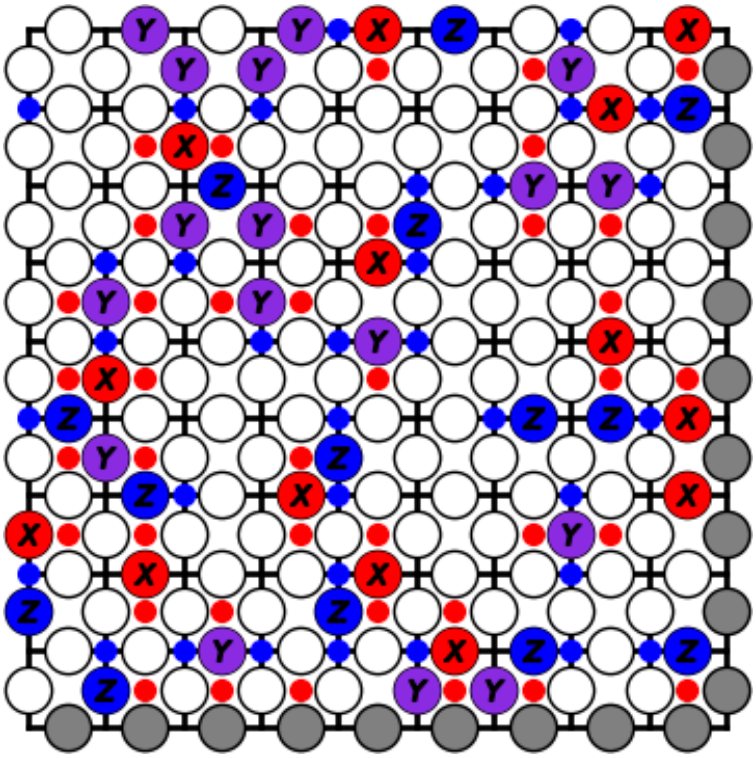
CHALMERS
UNIVERSITY OF TECHNOLOGY



Distributed Training for Deep Reinforcement Learning Decoders on the Toric Code

Master's thesis in Master Programme Algorithms, Languages and Logic

ADAM OLSSON & GABRIEL LINDEBY

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020:NN

Distributed Training for Deep Reinforcement Learning Decoders on the Toric Code

ADAM OLSSON
GABRIEL LINDEBY

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2020

Distributed Training for Deep Reinforcement Learning Decoders on the Toric Code
Adam Olsson & Gabriel Lindeby

Supervisor & Examiner: Mats Granath, Department of Physics

Master's Thesis 2020:NN
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A syndrome with underlying errors on a toric code of size 9.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Distributed Training for Deep Reinforcement Learning Decoders on the Toric Code
Adam Olsson & Gabriel Lindeby
Department of Physics
Chalmers University of Technology

Abstract

We distribute the training of a deep reinforcement learning-based decoder on the toric code developed by Fitzek *et al.* [9]. Reinforcement learning agents asynchronously step through multiple environments in parallel and store transitions in a prioritized experience replay buffer. A separate process samples the replay buffer and performs backpropagation on a policy network. With this setup, we managed to improve wall-clock training times with a factor ≥ 12 for toric code sizes of $d = 5$ and $d = 7$. For $d = 9$, we were unable to reach optimal performance but improved the decoder's success rate using a network with a parameter reduction of factor 20. We argue that these results pave the way for optimal decoders, correcting errors close to what is theoretically possible, based on reinforcement learning for toric code sizes ≥ 9. The complete code for the training and toric code environment can be found in the repository `https://github.com/Lindeby/toric-RL-decoder` and `https://github.com/Lindeby/gym_ToricCode`.

Keywords: Deep Reinforcement Learning, Distributed, Toric Code, Quantum Error Correction, Ape-X.

Acknowledgements

We would like to thank our supervisor Mats Granath and David Fitzek who always was available to discuss our thoughts and provide great idéas. Additionally, we would like to thank Hugo Strand for providing insights into how to efficiently implement the codebase. Finally, all computations were performed on the Vera cluster at Chalmers Centre for Computational Science and Engineering (C3SE).

Adam Olsson & Gabriel Lindeby, Gothenburg, June 2020

Contents

List of Figures

List of Tables

1
Introduction

The quantum computer's counterpart to bits in a classical computer are called qubits. Qubits are inherently fragile and susceptible to noise[26]. The qubits' fragile nature result in unavoidable qubit errors during quantum computations, making the results unusable. Surface codes[7, 14, 11, 28] are a structure of physical qubits located on a two-dimensional grid that provides topological protection to a logical qubit. Only chains of errors from end to opposite end on the grid cause logical bit-flips. Therefore, increasing the grid size results in greater protection. However, even though surface codes provide protection against disturbances, these codes still need to be monitored and corrected to prevent logical bit-flips. The difficulty in correcting these errors lie in that they can not be measured directly. Instead, quantum error correction algorithms observe the results of parity checks over the grid and incorporate statistics during the correction process.

Deep Reinforcement Learning has recently been suggested as a tool for quantum error correction because of its ability to learn advanced control policies from sensory input[17]. An agent learns to manipulate a quantum device and apply quantum logical operations to negate the errors. These Reinforcement Learning algorithms have proven to be more accurate than the benchmark algorithm Minimum Weight Perfect Matching[8, 10, 5] for error correction on surface codes[9, 6]. However, to learn these advanced control policies, large parts of the state-space need to be explored. Deep learning frameworks such as Tensorflow[2] and Pytorch[21] support distributed training methods that allow for faster exploration of the state-space. Previous work state that for a surface code of size $d = 9$, the correction algorithm was unable to converge to its theoretical optimal performance and suggest that the training method is inefficient[9]. Various optimization techniques to combat inefficient training by making more use of data and explore the state-space faster have been developed[13, 4] but never applied in a quantum error correction context. The main interest in applying these methods to the quantum error correction context lies not only in a faster convergence rate, but also if convergence can be reached on larger surface codes, $d \geq 9$. The latter would enable more robust quantum computations.

This Master's thesis is a continuation of the worked developed by Fitzek *et al.* [9]. Here, the quantum error correction algorithm's training process is improved by allowing for a distributed training approach. The work is inspired by the Ape-X[13] framework with minor modifications to better fit the quantum error correction context. The aim is to expand the training framework for quantum error correc-

tion algorithms based on deep reinforcement learning with a more efficient training method. Current work use a sequential training approach where generation of data and training are interweaved[9, 3]. The interweaving results in the computational resources not being used to its fullest potential because of the differences in computational intensity. By separating the concerns of data generation and training, we find that the wall-clock time for training can be decreased by a minimum of factor 12 without sacrificing error correction performance. Finally, the work here has been limited to only improve the training method. We do not consider means to improve the overall error correction performance, such as new neural network architectures or reward schemes. Instead, any increase in the correction performance should come through a more efficient exploration of the state-space.

2
Background

2.1 Toric Code Decoder

Error correction on qubits uses some concepts that are similar, if not the same, as some concepts in error correction on classical bits. These concepts are easier to grasp in the context of classical error correction. Therefore, the necessary concepts will first be explained using error correction on classical bits followed by the toric code for quantum error correction.

2.1.1 Error Corrections and Detection on Classical Bits

The smallest possible code that can correct errors in a classical computer is the three-bit code where three physical bits are encoded as a single logical bit.

$$\text{Logic}(0) = 000$$
$$\text{Logic}(1) = 111$$

The three-bit code is robust enough to withstand a single bit error by letting the majority value of the physical bits determine the logical bits value. As an example, consider the following two recoverable single bit errors:

$$\text{Correction}(100) = 000 = \text{Logic}(0)$$
$$\text{Correction}(101) = 111 = \text{Logic}(1)$$

Should a majority of the bits have an error, the value of the logical bits can no longer be recovered, which causes a bit-flip. The recoverable number of errors can be generalized to any code length. A code length of n can recover from $(n-1)/2$ errors. This metric is also known as *code-distance*. Furthermore, should the three-bit code be transmitted over a network, a receiver needs to be able to detect if any errors occurred during the transmission. A method for detecting errors is by adding a bit to the far left of the original data so that the total number of 1's transmitted is even:

$$\text{Original data} = 010 \rightarrow \text{Transmitted data} = 1010$$
$$\text{Original data} = 110 \rightarrow \text{Transmitted data} = 0110$$

The receiver then checks if the number of 1's in the data is even or odd and reports a successful or failed transmission respectively. This type of check is known as *parity*

check and it is a method for detecting errors. Error correcting codes on qubits, however, works in quite different ways but the notion of code-distance and parity checks provides a foundation to understand them.

2.1.2 Error Corrections and Detection on Qubits

Just like a classical bit, a qubit can only measure a logical state value of 0 or 1. Unlike a classical bit, qubits have a superposition of 0 and 1, where the state is neither 0 or 1 but a combination of both. A superposition can be held until a qubit's logical state is measured, which causes a collapse into a logical value of 0 or 1 [20]. This quantum phenomena cause difficulties in the quantum error correction process because it is not possible to measure the error directly. Doing so would destroy the superposition of the qubit and ruin the computation.

Kitaev's toric code[7, 14, 11, 28] is a surface code that provides topological protection to logical qubits. A square lattice with periodic boundaries represents a single logical qubit and embed physical qubits along its edges. The number of embedded physical qubits depends on the size of the lattice. A lattice of width d has $2d^2$ physical qubits embedded into the logical qubit, as shown in Figure 2.1. The logical qubit's state is protected by local disturbances because operations on the logical qubit require global changes to the physical qubits.

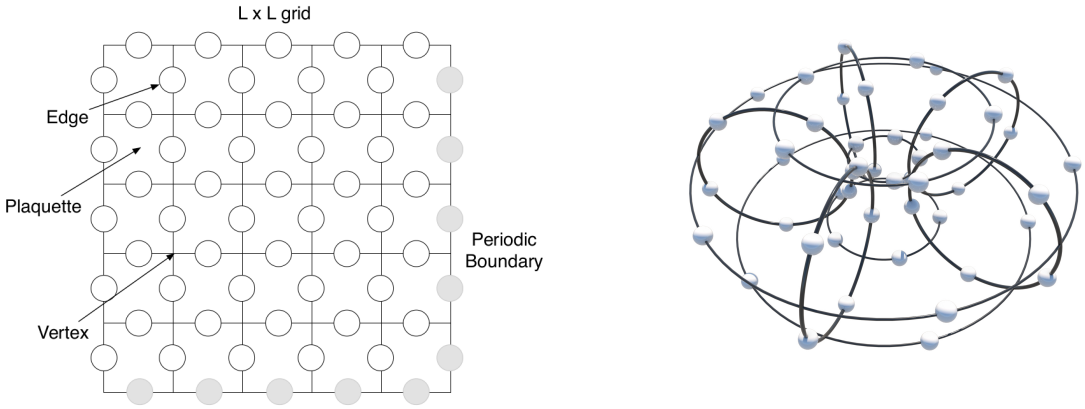


Figure 2.1: (Left): The toric code uses a $d \times d$ (here 5×5) lattice with edges, plaquettes and vertices. On each edge, a circle represents the embedded physical qubit. (Right): 3D Visualization of the same 5×5 toric-code showing the periodic boundaries

Errors on physical qubits come in three forms; bit-flip, phase-flip, and a combination of the two which are represented as X, Z, and Y respectively. These errors can not be measured directly because it causes the collapse of the superposition. Instead, parity checks can be performed on groups of physical qubits to provide non-destructive detection of errors. These parity checks are divided into two categories, plaquette and vertex operators. Any parity check which has been violated is referred to as a plaquette or vertex *defect*. As shown in Figure 2.2, X and Z errors produce two

defects on neighboring plaquettes or vertices respectively and Y errors produce both types of defects. Neighboring errors cause chains in the code where only the ends of the chain show a defect. Because of these error chains, the defects do not provide a unique picture of the underlying error since multiple error chains have the same appearing defects. Figure 2.3 display two appearing defects and three examples of possible underlying error chains.

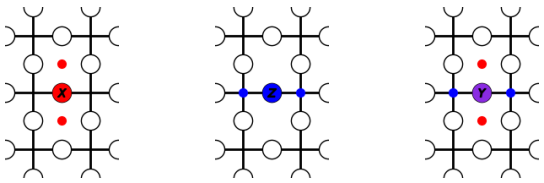


Figure 2.2: Visualization of a X, Z and Y errors and the defects they produce.

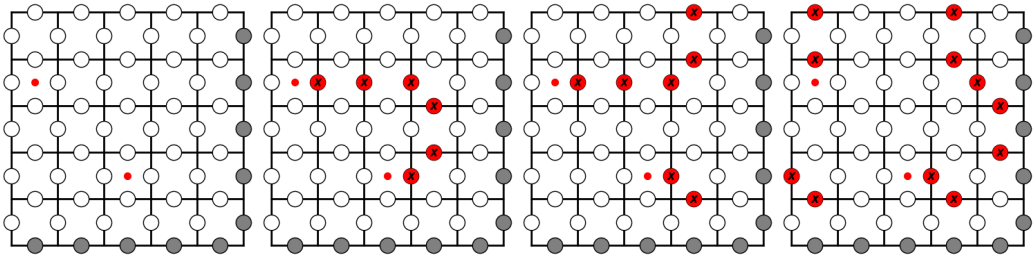


Figure 2.3: Two defects (left) do not uniquely determine the underlying error. Three examples of possible underlying errors that produce the same appearing defects.

Errors are removed by applying the corresponding quantum operator X, Z, or Y to the erroneous physical qubit. The three quantum operators have the same effect as their respective error: an X operator performs a bit flip, Z operator phase-flip, and Y operator a combination of X and Z. However, the quantum correction algorithm, called *decoder*, does not observe the underlying error. Because of the parity checks, the decoder observes the code in terms of its defects, known as *syndrome*. Whilst observing the syndrome and applying various operators, the defects can be seen traversing the code in different directions, disappearing or appearing due to the change in the underlying error. From the perspective of the decoder, a syndrome is corrected by removing all remaining defects. Two defects are removed once they have been moved onto the same plaquette or vertex.

Once a syndrome is corrected, there are three possible outcomes: (1) all underlying errors are removed, (2) the underlying errors have formed a chain that does not span from end to opposite end, referred to as *trivial loop*, (3) the underlying errors have formed a chain that spans from end to opposite end, referred to as *non-trivial loop* 2.4. The former two outcomes are considered a successful error correction where the state is preserved and the non-trivial loop a fail that caused a bit-flip on the logical qubit. The minimum amount of errors needed for a non-trivial loop to form is the code-distance, which here is d. A larger code-distance offers greater protection to the

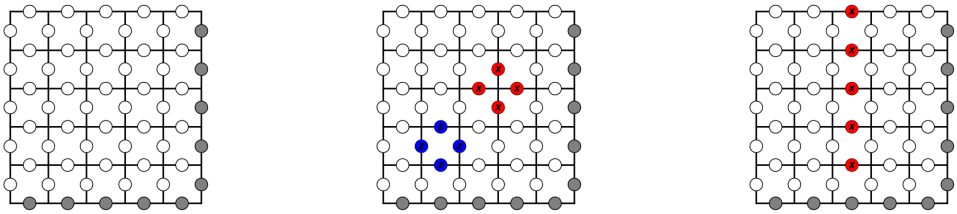


Figure 2.4: Outcome after corrected syndromes. (Left): No error's, (Middle): two trivial loops, (Right): non-trivial loop

logical qubit but also comes with a larger state space. A larger state space means a larger set of possible error combinations that the decoder needs to consider. Because the decoder base its operations on the syndrome, it needs to incorporate statistics into the correction process. A larger code-distance would, therefore, lead to a larger error distribution, which can be difficult to model. However, recently a method for modeling this distribution has emerged using deep reinforcement learning.

2.2 Deep Reinforcement Learning

In reinforcement learning[27] we consider an agent performing actions $a \in A$ in an environment. At every time step t, the agent evaluates its current state s_t and performs an action a_t. After each action, the agent receives a scalar reward r_t from the environment and progress to state s_{t+1}. The goal is to learn a optimal policy π^*, a mapping function from states to probabilities of selecting actions, that maximizes the discounted future reward in a finite Markov Decision Process. The notion of how 'good' an action is in a given state can be measured through *action-value* functions. Action-value functions are defined by the expected value of taking action a at time step t in state s, receiving reward R_t and thereafter following policy π using a discount factor $\gamma \leq 1$:

$$Q_\pi(s,a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^t R_{t+k+1} | S_t = s, A_t = a] \ , \tag{2.1}$$

The optimal action-value function Q^* is defined by the maximal achievable expected reward by following any policy after encountering state s and taking action a. Q^* can be derived from estimating Q_π using iterative updates of a Bellman equation. A Bellman equation expresses relationships between the values of states and successor states. Once Q^* has been estimated, the optimal policy π^* can be derived by taking the maximizing action over each state in Q_π.

In one-step Q-learning[30], Q_π is estimated through the Bellman equation given by Equation 2.2 which in this case also directly approximates Q^*.

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a). \tag{2.2}$$

For each iteration, the algorithm observes a new state s_t, selects action a according to $a_t = \max_a Q(s_t, a)$ using an ϵ-greedy policy and updates action-value function by

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[\ r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]\ . \tag{2.3}$$

where α is the learning rate. The ϵ-greedy policy introduces the notion of exploration by randomly selecting a non-maximizing action with a probability of ϵ. Exploration allows the algorithm to visit new states that could lead to Q^*. However, this gives rise to a dilemma: From any state, we seek to learn action-values conditional on future optimal behavior but need to behave after a non-optimal policy in order to explore all actions. How can we learn about an optimal policy by only taking actions according to a non-optimal policy? A solution to the dilemma would be to separate behavior and learning into two policies: one policy we try to learn, *target policy*, and one policy that creates a behavior, *behaviour policy*. By the separation, the target policy observes and learns from actions taken by a behavior policy which can be exploratory. This type of learning is called *off-policy* learning.

In environments where the state space is relatively small, the action-value function can be a table or a linear function approximator. However, these two methods can quickly become unfeasible when the state space increases. Instead, non-linear function approximators, such as neural networks, have become popular due to their ability to learn advanced policies. Training of neural networks in Q-learning tries to minimize the loss given by

$$L(\theta) = Loss(\ r_t + \gamma \max_a Q(s_{t+1}, a, \theta^-) - Q(s_t, a_t, \theta))\ , \tag{2.4}$$

where θ and θ^- are two sets of network parameters know as policy network and target network respectively. These two networks are fundamentally the same, but the target network is held constant and occasionally updated to match the policy network whereas the policy network is constantly updated through backpropagation. Stability during training is provided by having the slow-moving network θ^- provide target values when using supervised learning[18]. Furthermore, the measure of the difference between the target and policy network is known as Temporal Difference error, *TD-error*, and is an important notion of how 'surprising' the return was. Large TD-errors imply that the estimate by the policy network was far from target value which in turn gives a high loss. Finally, training of neural networks assumes independent data samples which is not the case in reinforcement learning. States are highly dependent on previous states because to reach state s_t, s_{t-1} must always precede. To train a neural network in a reinforcement learning setting these dependencies need to be broken.

2.2.1 Prioritized Experience Replay

To break the dependencies between states, an experience replay buffer[15, 25] store transitions between states in memory. At the core of the prioritized experience replay buffer is a criterion of how important each transition is, known as *priority*. Here, the TD-error is used because higher TD-errors imply that there exists information

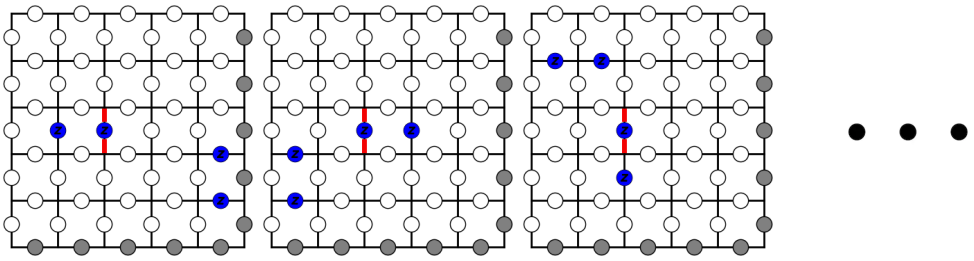


Figure 2.5: Example of an underlying error with a subset of the perspectives. The red line marks the centered qubit.

in the transition that yet has not been learned. Moreover, because the replay buffer contains states from a distribution generated by the behavior policy, they can not directly be used to obtain Q_π. It is important to note that the distribution over the behavior policy is completely different from the target policy we try to learn. A general technique for estimating values in a distribution using values from another distribution is importance sampling[27]. Returns from the environment are weighted according to the relative probability of occurring under the target and behavior policies. The weights are given by

$$w_i = \left(\frac{1}{N} \frac{1}{P(i)} \right)^{\beta} \tag{2.5}$$

where $P(i)$ is the probability of sampling transition i, N the current size of the replay buffer and scaling of how much to compensate β between the two distributions.

2.2.2 Network Representation

Due to the periodic boundaries of the toric code, a syndrome can be represented using an arbitrary qubit in its center [3, 9]. With qubit e_i centered, all other qubits are displayed in their relative positions in the syndrome and defines *perspective* P_i, as shown in Figure 2.5. The set of perspectives for a state s_t is keyed *observation* O_t. An agent will be given the option to apply an X, Z or Y operator to the centered qubit in each of the perspectives in the observation 2.6. The number of perspectives vary with the number of defects. However, observations can be fed as batches of different sizes to the network which allows for a fixed number of output nodes independent of the observation size 2.7. Additionally, the perspectives make the decoder work on only qubits who are adjacent to a defect. This avoids unnecessary computations of action-values for qubits who are not causing the defect.

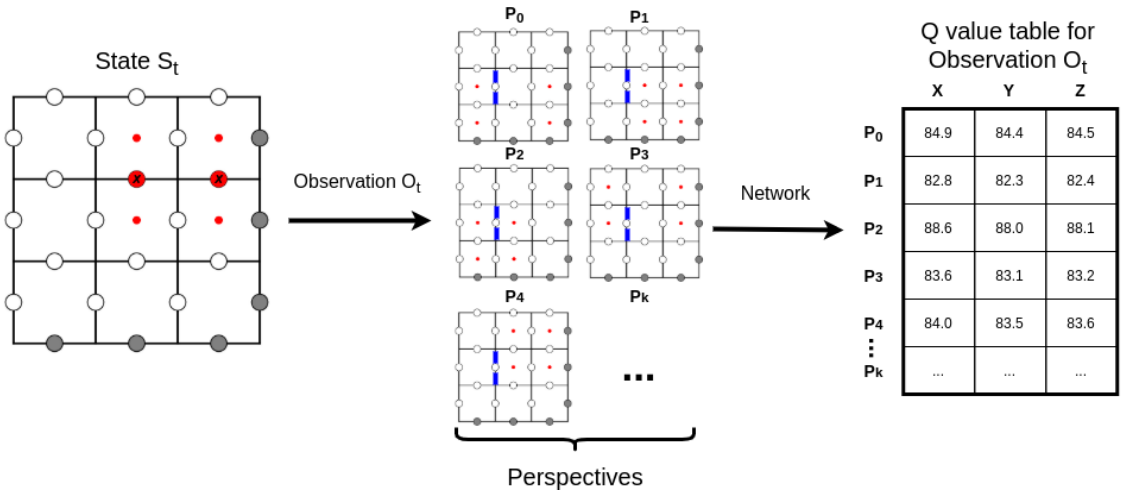


Figure 2.6: All possible perspectives are generated form state S_t, and keyed observation O_t and feed to a neural network. The network outputs a q-value table with one row for each perspective and one column for each operation, X, Y, and Z, where the action is done on the center qubit with blue marked edges. The highest q-value is the best action for state S_t according to the network. The q-values suggested by the network are not fully converged to optimal q-values. However, it is clear that the best action is an X operation in perspective P_2, which gives the highest reward. Note that all perspectives for this state are not included in the figure.

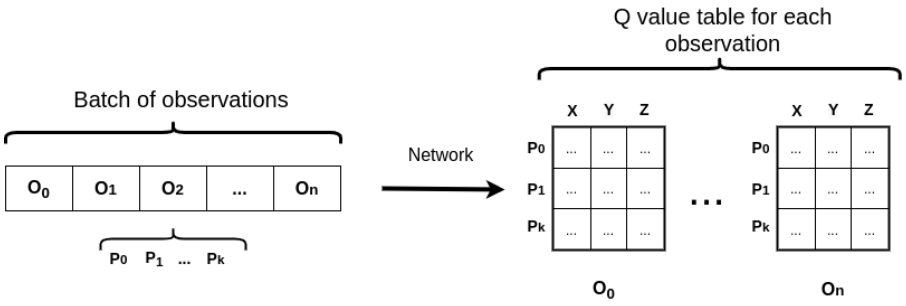


Figure 2.7: Observations can be feed as batches to the network to improve efficiency. The number of perspectives for an observation O_t depends on the underlying syndrome for state S_t and can be different from one observation to another. The network outputs one q-value table for each observation.

3
Methods

The system is built from processes having one out of three distinct roles which all run in parallel on a single machine. In the first process, reinforcement learning agents step through their copy of the environment to generate data, referred to as *acting*. In the second process, a replay memory stores data generated from the actor referred to as *IO*. Finally, the third process trains the policy network using the data stored in the replay memory, referred to as *learning*. Figure 3.1 shows a visual overview of the system. The data flow between processes is created using queues from pythons built-in multiprocessing library and all machine learning aspects are implemented using Pytorch[21]. Additionally, Numba[1] has been used where applicable to compile python code into optimized machine code.

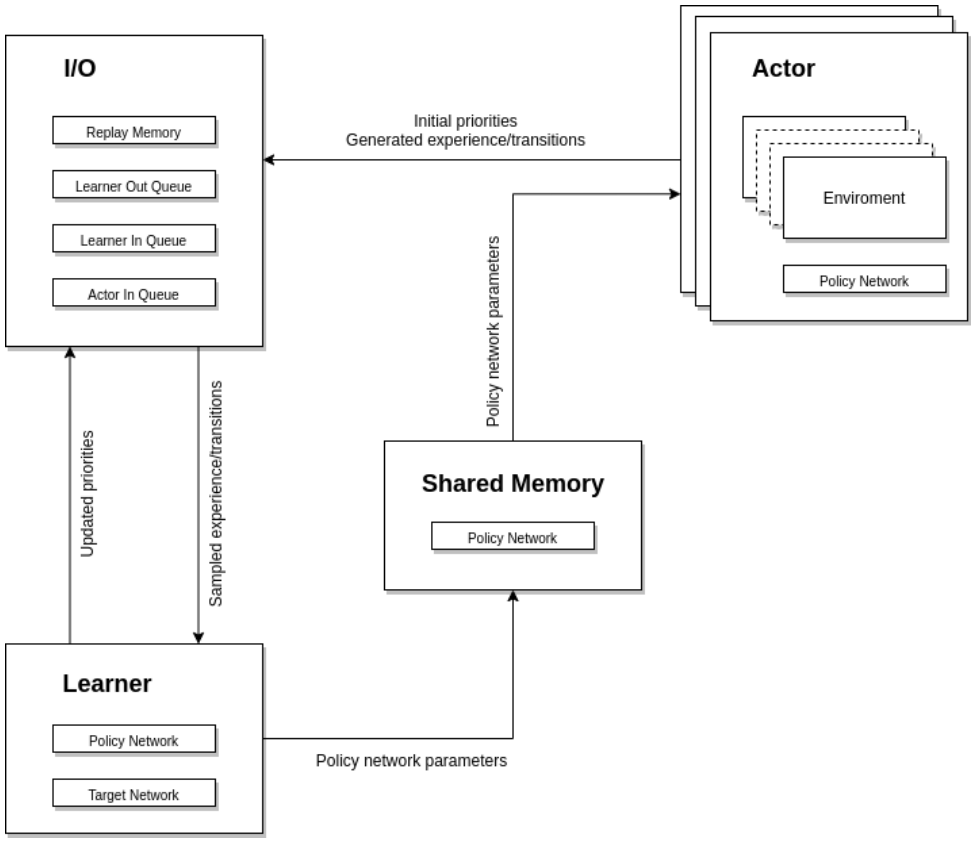


Figure 3.1: An overview of the architecture.

3.1 Actor

Algorithm 1 Actor

$\theta \leftarrow SharedMemory.Read()$
$envs \leftarrow InitializeEnviroments()$
$S_t \leftarrow envs.Reset()$
while True **do**
    $O_t \leftarrow$GenPerspectives(S_t)
    $a_t, q_t \leftarrow SelectActions(\theta, \epsilon, O_t)$ ▷ Select one action/enviroment
    $S_{t+1}, r_t, terminal \leftarrow envs.Step(a_t)$ ▷ Take a step in each enviroment
    $transition \leftarrow (a_t, r_t, S_t, S_{t+1}, terminal)$
    $LocalStorage \leftarrow (S_t, a_t, q_t, r_t, transition)$
    if $LocalStorage.Size() > B$ **then** ▷ Send transitions to I/O
        $priorities \leftarrow ComputePriorities(LocalStorage : a, r, q)$
        $ReplayMemory.Send(priorities, LocalStorage : transition)$
    end if
    if $NewPolicyNetAvaliable()$ **then**
        $\theta \leftarrow SharedMemory.Read()$ ▷ Update policy net
    end if
    $S_t \leftarrow S_{t+1}$
end while

The actor process steps through its copy of the environment. In each iteration, the actor encounters a syndrome in state s_t and generates observation O_t. The observation is forward propagated through the actor's copy of the policy network and returns state-action values for the X, Z, and Y operator on each perspective. From the action-values, an ϵ-greedy policy selects action a_t, which consists of an operator and perspective. The action is performed on the centered qubit of the perspective which results in progression to state s_{t+1} and the agent receiving reward r_t. See Algorithm 1 for a pseudo-code description. The reward scheme is designed to assume that the most likely error chain is the shortest. It is given by

$$r_t = \begin{cases} 100 & \text{if } s_{t+1} \text{ has no defects} \\ E_t - E_{t-1} & \text{otherwise} \end{cases} \tag{3.1}$$

where E_t is the number of defects left in the syndrome by time step t. Four circular local buffers store s_t, a_t, r_t and $Q(s_t, *)$. Once these buffers are full, transition tuples $(s_t, a_t, r_t, s_{t+1}, terminal)$ are formed, where $terminal$ is a boolean variable indicating if s_{t+1} is a terminal state, i.e. no remaining defects. From $Q(s_t, *)$, initial priorities are computed and coupled with respective transition tuple. The transitions and their priorities are then sent to the replay memory in batch. After the data has been sent, the buffers are overwritten and the actor cycle repeated. Note that the local buffers can be increased in size to lower the strain on the communication channel. Establishing communication is a time-consuming process but once a connection has been done, sending large amounts of data is relatively cheap.

Moreover, to further increase the rate at which transitions are generated each actor synchronously performs steps in multiple environments independently. Due to better scaling with larger batch sizes, it is more efficient to run multiple environments in a single actor process compared to a single environment in multiple actor processes. Therefore, whenever the actor extracts q-values, the latest observation from each environment is concatenated into a single batch and then propagated through the network. By doing so, the memory consumption is lowered because the additional networks that would have been needed for each actor process can be avoided.

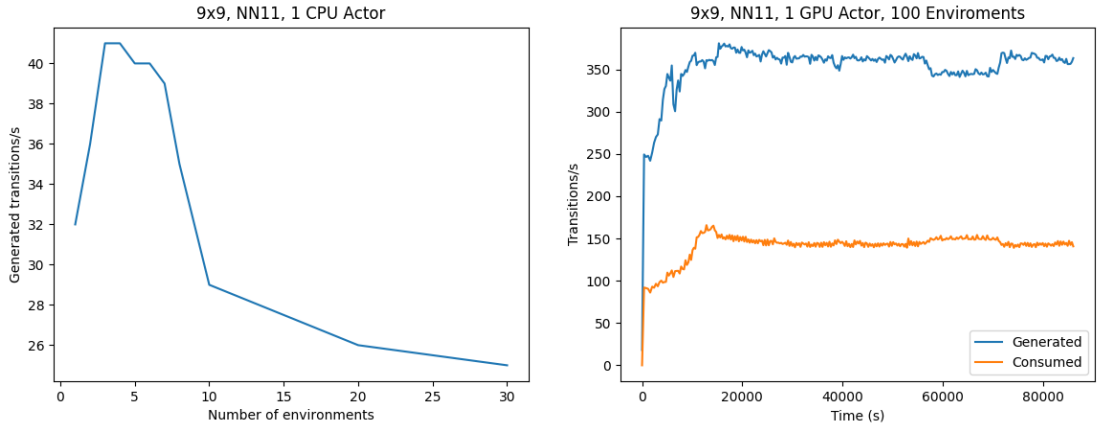


Figure 3.2: On the left, the rate of transitions generated from a single actor on a 16 core CPU for different number of environments. On the right, the rate of transitions generated for a single actor using 100 environments on the GPU (blue). Additionally on the right, the rate of consumed transitions by the learner on the GPU (orange).

Originally, the Ape-X framework ran its actor networks on CPUs. But due to hardware limitations to 16 CPU cores, it was unfeasible to run the actor policy networks on CPUs. Figure 3.2 displays a comparison between transitions generated per second when running the actor policy networks on the CPU and GPU. On the left, the best possible case, i.e. no other processes competing for resources, for a single actor is shown against the number of environments running in parallel. The most amount of transitions that can be generated per second is just above 42 when 4 environments are used. On the right, the worst case, i.e. there exist processes that compete for resources, for a single actor network running on the GPU. Here, the competing process is the learner that performs forward and backpropagations on the policy and target networks. The number of transitions generated per second from the actor on the GPU is stable around 350, using 100 environments in parallel. Letting the actor share resources with the learner results in the most amount of transitions generated per second. Therefore, the actor networks are running on the GPU. Additionally, running the actor on the CPU would not generate enough transitions for the learner to consume. This can be seen by the orange line in the right plot of Figure 3.2.

3.2 IO Buffer

Algorithm 2 I/O

```
ReplayMemory ← InitializeReplayMemory()
while True do
    while notActorInQueue.Empty() do        ▷ Receiveing batch of transitons
        (priorities, transitions) ← ActorInQueue.Get()
        ReplayMemory.Save(priorities, transitions)
    end while
    if LearnerOutQueue.Size() < 10 then
        idx, weights, transitions ← ReplayMemory.Sample()
        LearnerOutQueue.Send((idx, weights, transitions))
    end if
    while notLearnerInQueue.Empty() do      ▷ Receiveing batch of priorities
        idx, priority ← LearnerInQueue.Get()
        ReplayMemory.UpdatePrioritis(idx, priority)
    end while
end while
```

The IO process is an experience replay buffer implemented using a SumTree data structure. Algorithm 2 outlines the process in pseudo-code. Each node in the SumTree contains the sum of both its children's value, making the root node contain the sum over the entire tree. Values in the leaf nodes represent a priority of a single transition. Additionally, the SumTree is also a FIFO queue which allows for old data to be overwritten once the buffer is full. Whenever data is added, the priority of the corresponding leaf node to the new transition is overwritten and the parent nodes recursively updated. Each transition has a probability of being sampled by

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{3.2}$$

where $P(i)$ is the probability of sampling transition i with priority p. The exponent α controls how much prioritization should be used where $\alpha = 0$ corresponds to the uniform sampling probability over the memory. Finally, once the batch of data has been sampled from the replay memory, the corresponding weights are computed in the *Sample*() function using Equation 2.5.

3.3 Learner

The purpose of the learner process is to train the policy network. Algorithm 3 outlines the process in pseudo-code. Upon initialization, the learner creates two networks: the policy network and the target network. A copy of the parameters of the policy network is written to shared memory for actors to access. The policy network, therefore, acts as a behavior policy the actors base its actions upon. In the training loop, batches of transitions $(s_t, a_t, r_t, s_{t+1}, terminal)$ are sampled from the replay

Algorithm 3 Learner

$\theta, \theta^- \leftarrow InitializeNetwork()$
$SharedMemory.Write(\theta)$ ▷ Initial update of shared memory parameters
for $t = 1, 2, \ldots, T$ **do**
    Periodically:
        $\theta^- \leftarrow \theta$ ▷ Update target network parameters
        $SharedMemory.Write(\theta)$ ▷ Update shared memory parameters
    $idx, weights, transitions \leftarrow Replay.Sample()$
    $target = r + \gamma \max(transitions, \theta^-)$
    $policy = \max(transitions, \theta)$
    $loss = L\big(target, policy\big)$
    $\theta \leftarrow UpdateParameters(loss, \theta, weights)$
    $Replay.UpdatePriorities(idx, weights, loss)$ ▷ Send new priorities to I/O
end for

memory. The batch of s_{t+1} is forward propagated through the target network and s_t through the policy network to create the two-dimensional vectors $Q(s_{t+1}, *, \theta^-)$ and $Q(s_t, *, \theta)$. From $Q(s_{t+1}, *, \theta^-)$, the maximum action-value is selected for each sample in the batch to create the one-dimensional vector $\max_a Q(s_{t+1}, a, \theta^-)$ according to Equation 2.4. Likewise, $Q(s_t, a_t, \theta)$ is created by selecting the corresponding action-values in $Q(s_t, *, \theta)$ using the batch of a_t. From $Q(s_{t+1}, a, \theta^-)$ and $Q(s_t, a_t, \theta)$ the loss is computed and multiplied with their respective weights, calculated as in Equation 2.5, to remove any bias. From this product, the network is backpropagated using mini-batch gradient descent. Finally, the priorities of the sampled transitions in the replay memory are updated using computed loss. Periodically, the target network is updated with new parameters from the policy network and at the same time, the parameters in the shared memory are updated.

3.4 Gym Environment

Open AI Gym is a toolkit for the standardized development of reinforcement learning environments that simplifies comparisons between algorithms through a common interface. The interface consists of an initialization function for the setup of a new instance of an environment, a step function that performs an action in the environment and returns the next state, and a reset function that resets the environment. By using the Gym toolkit, the toric code environment used in [9] has been re-implemented to provide a standardized environment for future use. The full code can be found in repository `https://github.com/Lindeby/gym_ToricCode`.

3.5 Training

Network architecture and hyper-parameters were kept constant during the training of toric code grid sizes $d = 5$ and $d = 7$. For a grid of size $d = 9$, the only change was a decreased update rate of the target network. We used mean squared error as a loss

function throughout the work. A complete table outlining the network, called NN11, and hyper-parameters can be found in Appendix A. Furthermore, the exploration parameter ϵ was linearly decreased throughout the training from 1 to a final value selected individually for each environment. The final ϵ for each environment was selected according to

$$\epsilon_i = \beta^{1+\frac{env_i * \alpha}{\max(1, n_{envs}-1)}} \tag{3.3}$$

where subscript i denote the environment number, n_{envs} the number of environments in an actor and α and β two tweakable constants. Moreover, syndromes were generated using a depolarizing noise model where each error has an equal probability p of appearing, $p = p_x = p_y = p_z$, at each physical qubits location. For each new syndrome, p was randomly selected from an interval that grew in size over the training period. Initially, p was selected from the interval $[0.1, 0.1]$ to generate easy syndromes but for each solved syndrome the interval size grew by 0.00005 until reaching $[0.1, 0.3]$. The hardware used during training was single Nvidia Tesla V100 SMX2 GPU with 32GB of VRAM.

4
Related Work

4.1 Gorila

Gorila[19] was the first massively distributed architecture for deep reinforcement learning. Here multiple actors and learners run asynchronously in parallel. Additionally, parameters for the policy network are stored in a distributed database. The database receives gradients from the learners and each machine in the distributed database maintains and updates a subset of parameters. These asynchronous updates of parameters provide significant speedups in training and additional work has proven that it can be achieved on a single machine using only CPU-cores[16]. However, the massively distributed approach creates a complex data flow when machines are working asynchronously. A large network of machines that needs to communicate also has the potential to be error-prone or not reach is full training speed capabilities due to the overhead that comes with communication. Compare the massively distributed approach with this work where the entire system can be implemented on a single machine which removes a large portion of expensive communication that would occur over the network.

4.2 Asynchronous n-step Q-learning

Asynchronous n-step Q-learning[16] uses a framework that takes the distributed approach developed in Gorila and parallelizes it to a single machine. By running multiple threads on a single machine, the overhead in communication can be removed and enables Hogwild![24] style of network parameters updates. Furthermore, because all actor-learner threads run different behavior policies, the experience replay buffer can be removed. Because multiple actor-learners perform online updates of the parameter server, the overall changes are likely to not be correlated. Training stability is then provided by actor-learners running different behavior policies. By removing the need for a replay buffer, the overall memory consumption of the framework is reduced. However, by using a replay memory, transitions that rarely occur have a chance to be included in the training step again if there is still information to be learned. Without the replay buffer, an actor-learner process needs to find the same state and perform the same action. Furthermore, the globally shared policy network parameters can provide delays by threads waiting to access the parameters or by the read of the parameters before every gradient computation. By instead sending transitions to a replay memory, as done in this work, these delays are avoided since the only time the shared memory is read is when the actor networks

are updated.

4.3 Ape-X

Following the Gorila architecture was the Ape-X[13] architecture. Much like in this work, the Ape-X architecture uses multiple actors that generate experiences that are sent to a replay buffer. A single learner samples the replay buffer and performs backpropagation on the policy network. However, unlike Ape-x we make use of the efficient scaling of batch sizes and have multiple environments running in parallel on a single actor. We see that the rate of which the state space can be explored is increased batching states from the environments and propagated at once. Furthermore, the Ape-X architecture makes use of multiple actor machines running in parallel which makes it a more scalable approach where our approach is limited to a single machine. Finally, Ape-X makes use of a dueling network architecture[29] that after the convolutional layers, splits into two data streams that separately predict the value and advantage function. The split data stream results in faster learning during the training and convergence at the same result as using a single stream network.

5
Results

The main results of the thesis can be found in Figure 5.1 where the mean Q-value can be seen over the training period. The policy networks for $d = 5$, $d = 7$ and $d = 9$ has converged after roughly 0.4, 5 and 9 hours respectively. After these points, no significant improvements to the Q-value is seen. The initial spike for $d = 5$ is most likely a result of syndromes being easy to solve. Once p increases the learned pattern is not as useful anymore which causes the drop down to a Q-value of 70. (Note that the mean Q-value for $d = 9$ is collected for a smaller set of p values due to the excessive time to gather reliable statistics during training.)

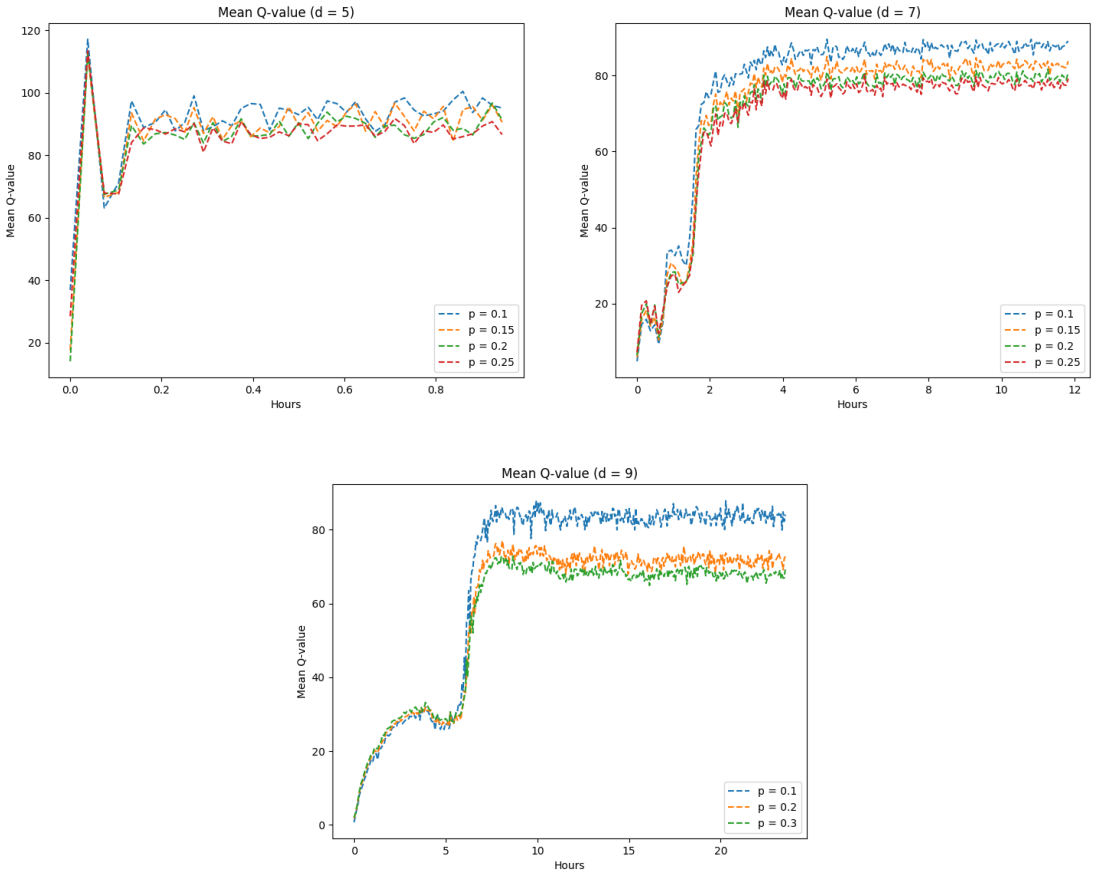


Figure 5.1: Mean Q-value over the training period for different probabilities of error, p and toric code sizes of $d = 5$, $d = 7$ and $d = 9$.

Table 5.1 displays the speedups in the training time of the decoder using the dis-

Table 5.1: Speed ups using our distributed training approach versus the sequential training.

	d=5	d=7	d=9
Sequential Convergence Time	5h	96h	-
Distributed Convergence Time	0.4h	5h	10h
Speed Up	12.5	19.2	-

tributed training approach compared to the sequential approach used in [9]. Training times from the sequentially trained decoder on $d = 9$ is unavailable. The data in Table 5.1 displays wall clock times which is sub-optimal. Variations in programming techniques, hardware, and knowledge in the implementation language can cause large differences in run times of the same program developed by different people. However, because wall clock times are the only data available from the sequential training, there are no other means of measuring speed up. Nevertheless, this does not invalidate the results because of two reasons: (1) both the sequentially and distributed trained decoders have been training on the same hardware, and (2) at least an order of magnitude in speed up is a strong argument for the distributed approach being faster. For future comparison, the mean q-values with respect to training steps can be found in Appendix B.

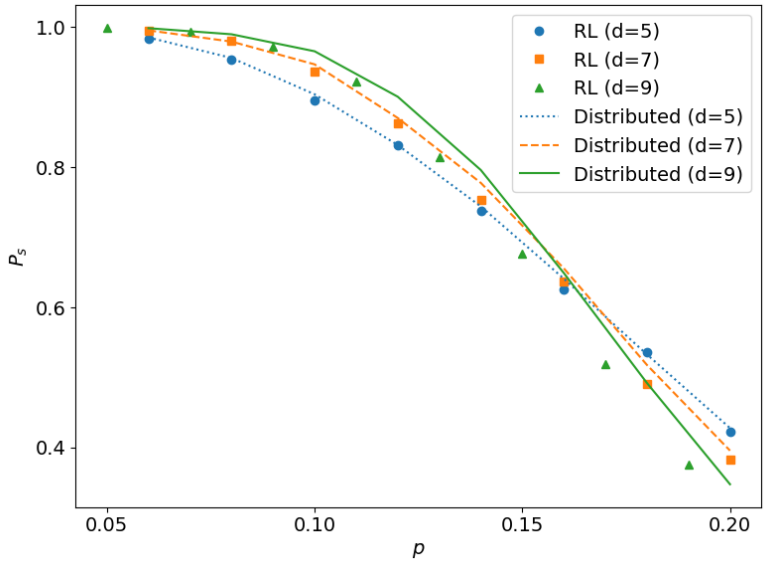


Figure 5.2: Error correction success rates, P_s, versus probability of generating an error, p. A comparison between the distributed training approach and sequential training approach, *RL*.

Comparing performances between the distributed training approach against the sequential training approach, referred to as *RL* in following Figures, reveal that the distributed performs just as well for $d = 5$ and $d = 7$. Figure 5.2 displays the result of decoders correcting syndromes from a depolarized noise model where P_s is the

success rate of correcting a syndrome and p the probability of generating an error. For a syndrome to be counted as successfully corrected, the final state must not only have all defects removed but must not have any non-trivial loops. From the crossing of $d = 5$ and $d = 7$ the threshold of 16.5% can be observed, just as in the sequential training. Below this threshold, an error correction can be guaranteed. From this, we conclude that a more efficient training approach will likely not increase the performance of the decoder. Instead, a new reward scheme needs to be developed as discussed in [9]. However, even though the mean Q-value in Figure 5.1 for the $d = 9$ decoder seems to have converged, we note that its performance is non-optimal, to a decoder with our reward scheme, from its crossing of $d = 7$ at roughly $p = 0.16$. Moreover, a comparison between the distributed and sequentially trained decoders for $d = 9$ shows that the distributed training approach performs better. Here, the sequentially trained decoder used a ResNet34[12] model[9] that has 21 million parameters whereas we used NN11 with roughly 900.000 parameters. A ResNet18 model for $d = 9$ was also trained using the distributed approach but was unable to converge. More on this in the Conclusion.

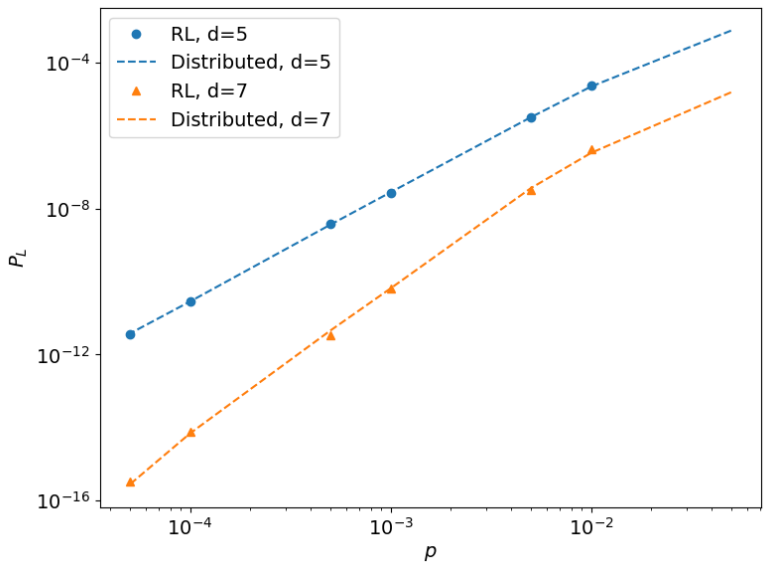


Figure 5.3: Fail rates P_L of the distributed trained decoder and the sequentially trained decoder, RL, for small error rates p.

The decoder is also benchmarked on small error rates by analytically deriving an expression for the theoretical fail rates using a depolarising noise model to lowest non-vanishing order of p [9]. This analytical expression is based on finding the shortest correction chains, which is similar to the behavior of a decoder with our reward scheme. Due to the difficulties in collecting statistics about failure rates for small p, testing of the decoder consisted of generating syndromes with errors in a single column or row that are likely to fail. Additional errors were then added to the syndrome with probability p. Figure 5.3 compares fail rates for the decoders trained in a distributed and sequential approach. The distributedly trained decoder performs just as well as the sequentially trained decoder which in turn performs ideally to our

reward scheme[9]. Due to the excessive time needed to gather reliable statistics for a decoder on $d = 9$, we have only compared this decoder for asymptotic fail rates. Table 5.2 displays the results of the asymptotic fail rates, i.e. the fail rate for the shortest error chains that can fail. Again, the decoder trained using a distributed approach performs just as well as the sequentially trained decoder for $d = 5$ and $d = 7$. Although the distributed training seems to be better than the sequential training and the analytical expression for $d = 9$, we believe that these statistics are not reliable. Mostly because of the reason previously stated; it is difficult to gather reliable statistics due to excessive time the test needs to run.

Table 5.2: Comparison of the asymptotic fail rates P_L.

	Analytic	Sequential	Distributed
d=5	1.51e-3	1.45e-3	1.48e-3
d=7	2.12e-5	2.07e-5	2.05e-5
d=9	2.50e-7	4.30e-7	2.00e-07

As an attempt to improve the success rate on a decoder for size $d = 9$, a linearly increasing p over the first 100.000 steps during training was tested. Figure 5.4 (left) displays a comparison of the success rate P_s between the decoders trained using a linear and a random strategy for p described in Method. Even though the performance looks relatively similar to each other, the random strategy is significantly better because of its result in Figure 5.4. Figure 5.4 (right) displays the result of decoders correcting syndromes from a depolarized noise model and p the probability of generating an error. Unlike Figure 5.2 we are here only interested in whether the decoder can solve the syndrome, regardless of any resulting loops in the final state. The decoder on $d = 9$ trained using a linear strategy for p has problems compared to the decoder trained with a random strategy. For comparison, the decoder for $d = 5$ and $d = 7$ has been added. These results provide more evidence that our decoder on $d = 9$ still behaves non-optimally.

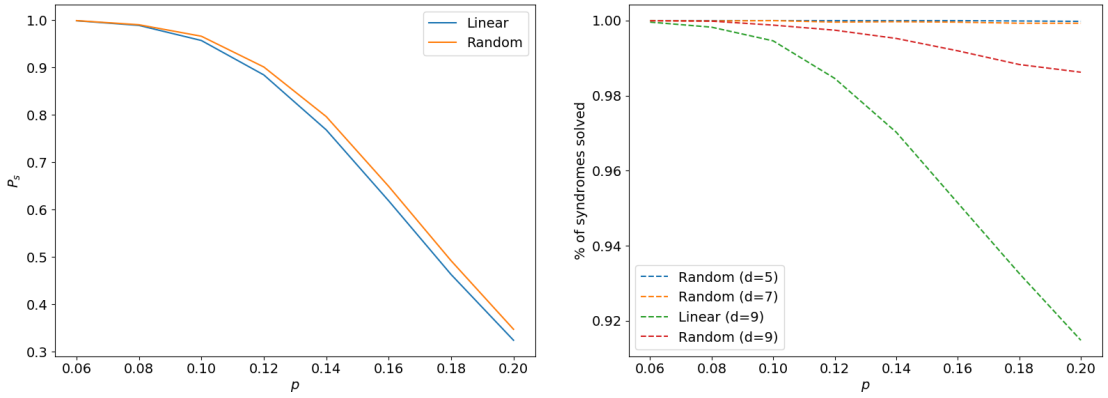


Figure 5.4: (Left) A comparison of success rates P_s between decoders trained using a linear and random strategy for p on toric code size $d = 9$. (Right) A comparison between different decoders on how many syndromes they solve, regardless of resulting loops in the final state.

6
Conclusion

We have found that by distributing the training for a decoder based on deep reinforcement learning, the training time can be decreased with at least a factor of 12. Performance of the decoders remain the same or slightly increase for system sizes of up to $d = 9$. We used the same network across all grid sizes compared to [9] who used NN11 for $d = 5$ and $d = 7$ and a ResNet34 model for $d = 9$. Despite NN11 having almost 20 times fewer parameters, NN11 performs better than ResNet34. We believe the faster rate of exploration of the state-space using the distributed training approach managed to find patterns the sequential training method was unable to find due to a time limit. We argue that these results pave the way for optimal decoders on larger grid sizes due to the separation of the training method and the decoder based on reinforcement learning. For example, evaluating new neural network architectures or reward schemes is only a matter of substituting the respective class or function.

However, even though our smaller network architecture performed better, we were not able to reach the code threshold of 16.5% for $d = 9$. The linear strategy for increasing p during the training turned out to perform worse. A possible explanation for this could be that the rate of which p increased was too high, resulting in the syndromes became too difficult too fast. Using the data for the number of transitions generated per second from Figure 3.2(right) and the size of the replay of according to Appendix A, the replay buffer was overwritten roughly every 47th minute. After 100.000 steps, p reached 0.3 which by comparing Figure 5.1($d = 9$) and Appendix B($d = 9$) roughly took 7h. This means that after about 7h and 47min into a 24h training, the replay buffer was filled with syndromes generated using a $p = 0.3$. Surprisingly, the 16h and 13min of training on syndromes generated using a $p = 0.3$ were not enough to improve performance.

Additionally, to improve performance on a decoder for size $d = 9$, we tried training a ResNet18 model to the task. But due to ResNet18 having almost 10 times more parameters, the batch size and number of environments running in parallel had to be restricted to avoid running out of VRAM on the GPU. In the end, a batch size of 16 and 16 environments in parallel was used. After 6 days of training, the ResNet18 model had not learned anything. To try to understand why we again trained the ResNet18 model but on a fixed syndrome to evaluate if whether it was a limitation to the training method or a bug in the model. The model did not either manage to solve a fixed syndrome after 12h of training, indicating that there could be a bug in our implementation of the ResNet18 model.

During the development, we ran into a problem that caused the network to only learn to apply a single operator, regardless of the syndrome. We observed that due to the random initialization of the policy network, there was always a bias towards one operator. At this time, instead of having a linearly decreasing ϵ, as described in the Method, we had a fixed ϵ for each environment as in the Ape-X framework. We believe that the ϵ-greedy policy, which uses the policy network, overflowed the replay buffer with actions according to the random initialization. The transitions sampled from the replay memory were then mostly of greedy origin which the policy network learned from. This became a feedback loop when the actors update their network from the policy network. To solve this problem we implemented the method of decreasing ϵ every update of the network, where exploration is initially forced.

6.1 Future Work

Most of the future work to speed up the training requires expanding the computational resources with additional GPUs. However, future work to speed up training without adding additional hardware would be to expand the neural network to exploit a dueling network architecture[29].

To run larger networks, such as ResNet18, and still utilize our efficient training approach, additional hardware needs to be incorporated. Training can be distributed across multiple machines, similar to the Ape-X framework, or distributed locally over multiple GPUs. By having access to more GPUs, the learner does no longer need to compete for resources during the training. Additionally, increasing the environment count for training of the decoder on size $d = 9$ could be the key for an optimal decoder. This was not possible during our work because of having only access to computational nodes with a single GPU and CPU.

With potentially additional hardware, Pytorch's Data Parallel[22] and Distributed Data Parallel[23] libraries allows for easy distribution of the training. In the Data Parallel library, the model is replicated on multiple local GPUs and the training batches are then divided into chunks between the models. The Distributed Data Parallel library allow training on multiple machines. But in order to do so, Pytorch needs to be built with a MPI backend. However, we found the asynchronous feature of MPI to be lackluster due to not being able to provide non-synchronizing asynchronous message passing. For future work, the communication of our training approach would need to be expanded to distribute training on GPUs to multiple machines.

Bibliography

[1] Numba. `http://numba.pydata.org/`, [Accessed 13 May 2020].

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[3] Philip Andreasson, Joel Johansson, Simon Liljestrand, and Mats Granath. Quantum error correction for the toric code using deep reinforcement learning. *Quantum*, 3:183, 2019.

[4] Enda Barrett, Jim Duggan, and Enda Howley. A parallel framework for bayesian reinforcement learning. *Connection Science*, 26(1):7–23, 2014.

[5] Sergey Bravyi, Martin Suchara, and Alexander Vargo. Efficient algorithms for maximum likelihood decoding in the surface code. *Physical Review A*, 90(3):032326, 2014.

[6] Laia Domingo Colomer, Michalis Skotiniotis, and Ramon Muñoz-Tapia. Reinforcement learning for optimal error correction of toric codes. *Physics Letters A*, page 126353, 2020.

[7] Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. Topological quantum memory. *Journal of Mathematical Physics*, 43(9):4452–4505, 2002.

[8] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.

[9] David Fitzek, Mattias Eliasson, Anton Frisk Kockum, and Mats Granath. Deep q-learning decoder for depolarizing noise on the toric code. *arXiv preprint arXiv:1912.12919*, 2019.

[10] Austin G Fowler. Minimum weight perfect matching of fault-tolerant topological quantum error correction in average $o(1)$ parallel time. *arXiv preprint arXiv:1307.1740*, 2013.

[11] Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3):032324, 2012.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[13] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[14] A Yu Kitaev. Fault-tolerant quantum computation by anyons. *Annals of Physics*, 303(1):2–30, 2003.

[15] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

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

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

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[19] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[20] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[22] PyTorch. DataParallel. `https://pytorch.org/docs/stable/nn.html#dataparallel`, [Accessed 6 May 2020].

[23] PyTorch. DistributedDataParallel. `https://pytorch.org/docs/stable/nn.html#distributeddataparallel`, [Accessed 6 May 2020].

[24] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[25] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[26] Andrew M Steane. Error correcting codes in quantum theory. *Physical Review Letters*, 77(5):793, 1996.

[27] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[28] Barbara M Terhal. Quantum error correction for quantum memories. *Reviews of Modern Physics*, 87(2):307, 2015.

[29] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[30] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

A

Training Parameters

Table A.1: Hyper-parameters for training of toric code sizes $d = 5$ and $d = 7$.

Learning rate	0.00025
Discount factor	0.95
Batch size	32
Update frequency of target network and actor policy network	50
Actor max steps per episode	75
Actor local buffer size	100
Actor no environments in parallel	100
Replay memory size	1 000 000
Replay memory α	0.6
Replay memory β	0.4
Replay memory size before sampling	5000
α for exploration parameter ϵ	7
β for exploration parameter ϵ	0.8

Table A.2: Hyper-parameters for training of toric code size $d = 9$.

Learning rate	0.00025
Discount factor	0.95
Batch size	32
Update frequency of target network and actor policy network	1000
Actor max steps per episode	75
Actor local buffer size	100
Actor no environments in parallel	100
Replay memory size	1 000 000
Replay memory α	0.6
Replay memory β	0.4
Replay memory size before sampling	5000
α for exploration parameter ϵ	7
β for exploration parameter ϵ	0.8

Table A.3: Network architecture for $d = 5$, $d = 7$ and $d = 9$. All convolutional layers uses kernel size 3 and stride 1. Layer 2 and on uses zero-padding of 1.

	Layer Type	Size	No. Parameters
1	Conv2d	128	2,432
2	Conv2d	128	147,584
3	Conv2d	120	138,360
4	Conv2d	111	119,991
5	Conv2d	104	104,000
6	Conv2d	103	96,511
7	Conv2d	90	83,520
8	Conv2d	80	64,880
9	Conv2d	73	52,633
10	Conv2d	71	46,718
11	Conv2d	64	40,960
12	Linear	3	1,731
			899,320

B

Mean Q-value Vs Training Steps

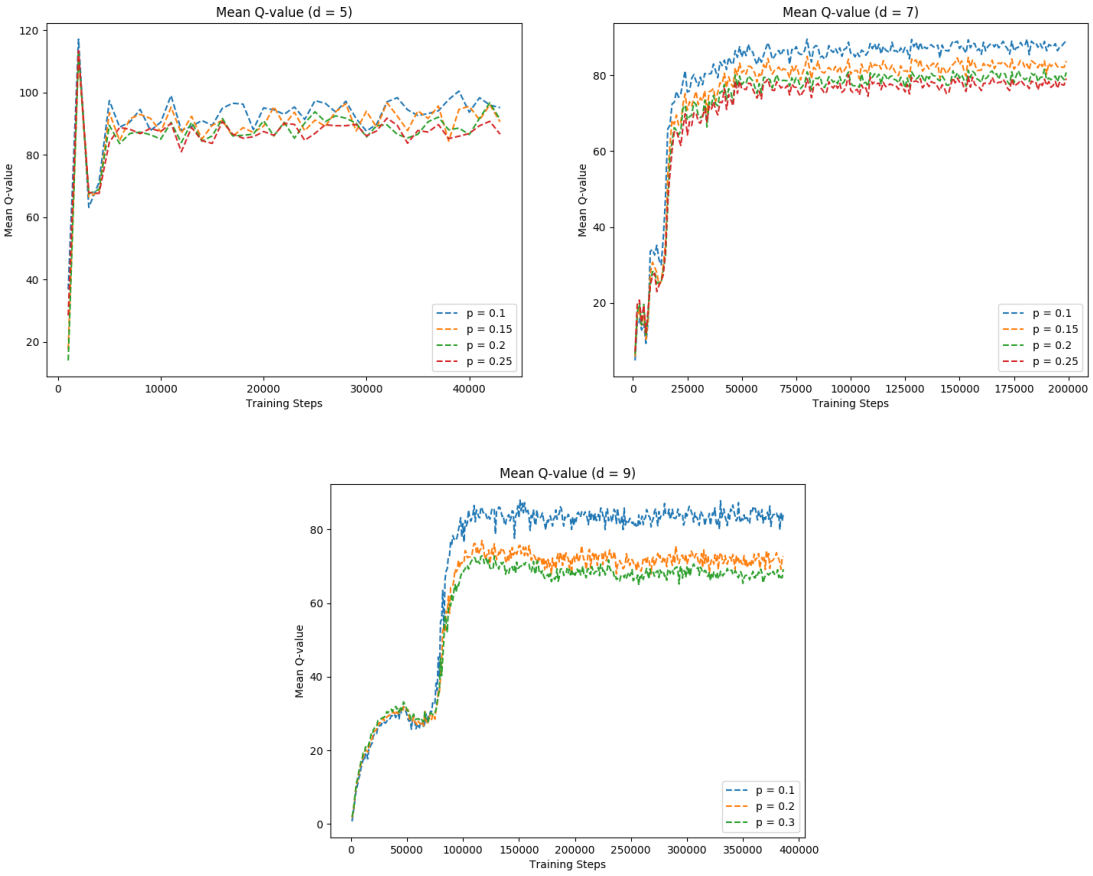


Figure B.1: Mean Q-value versus training steps for different probabilities of error on toric code sizes of $d = 5$, $d = 7$ and $d = 9$. Note that $d = 9$ is has a different set of p values due to the excessive time to test during the training.