEAL_LR`) using a linear schedule as the training evolves, see Table 2 for specific numerical values of these hyperparameters.

Hyperparameter	**Value**
`LR`	5×10^{-4} - 1×10^{-3}
`NUM_ENVS`	16-128
`NUM_STEPS`	32-190
`NUM_EPOCHS`	1000-10000
`UPDATE_EPOCHS`	4
`NUM_MINIBATCHES`	4-16
`GAMMA`	0.99
`GAE_LAMBDA`	0.95
`CLIP_EPS`	0.2
`ENT_COEF`	0.01-0.05
`VF_COEF`	0.5
`MAX_GRAD_NORM`	0.5
`ANNEAL_LR`	True

Table 2. Hyperparameters that were used during training with some typical range of values that we have seen to lead to good performance (see text for a description of each hyperparameter). We also include in our repository (qdx) specific values for all experiments reported in Table 1.

C. Circuit structure of CSS codes

Here we give a proof of the claim that codes resulting from circuits with an initial block of Hadamard gates on a subset of the qubits and followed by CNOT gates thereafter can only be CSS.

Let us label physical qubits with index $1 \le q \le n$ and target a CSS code with parameters $[[n,k,d]]$. Let's assume for simplicity that the initial block of Hadamard gates is applied to qubits $k+1,\dots,k+n_H$, with $n_H < n-k$. The initial tableau of the would-be code reads

$$\begin{aligned} g_1 &= X_{k+1} \ , \\ g_2 &= X_{k+2} \ , \\ &\dots \\ g_{n_H} &= X_{k+n_H} \ , \\ g_{n_H+1} &= Z_{k+n_H+1} \ , \\ &\dots \\ g_{n-k} &= Z_n \ . \end{aligned} \tag{17}$$

From this moment forward, only CNOT gates are allowed. Let's start by considering what is the effect of a CNOT gate with control qubit inside the H-block, i.e. control $\in \{k+1, \dots, k+n_H\}$. For whatever target qubit, what such a CNOT does is populate the target position of the corresponding stabilizer g_{control} with an X. Subsequent CNOT gates affecting those positions, either as control or target qubits, will either introduce additional X's or simply do nothing. Since $X^2 = I$, the stabilizers $g_1, g_2, \dots g_{n_H}$ will only ever contain either X's or 1's. Similarly, the effect of CNOTs on stabilizers $g_{n_H+1}, \dots, g_{n-k}$ is simply populating them with Z's or I's. Since the set of stabilizer generators can be clearly separated into a subset built with only X's and I's and another one with only Z's and I's, such a tableau describes a CSS code.

D. Softness parameter

The second KL condition Eq. (3) requires checking whether any error operator $E_\mu \in S_{\mathcal{C}}$. In principle, the full stabilizer group of 2^{n-k} elements must be built at every time step of our simulations. In practice, not many error operators end up being in $S_{\mathcal{C}}$, which we leverage by introducing an integer *softness* parameter s, such that only a subgroup of $S_{\mathcal{C}}$ is built. More precisely, $s = 0$ means that this subgroup is empty, $s = 1$ means taking only the generators g_i as the subgroup, $s = 2$ means taking the generators g_i and all pairwise combinations of generators $g_i g_j$, and so on for larger s.

Since X-type generators are independent from Z-type generators, so are their Stabilizer groups. Thus, at every timestep two subgroups are generated: one for S_X and another one for S_Z.