Quantum error correction on the toric code using two distinct reinforcement learning game frameworks

Spoor, Lindsay

Quantum error correction on the toric code using two distinct reinforcement learning game frameworks

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
PHYSICS

Author :	Lindsay Spoor
Student ID :	1983822
Supervisor :	Evert van Nieuwenburg
Second corrector :	Aske Plaat

Leiden, The Netherlands, March 1, 2024

Quantum error correction on the toric code using two distinct reinforcement learning game frameworks

Lindsay Spoor

Huygens-Kamerlingh Onnes Laboratory, Leiden University
P.O. Box 9500, 2300 RA Leiden, The Netherlands

March 1, 2024

Abstract

This project employs reinforcement learning techniques to explore novel decoding strategies for quantum error correction, particularly focusing on the toric code, to address the challenge of maintaining stable quantum states for fault-tolerant quantum computing. Two game frameworks are established, including a novel dynamic game framework applicable to the training and measuring of RL agents and potential application in multi-agent scenarios. The RL agents use Stable Baselines 3's Proximal Policy Optimization and show to achieve Minimum Weight Perfect Matching performance on 3×3 toric code lattices in both the static and dynamic game frameworks.

Contents

Version of March 1, 2024– Created March 1, 2024 - 18:58

Chapter 1

Introduction

The realm of quantum computing has expanded into various specialized areas of research. Whether it is quantum algorithms, hardware advancements, or quantum error correction, researchers are met with distinct challenges in their fields. Nonetheless, they share a common obstacle: the requirement to maintain stable quantum states to effectively execute quantum computations. Addressing this challenge, particularly in reducing errors due to state decoherence, remains central to the progress of quantum computing. Unlike classical computers, quantum computers rely on the delicate rules of quantum mechanics. The fragile nature of quantum states necessitates robustness in order to preserve the integrity of information. Obstacles being faced in preserving state stability are due to imperfect control mechanisms, environmental instability and the quantum systems inherently being subject to noise. Being able to sustainably execute large-scale quantum computations, requiring quantum states that can persist for long enough and are robust to the described obstacles, is called fault-tolerant quantum computing. Mitigation of errors arising from state decoherence is one of the biggest challenges being faced in this field of research [1].

One of the most prominent approaches to achieve this fault-tolerance is quantum error correction (QEC). This approach involves encoding quantum information redundantly across multiple qubits, enabling the detection and correction of errors without disturbing the encoded information. However, QEC frameworks can not be set-up as straightforward as for classical error correction. QEC is subject to constraints imposed by quantum information theory, necessitating clever and sophisticated protocols for the encoding and decoding of information [1, 2]. One of the most popular frameworks that fulfil these constraints are topological error correct-

ing codes, from which Kitaev's toric code is a subset [3]. The toric code exhibits robustness against local errors and provides a foundation for the exploration of error correcting strategies. The standard error decoder for the toric code is the Minimum Weight Perfect Matching (MWPM) algorithm [4, 5], although this algorithm has inherent limitations and may not generalize well to real-world scenarios.

In recent years, machine learning techniques have gained prominence in diverse fields of research, offering novel approaches to complex physics problems which were otherwise oftentimes tackled by deterministic and statistical models. Machine learning is capable of capturing structures and learn strategies autonomously, either via i) supervised learning or ii) unsupervised learning, where information can be derived from provided datasets, or iii) reinforcement learning (RL), presenting a dynamic learning setup in which the learner is able to interact with the problem and receives feedback on its applied strategy. Moreover, the field of deep reinforcement learning (DRL) leverages on deep neural networks, and has recently demonstrated remarkable milestones in achieving super-human performance, especially on board games [6]. Treating a QEC problem on the toric code as a strategic interactive board game offers the possibility to explore novel decoding strategies tailored to real-world quantum error scenarios.

This research introduces two distinct frameworks for the training and evaluation of RL agents on the toric code: i) a static game scenario, whereby the sequence of decoding actions does not affect the end result, offering the ability to benchmark the performance against the prominent single-shot MWPM decoder, and ii) a dynamic game scenario similar to a survival-type game called Decodoku, allowing the RL agent to learn a strategy that involves sequential decision-making [7]. The latter framework aims to mimic a scenario for real-time error correction in a quantum computer. Furthermore, it aims to lay out the groundwork for future investigations into multi-agent RL frameworks, allowing for further exploration of novel RL-driven decoding strategies, which will ultimately deepen our understanding of QEC.

In Chapter 2, a theoretical background on error correction is outlined, necessary for understanding the remainder of this research. Chapter 3 provides a general overview of RL and delves deeper into the algorithms applied in this study. Chapter 4 introduces the decoding game tailored for the RL agents, in which both the environment and the two distinct frameworks are explained. Chapter 5 presents the results for both frameworks. Chapter 6 concludes the thesis by providing a summary and discussion of the results presented, alongside offering an outlook for future research.

Chapter 2

Error Correction

2.1 Classical Error Correction

Classical information sciences rely on binary representations, where data is expressed as sequences of bits with values of '0' or '1'. In order to meet the demands on high-performance communication networks such as telephone networks and the internet, it was necessary for classical computer computations to be performed consistently and reliably [8]. Therefore, classical error correction theory was established. The fundamental idea behind error correction is to add redundancy to the system, i.e. increasing the number of bits used to represent a given amount of information, which can be achieved by employing a set of instructions. This is called an *error correction code*. The simplest example of such a code is the three-bit repetition code, which maps each bit value:

$$\{0, 1\} \rightarrow \{000, 111\}, \tag{2.1}$$

in which '000' and '111' represent the *logical codewords*. For example, if a message would be transmitted including a single bit '0' $\rightarrow$ '000' and a single bit-flip error occurs during transmission, yielding the received message to be '010', the recipient can deduce the initial message by looking at the majority vote of the codeword. However, this reasoning does not hold for cases in which more than one bit-flip error occurs. In the two bit-flip error case, the majority vote will lead to an incorrect message deduction, and in the case of three bit-flip errors this may even yield a valid codeword in which it is impossible to detect the presence of an error. Referring to this last case, the *code distance d* is defined by the number of errors it can correct t,

$$d = 2t + 1. \tag{2.2}$$

For example, following Eq. 2.2, the three-bit repetition code thus has $d = 3$ and $t = 1$ [2]. Adding more redundancy to the system by enlarging the code distance therefore provides a way to construct a more robust error correction protocol.

2.2 Quantum Error Correction

Transferring the classical information system into a quantum information system, the classical bit is replaced by the qubit and can be written in its general form as

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \tag{2.3}$$

where α and β are the probability amplitudes represented as complex numbers and are constrained to the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. Here,

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.4}$$

are the orthonormal basis states in which a qubit can be represented. Referring to Eq. 2.3, information encoded by qubits can therefore be situated in a superposition of those basis states. Thereby the quantum computational space scales as 2^n, where n is the total number of qubits of the system. The general form from Eq. 2.3 can be written in a geometric representation as

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle\,, \tag{2.5}$$

such that $|\cos\frac{\theta}{2}|^2 + |e^{i\phi}\sin\frac{\theta}{2}|^2 = 1$ still holds. Qubit states can then, using this representation, be projected onto the surface of the Bloch sphere, as shown in Fig. 2.1.

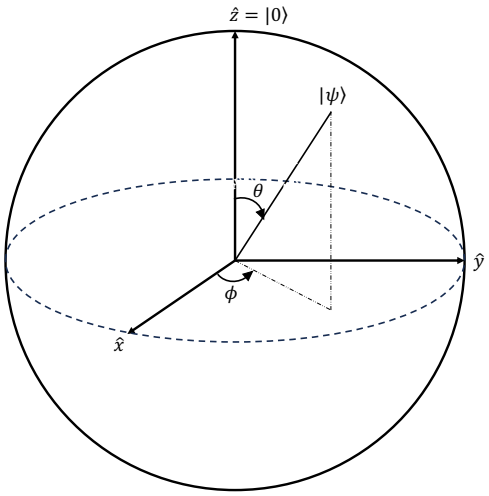


Figure 2.1: *The geometrical representation of a qubit state can be represented as a point on the surface on the Bloch sphere.*

Any point on the Bloch sphere represents a quantum state, automatically implying that the qubit is susceptible to infinitely many possible errors. This makes the construction of error correcting codes challenging. However, this infinite set of errors can be digitised in such a way that one can sufficiently correct for a finite set of errors [9]. Designing error correcting codes must therefore be done by encoding quantum information redundantly and in a clever way, as quantum error correction is not as straight-forward as classical error correction. Quantum error correction brings complications that need to be taken into account when designing a valid error correction code:

- **No-cloning theorem** [10]: Qubits are subject to the so-called no-cloning theorem, stating that it is impossible for a qubit state $|\psi\rangle$ to be cloned by performing a unitary operation on it which would allow for
$$U_{\text{clone}}(|\psi\rangle \otimes |0\rangle) \rightarrow |\psi\rangle \otimes |\psi\rangle \,. \tag{2.6}$$

- **Bit-flips and phase-flips**: Not only must one account for bit-flip errors on the quantum code; one must also consider the possibility of phase-flip errors, in contrast to the classical case in which only bit-flips are possible.

- **Wavefunction collapse**: Measuring a qubit results in a collapsing wavefunction. Therefore, measurements necessary to carry out an error correction procedure must be chosen cleverly and carefully to prevent encoded information from being inadvertently erased.

Adding redundancy to the system therefore needs to be done according to the principles from above. The *logical qubit* can be introduced as the piece of quantum information one wants to encode and protect. The logical qubits can be represented in a k-dimensional subspace, and its basis states can be mapped onto a larger n-dimensional Hilbert space, allowing for error detection and correction [9]. Once an error occurs on the logical qubit, this means the information encoded in this space has been compromised, which is defined as a *logical error*.

2.3 Toric Code

One of the most popular classes of quantum error correcting codes rely on the principle of topological error correction. These types of codes are composed of qubits placed on a lattice structure, such that only interactions with nearby qubits are possible. This gives topological quantum codes the characteristic to be intrinsically local. Information is encoded in logical qubits, across multiple physical qubits in a non-local collective manner by forming non-trivial closed loops on the topological structure. Non-triviality here means that a closed loop is not contractible on the surface, and therefore could not enclose a homologically trivial subset of the surface, as shown in Fig. 2.2. This means that information is stored in global degrees of freedom, rather than being localized on the individual physical qubits. Once an error is introduced on a physical qubit, this does not imply that there is also a logical error. Therefore, logical operations can still be done if the code is robust to logical errors [11]. This locality characteristic makes topological quantum codes naturally protective against quantum states suffering from decoherence, possibly achieving the desirable fault-tolerance for quantum computing [12].

One of the most basic types of topological error correcting codes is the surface code. The surface code represents code states as logical qubit values encoded across the entire code distance d, which are constructed out of $2d \times d$ physical qubits placed on a two-dimensional lattice.

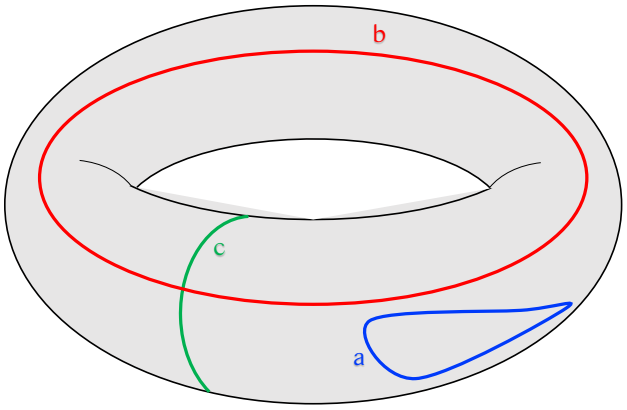


Figure 2.2: *Examples of several closed loops on a surface wrapped around a torus. Loop a is a curve enclosing a homologically trivial surface, whereas loops b and c are homologically non-trivial loops as they are not contractible on the surface.*

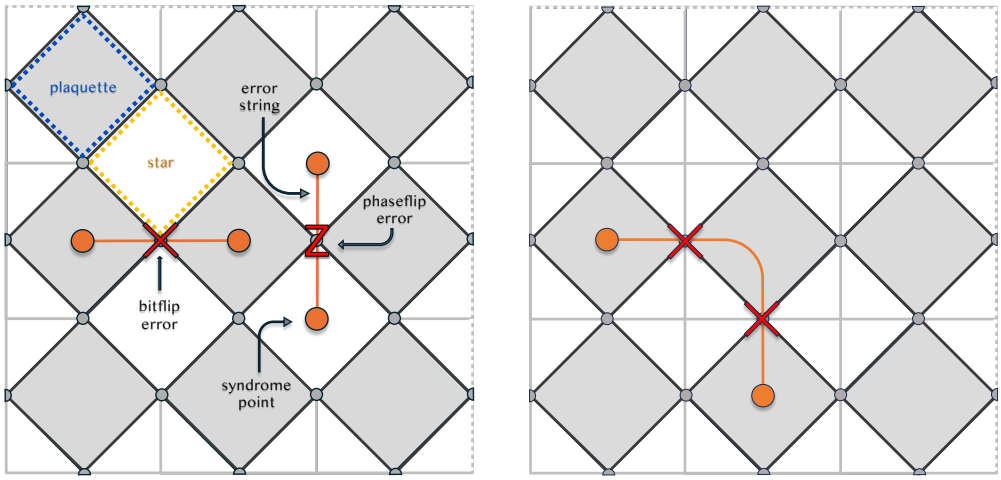
Surface codes under the constraint of periodic boundary conditions, as a square lattice drawn on a torus like the shape of the object in Fig. 2.2, can be identified as the toric code [3]. Toric codes are characterised by two types of stabilizer operators: plaquettes and stars. These stabilizers can be viewed as mutually commuting check operators, which are constructed out of the products of the adjacent 4 Pauli Z (plaquette) or X (star) operators. Fig. 2.3 shows a visualisation of a toric code. The system can be described by the Hamiltonian from Eq. 2.7:

$$H = -\sum_{\text{plaquette}} P - \sum_{\text{star}} S = -\sum_{\text{plaquette}} \prod_i Z_i - \sum_{\text{star}} \prod_i X_i, \tag{2.7}$$

where i sums over all edges adjacent to the plaquettes and stars. The groundstate is spanned by the space for which all stabilizers have eigenvalue $+1$ [13]. One can check that this groundstate is 4-fold degenerate, resulting in 4 classes of loops that can not be deformed into each other as shown in Fig. 2.4 [14].

Bit-flip and phase-flip errors can occur on one of the physical qubits in the form of Pauli X and Pauli Z operations respectively. If an odd number of qubits adjacent to their plaquettes and stars have an error, the eigenvalues of those stabilizers will be -1, and, once measured, this will result in a so-called syndrome [1]. As shown in Fig. 2.3a, syndrome endpoints

appear at plaquettes or stars if their eigenvalues are -1 due to bit-flip or phase-flip errors respectively, and are attached to each other through an error string. Errors adjacent to the same plaquette or star can move syndrome points across the lattice and the error string can become longer, as shown in Fig. 2.3b. When an error string forms a contractible loop on the lattice, the syndrome endpoints will then vanish and the system is brought back to the ground state energy. The decoding problem therefore has as its main goal to bring back the system to its ground state space without forming non-contractible loops on the toric code, so that the logical qubits can be protected from errors. A decoder could therefore successfully remove syndromes only by undoing all original errors, or closing the errors strings into trivial loops.



(a) ***(b)***

Figure 2.3: *Toric code ($d = 3$). 2.3a) Plaquette and star operators are formed by qubits located on the vertices of the thereby enclosed squares. The dotted lines on the right and top edges of the lattice indicate the periodic nature of the toric code. Bit-flip and phase-flip operations indicated by the red X and Z respectively on one of the vertices induces a bit-flip and phase-flip errors leaving behind error strings with at its ends syndrome points. Syndrome points introduced by bit-flip or phase-flip errors occur at plaquettes or stars respectively. 2.3b) Two bit-flip operations happening on vertices adjacent to the same plaquette can connect syndrome points through an error string.*

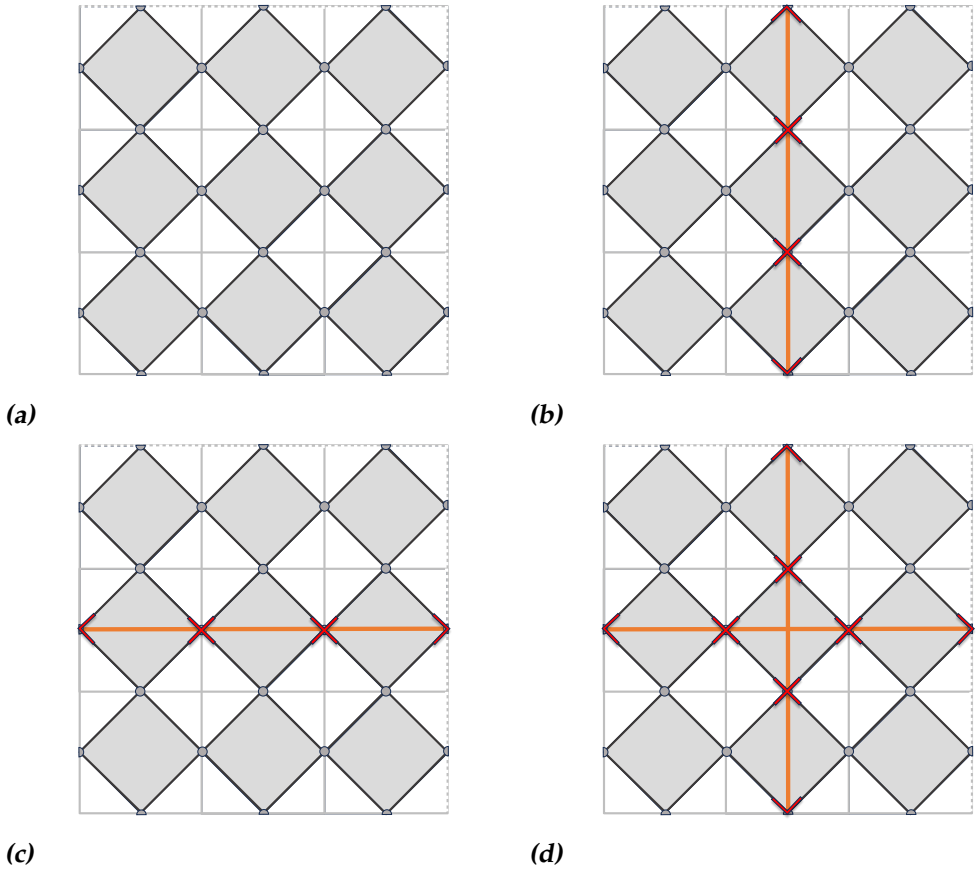
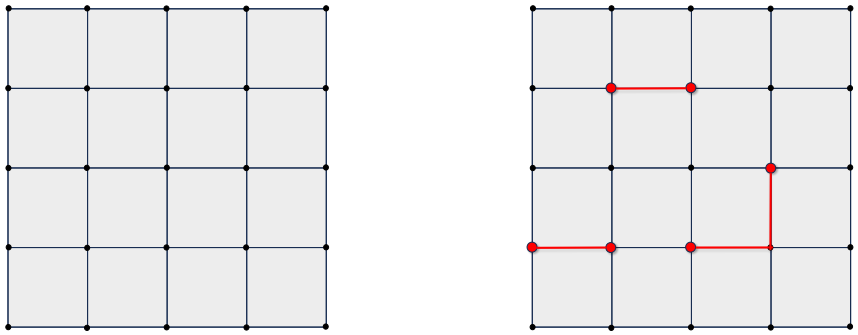


Figure 2.4: *Examples of 4 classes of non-trivial loops that span the basis of the 4-fold degenerate ground state Hamiltonian of the toric code. Each of these classes can not be deformed into each other.*

The physical qubit error rate p_{error} denotes the probability of a physical qubit encountering an error. On small lattice sizes, equivalent to small code distances d in the toric code, the number of errors on qubits increases with p_{error}. Consequently, the code's robustness improves with larger lattice sizes at low error rates [15]. However, as error rates rise, scaling to larger system sizes becomes less robust. An *error threshold* p_{th}, analogous to the order-disorder phase transition in the random-bond Ising model [12], defines this transition behavior. Consequently, error decoders on topological quantum codes are typically assessed based on this error threshold, which sets the upper limit for p_{error} necessary for achieving fault-tolerant quantum computing across larger system sizes.

2.4 Minimum Weight Perfect Matching

A prominent decoding algorithm in quantum error correction is the Minimum Weight Perfect Matching (MWPM) decoder. MWPM is oftentimes used as a benchmark to assess performance of a decoder in quantum error correction, specifically for decoding problems on toric codes. The four stages of MWPM are visualized in Fig. 2.5. The algorithm operates by finding the minimum weight perfect matching in a graph representation of the board with syndrome points. Weights are assessed to the given matching graph representation of the syndrome by identifying all possible error configurations. This is done by finding all possible error strings from one syndrome point to another, for all syndrome points in the graph. A minimum weight perfect matching is then established by selecting the matching with the minimum weight, that is, selecting the paths on the graph with the shortest error strings [4, 16, 17].



(a) *Matching graph* ***(b)*** *Error*

(c) *Assess weights* ***(d)*** *Minimum-weight perfect matching*

Figure 2.5: *Stages of MWPM decoding algorithm for a 5x5 surface code. Vertices correspond to the check operators/stabilizer operators.*

Version of March 1, 2024– Created March 1, 2024 - 18:58

It is important to note that the MWPM decoder represents the optimal decoding solution for quantum error correction only under the assumption of uncorrelated noise, where only Pauli X operators are applied to the qubits with a probability of p_{error}. For depolarizing noise, not only Pauli X or Z are possible operations, but also Y operators can be applied on the qubits with error rate p_{error}, for which MWPM fails to have an optimal decoding strategy. Errors can in this case arise on two distinct lattices, one accounting for the plaquettes, and one for the stars. Y-errors are constructed on the combination of both lattices, making it therefore impossible to identify the origin of syndromes arising on either the stars or plaquettes. Errors therefore must be assumed to be bit-flip only and independent from each other in order for MWPM to succeed, whereas it is unlikely that in real-world scenarios noise is uncorrelated. Furthermore, the existence of multiple error paths with the same weight can inadvertently lead to logical errors, like the scenario in Fig. 2.6. Due to the statistical properties of the noise and the underlying physics of the toric code described in section 2.3, the MWPM decoder achieves a performance limit of $p_{\text{th}} \approx 0.11$ for bit-flip noise and $p_{\text{th}} \approx 0.15$ for depolarizing noise [4, 12, 18].

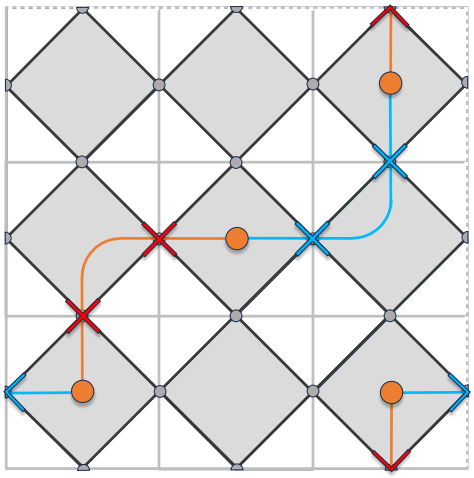


Figure 2.6: *Toric code ($d = 3$). The red bit-flip operations are introduced, causing the orange syndrome endpoints attached by their error strings. The decoder applies the blue bit-flip operations, causing to a non-trivial loop which leads to a logical error.*

Another decoding algorithm resting on the same principle as the MWPM decoder is the Maximum Likelihood (ML) decoder [18]. This decoder identifies the most probable error configuration given both the noise model and the syndrome, providing a more general approach. However, this al-

Version of March 1, 2024– Created March 1, 2024 - 18:58

gorithm also involves more complex and intensive calculations, which is why the MWPM decoder is preferable on error correction problems on surface codes. Furthermore, it is only optimal on the uncorrelated noise model like the MWPM decoder.

In cases where errors do not independently arise on the toric code, MWPM might not be able to accurately capture the underlying error structure of syndrome points, potentially leading to sub-optimal error correction. Furthermore, because of its single-shot nature, MWPM is not suitable for dynamic decoding scenarios. Time-evolving noise or other dynamic error situations require real-time correction strategies. Therefore, one needs to consider other clever approaches to pursue increased decoder performance. Reinforcement learning algorithms have the potential to capture these complex error patterns on a time axis, possibly even for correlated noise, and act on them in a time-dynamic playground. Framing the decoding problem as a game to be solved by RL might give rise to more optimal results and yield a more robust decoder [13].

Chapter 3

Reinforcement Learning

Reinforcement Learning (RL) is a subset of machine learning that focuses on training a learner to make decisions by learning from interactions with the problem and receiving feedback from it. Unlike supervised learning, where the algorithm is provided with labeled data, and unsupervised learning, where the algorithm discovers patterns in unlabeled data, RL involves a decision-maker called an agent. The agent learns through trial and error in a so-called environment. This environment can be thought of as the playground the agent interacts with, representing the formulated problem to be solved. The agent receives feedback from the environment in the form of a positive or negative reward, allowing it to improve its decision-making over time. Its strategy is described by a policy, which is strongly influenced by the rewards the agent gets for different actions to perform on the environment in a certain state. The agent will adjust the policy according to whether it gets a (relatively) high or a low reward in return for the chosen action in a certain state. A common trade-off in RL that has to be made is on how much the agent is allowed to explore new strategies, and how much it exploits its well-known strategy over the course of the training process. This decision making and optimization process during training is influenced by the current state of the environment, and the agent aims to learn the best actions to take in different states to maximize its cumulative reward over time, i.e. finding the optimal policy [19].

A reinforcement learning agent can optimize its strategy using either a value-based or a policy-based approach. A value-based method makes use of a value function which determines what are the best actions to take given a state. Values are assigned to states such that the expected return of all future states that will likely follow up the current given state will be maximized. Policy-based methods, on the other hand, use a so-called

policy that determines what action should be taken at a given state. This policy could for example be a neural network, whose parameters can be optimized during the training process, taking the state observation of the environment as an input and the next action as an output. The described RL cycle is visualised in Fig. 3.1. This research uses a policy-based method to find a decoding strategy for the toric code, which will be described more in detail in the next sections.

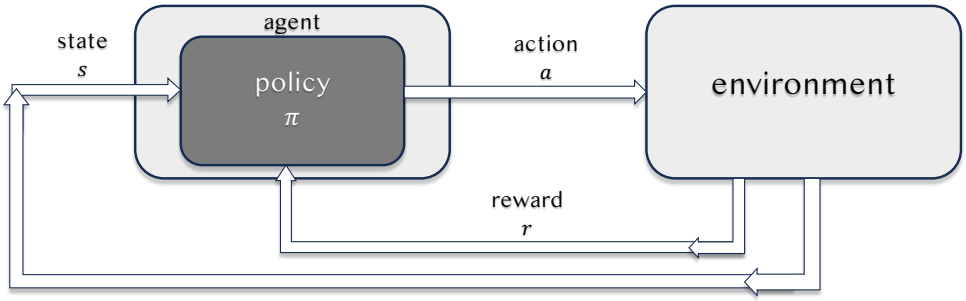


Figure 3.1: *Reinforcement learning cycle. The agent observes state s of the environment, and uses this as an input into its current policy π, which will then output the action a to take. This action is performed onto the environment, which will return a reward r.*

3.1 Deep Reinforcement Learning

Deep reinforcement learning (DRL) extends traditional RL by employing deep neural networks to approximate the agent's strategy. This allows DRL to handle higher-dimensional state and action spaces. However, the use of neural networks introduces complexity and opacity to the learning process, which makes DRL oftentimes look more like a black box compared to traditional RL algorithms. This is because deep neural networks involve many layers of interconnected neurons, making it difficult to interpret how decisions are made based on input data.

In DRL, hyperparameters are parameters that are not learned during

training but must be set beforehand and can significantly impact the performance of the algorithm. Examples of hyperparameters include the learning rate, exploration rate, and network architecture. Tuning hyperparameters involves experimenting with different values for these parameters to find the combination that maximizes the performance of the DRL algorithm on a given task.

3.2 Policy-based RL

One can define the policy π as the agent's strategy, which determines what action $a \in A$ to take given a state $s \in S$ which is observed from the environment [19]. The policy could be deterministic, where a function can map all possible states onto all possible actions. In this case, the policy would be described by Eq. 3.1.

$$\pi(s) = a. \tag{3.1}$$

A policy can also be non-deterministic, in this case called stochastic. A stochastic policy determines the probability of action a given state s, using a distribution $\pi(a|s)$, described by Eq. 3.2:

$$\pi_\theta(a|s) = \mathbb{P}(a|s, \theta), \tag{3.2}$$

θ being the set of trainable parameters that describes the probability distribution in the unit interval $[0, 1]$. One of the primary differences between a deterministic and a stochastic policy is that a stochastic policy may choose different actions for the same provided state, whereas for a deterministic policy it will always select the same single action for each given state. This gives a stochastic policy the advantage that it is able to capture uncertainty in the environment. Therefore, the decoding problem in this study will involve a stochastic policy to optimize a strategy for an RL decoder.

Another distinguishment in policy-based RL can be made on the learning approach of the agent. One can sub-divide RL into model-based and model-free approaches. Model-based RL uses a model that predicts the environment, the agent learns by predicting the consequences of its actions. Model-free RL, however, can only learn through experience, as it does not learn through a predictive model that describes the environment. Since the decoding problem does not provide an accurate model for the decoding strategy and the interest of this study leans more towards discovering any novel solutions to decode syndromes, the latter approach will be used for this study.

3.2.1 Policy Gradient Method

There are several optimization techniques RL can use to optimize the agent's policy in order to find the best strategy. One of these optimization techniques is called the Policy Gradient method in which the parameters of the policy are directly being optimized to improve the agent's policy. Optimization happens by following the directions of the parameter updates that result in higher rewards, i.e. calculating the gradients of the parameter updates. The parameters of the policy can be described by θ, such that the policy can be described by a probability distribution for actions $a \in A$ given states $s \in S$, as defined in Eq 3.2 [19]. In order to do so, one must first define an objective function to optimize. Eq. 3.3 defines this in the form of the expected discounted return from a given initial state s_0:

$$L^{PG}(\theta) = \mathbb{E}_{\theta\pi}\big[G_t|s_t = s_0\big] = \mathbb{E}_{\theta\pi}\Big[\sum_{t=0}^{\infty}\gamma^{t-1}r_t|s_t = s_0\Big]. \tag{3.3}$$

In Eq. 3.3, G_t is the discounted return, which represents the trajectory of action a_t, state s_t and reward r_t over all timesteps t for a given episode, described in Eq. 3.4:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots . \tag{3.4}$$

The discount factor γ values the worth of future rewards $R_{t+\ldots}$, where $0 < \gamma < 1$. If γ is close to 0, this means that future rewards are viewed as less important than rewards more towards the present of the sequence. In order to optimize θ, gradient ascent is performed, with the objective to maximize the expected discounted return, which is shown by Eq. 3.5:

$$\nabla_\theta L^{PG}(\theta) = \mathbb{E}_{\theta\pi}\big[\nabla_\pi(\log(\pi_\theta(a_t|s_t))G_t\big]. \tag{3.5}$$

The optimal policy π_{θ^*} is then found by the optimal parameters θ^* that maximize the expected discounter return.

3.2.2 Actor Critic

Even though policy-based RL excels in learning stochastic policies, its challenge lies in minimizing the gradient variance. However, value-based RL offers sample efficiency and stability. A convenient method to tackle those issues is to combine both policy-based and value-based methods. The variance of the policy gradient can be reduced by making the discounted return G_t smaller. Actor Critic methods do this by having an *actor* update the policy distribution, and a *critic* suggesting the direction of the policy

update, which is determined by the value function. One of the ways to do so, is to define an estimated advantage shown by Eq. 3.6:

$$\hat{A}(s_t, a_t) = r_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t). \tag{3.6}$$

Here, r_{t+1} is the reward of the next timestep and the advantage estimate is parameterized by network parameters θ and $V(s_t), V(s_{t+1})$ are the value functions at timesteps t and $t+1$ respectively, which determine the values of taking action s at their respective timesteps. The policy gradient update from Eq. 3.5 can then be rewritten to Eq. 3.7:

$$\nabla_\theta L^{PG}(\theta) = \mathbb{E}_{\theta\pi}\big[\nabla_\pi(\log(\pi_\theta(a_t|s_t)\hat{A}(s_t, a_t)\big]. \tag{3.7}$$

3.2.3 Proximal Policy Optimization

One of the most important trade-offs in RL is finding the right balance between performance and algorithm complexity. Proximal Policy Optimization (PPO) is a Policy Gradient Method that makes sure the policy updates are not too large. It uses a clipping parameter that constrains the policy in a small update range, in order to avoid large weight updates. Eq. 3.8 describes this small update range by using a clipped surrogate objective function in which $\hat{A}(\theta_{\text{old}})$ is the estimated advantage using the previous policy parameters, making this method Actor Critic.

$$L^{CLIP}(\theta) = \mathbb{E}_{\theta\pi}\big[\min\big(r(\theta)\hat{A}(\theta_{\text{old}}), \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}(\theta_{\text{old}})\big)\big]. \tag{3.8}$$

The ratio function $r(\theta)$ describes the probability of taking action a at state s using the current parameters, as compared to the policy using the previous parameters, as defined in Eq. 3.9.

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}. \tag{3.9}$$

If $r(\theta) > 1$, the action a at state s is more likely in the current policy's parameters than in the old policy. If $0 < r(\theta) < 1$, the action was more likely for the old policy's parameters than for the current one. This way, $r(\theta)$ describes the divergence between the old and the current policy, which can be clipped between $1 \pm \epsilon$, with ϵ being the clipping parameter. This ensures the policy update to not be too large [20]. The RL agents trained on the decoding problem in this research use PPO to find the optimal set of parameters.

Version of March 1, 2024– Created March 1, 2024 - 18:58

3.3 Reinforcement Learning applied to QEC

Recently, researchers have turned to reinforcement learning (RL) as a promising approach to tackle the complexities of QEC. Several publications have explored the applications of RL techniques applied to QEC, offering insights into the potential of these methods [21]. For example, training an agent to find optimal error correction paths using deep Q-networks achieved accuracy close to that of MWPM decoders [22, 23]. Other approaches use evolutionary algorithms and have shown to generate a policy network with decoding performance comparable to other RL approaches, including Q-learning [13]. Furthermore, Graph Neural Networks have shown promising results on stabilizer codes. Each of these publications contributes valuable insights into the application of reinforcement learning techniques for quantum error correction, paving the way for more efficient and reliable quantum computing systems.

Chapter 4

Decoding Game

The primary objective of this research is to develop an RL agent tailored to address the decoding problem associated with the toric code. This problem can be framed as a game wherein the RL agent undertakes the role of decoding errors on a toric code lattice. The toric code can be represented as a board, which is initialized in its ground state, characterized by a code distance d, equivalent to the lattice size, and subsequently subjected to errors on qubits. The RL agent receives state observations in the form of syndrome endpoints resulting from syndrome measurements. The agent's task involves manipulating the qubits strategically to eliminate the syndrome and restore the system to its ground state, thereby rectifying any introduced errors by undoing the initial bit-flips or forming trivial closed loops of error strings on the board. The rules and rewards governing this decoding game must be chosen carefully, as they significantly influence the agent's policy optimization during training, which can consequently impact the performance results during evaluation. An important aspect of this study involves comparing the RL agent's performance with the MWPM decoder, offering insights into the viability of RL as a decoder. Because of the single-shot nature of MWPM, first, a static framework is constructed such that the RL decoder's strategy is similar to that of a MWPM decoder. Subsequently, the framework is extended to a dynamic setting, necessitating adjustments to the game rules, allowing the RL decoder to endure on the board for prolonged periods without encountering logical errors. For simplicity, this research mainly focuses on uncorrelated bit-flip noise only, although the constructed environment allows for translation to correlated bit-flip noise. The following sections will delve deeper into the environment construction and the choice of rules for each of the game frameworks.

4.1 Environment

The environment describing the toric code decoding problem utilizes the OpenAI Gym [24] framework and was originally constructed for the Neuro Evolution of Augmenting Topologies (NEAT) algorithm [13]. The environment is tailored to the purpose of this research, such that it is compatible with Gym's observation space and action space implementations.

The toric code lattice is defined on a square board of dimensions $d \times d$, with d being the code distance. This lattice contains $2d \times d$ qubits and features $d \times d$ plaquette and star operators. The periodic boundaries are defined at initialization to account for the periodic nature of the toric code, crucial for defining the logical qubit over which the information is stored. Each qubit in the toric code lattice is susceptible to errors, specifically uncorrelated bit-flip operations in the case of this research, with a probability denoted as p_{error}. The generation of these errors by the environment leads to the occurrence of syndrome endpoints at the corresponding plaquette locations on the board, and the measurement of the board syndrome is fed to the agent as a state observation s. Therefore, the observation space is of size $d \times d$, corresponding to the $d \times d$ plaquettes on the defined toric code. The agent can, in turn, select an action a to perform a bit-flip operation in order to clear the board from syndrome endpoints. The action space of the agent is designed to address the task of clearing the board from syndrome endpoints. Therefore, the total action space is of size $2d \times d$, with each action corresponding to a qubit location to perform a bit-flip on. It is important to note that, in order to reduce the size of the action space, the agent is only allowed to choose actions that flip qubits adjacent to syndrome endpoints. Thus, an action mask is applied to all actions that are not allowed, as visualized in Fig. 4.1 [25]. This decision is grounded in the philosophy that actions on qubits non-adjacent to the plaquettes of syndrome endpoints may only introduce additional errors, rather than shifting or removing the already existing syndrome endpoints. Besides, training convergence will happen faster as the chance of choosing the right action will be higher when the action space becomes smaller.

Another important aspect of the game setup is that the environment has access to all information about which qubits are subject to an error, whereas the agent only has access to information about the locations of syndrome endpoints and the locations of the qubits it has flipped itself. Therefore, the qubits flipped by the environment are stored in a so-called *hidden state*, which remains secret to the RL agent. The RL agent does not have access to this information during training. To assess the performance of the decoding game, the board's logical error is evaluated, there-

fore the access to this hidden state is necessary to the environment. This assessment involves checking for logical errors based on a specific criterion, which is detailed in Algorithm 1.

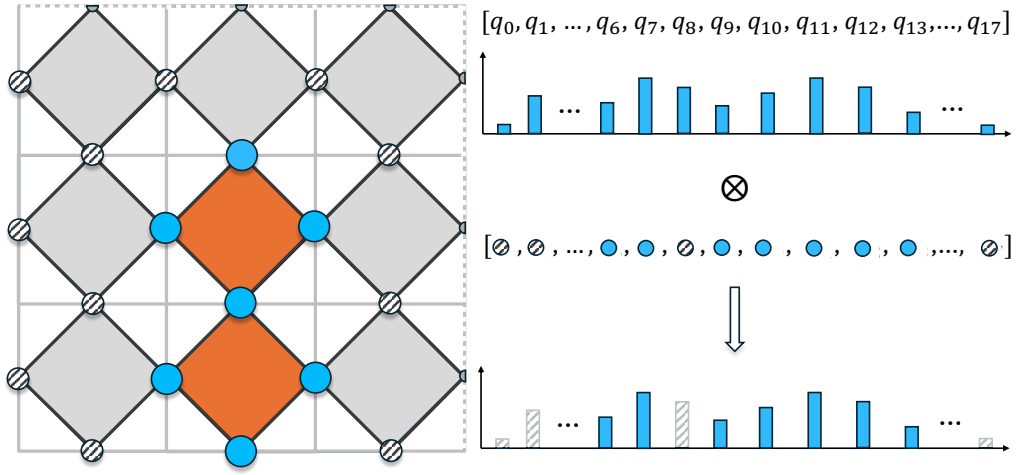


Figure 4.1: *Action mask procedure. Only qubits adjacent to plaquettes indicating a syndrome endpoint in orange are allowed actions to select for the agent, indicated by the blue locations. The probability distribution over the entire action space is masked to 0 on qubit locations not adjacent to the syndrome endpoints.*

In Algorithm 1, N_x and N_y count the total number of logical errors present on the board across the horizontal and vertical code distances of the board, respectively. In theory, it could be that there is more than one logical error present on the toric code, and in that case, multiples of two logical errors would cancel each other out, as this is equivalent to removing an already existing error from the logical qubit. Therefore, this check will indicate a logical error when either N_x or N_y counts an odd number. For each qubit defined on the boundary, the algorithm checks whether it is flipped. This is because if there would be a non-trivial loop present on the board, this loop should, by definition, cross the boundary, since the logical qubit is encoded across the whole code distance in either the horizontal or the vertical direction. From a flipped qubit on the boundary, all possible error strings are computed by evaluating possible error string attachments between all flipped qubits, including the hidden state qubits, on the board. When, during this error string calculation, a syndrome endpoint is encountered, this instantly means that there is no loop since loops

would erase syndrome endpoints. Therefore, only when an error string is closed, i.e., the error string starts at the same qubit as where it ends its path, must the algorithm check how many times the string has crossed the boundary. Referring back to Fig. 2.2, only if the loop crosses the boundary an odd number of times does this imply a logical error, as these types of loops are non-contractible on the surface at the boundary.

Algorithm 1 Check Logical Errors

```
N_x, N_y ← 0
for each q in boundary qubits do
    if q is flipped then
        find error string
        if string is closed then
            n_x, n_y ← #times string crosses x or y boundary
            N_x += n_x, N_y += n_y
        end if
    end if
end for
if N_x or N_y is odd then
    Logical error
else
    No logical error
end if
```

The selection of timestep rewards and the moments at which the board is assessed for logical errors vary across different frameworks, as elaborated in the subsequent sections.

4.2 Static Framework

In the static game framework, the agent aims to reach the terminal board state, defined as the ground state of the system. This terminal state can be achieved when all syndrome endpoints are removed, rendering the board *empty*. The agent's objective is to reach this terminal board state without resulting in a logical error. To accomplish this, the agent is permitted to perform as many actions as necessary to achieve its goal. Only at the conclusion of the game, when no syndrome endpoints remain for decoding, will the board be evaluated for logical errors using Algorithm 1. In this manner, the agent's performance can be compared to that of the MWPM

decoder. The reward system should be designed to enable the agent to learn a strategy similar to the MWPM algorithm while also allowing flexibility in the agent's strategy. Therefore, three different timestep rewards, r_c, r_l, and r_s, can be defined for different game situations. The agent receives a *continuing reward* r_c for each action performed on the environment that does not result in a terminal board state. If an action leads to a terminal board state with a logical error, the agent receives a *logical error reward* r_l. Conversely, if the terminal board state is reached without a logical error, the agent receives a *success reward* r_s. Fig. 4.2 and 4.3 illustrate two different game scenarios the agent may encounter during a single game episode, one involving a logical error and the other resulting in success, respectively. Although each game episode consists of multiple timesteps, this game framework can be considered a single-shot game because the board is evaluated only at its terminal state, rendering the order in which the agent performs actions irrelevant to the agent's policy. Algorithm 2 provides an overview of the static game framework.

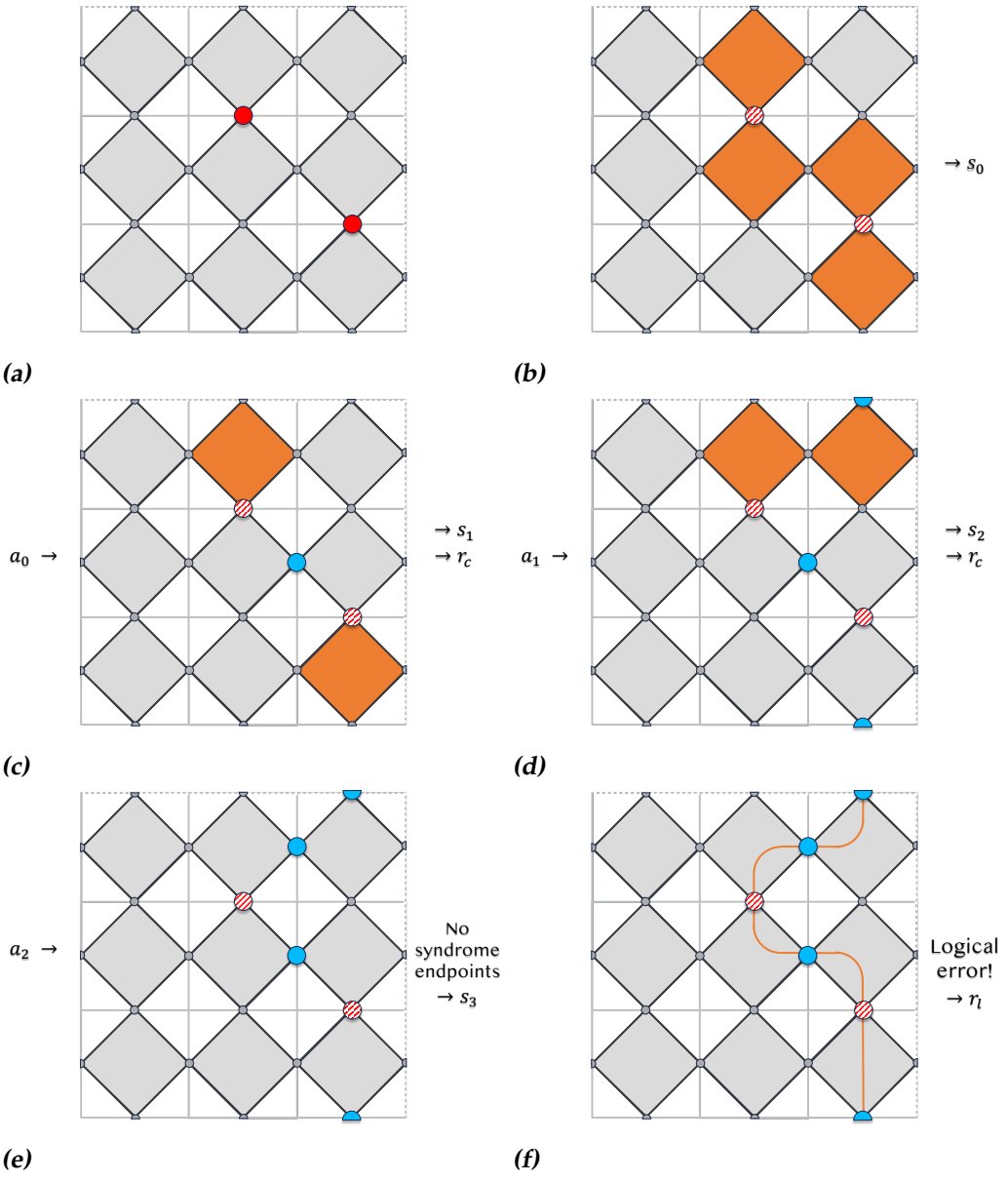


Figure 4.2: *Static framework scenario that results in a logical error. 4.2a) Initial errors generated by the environment in red. 4.2b) Syndrome measurement indicated by the orange plaquettes, board state s_0 fed as input to agent. The red striped qubits indicate that these flips are a secret to the agent. 4.2c, 4.2d, 4.2e) Agent performs action a_0, a_1, a_2 in blue on environment subsequently, syndromes are measured and agent receives subsequent board states s_1, s_2, s_3 with for s_1 and s_2 their corresponding timestep rewards r_c. 4.2f) Empty board, logical error check results in termination step reward r_l.*

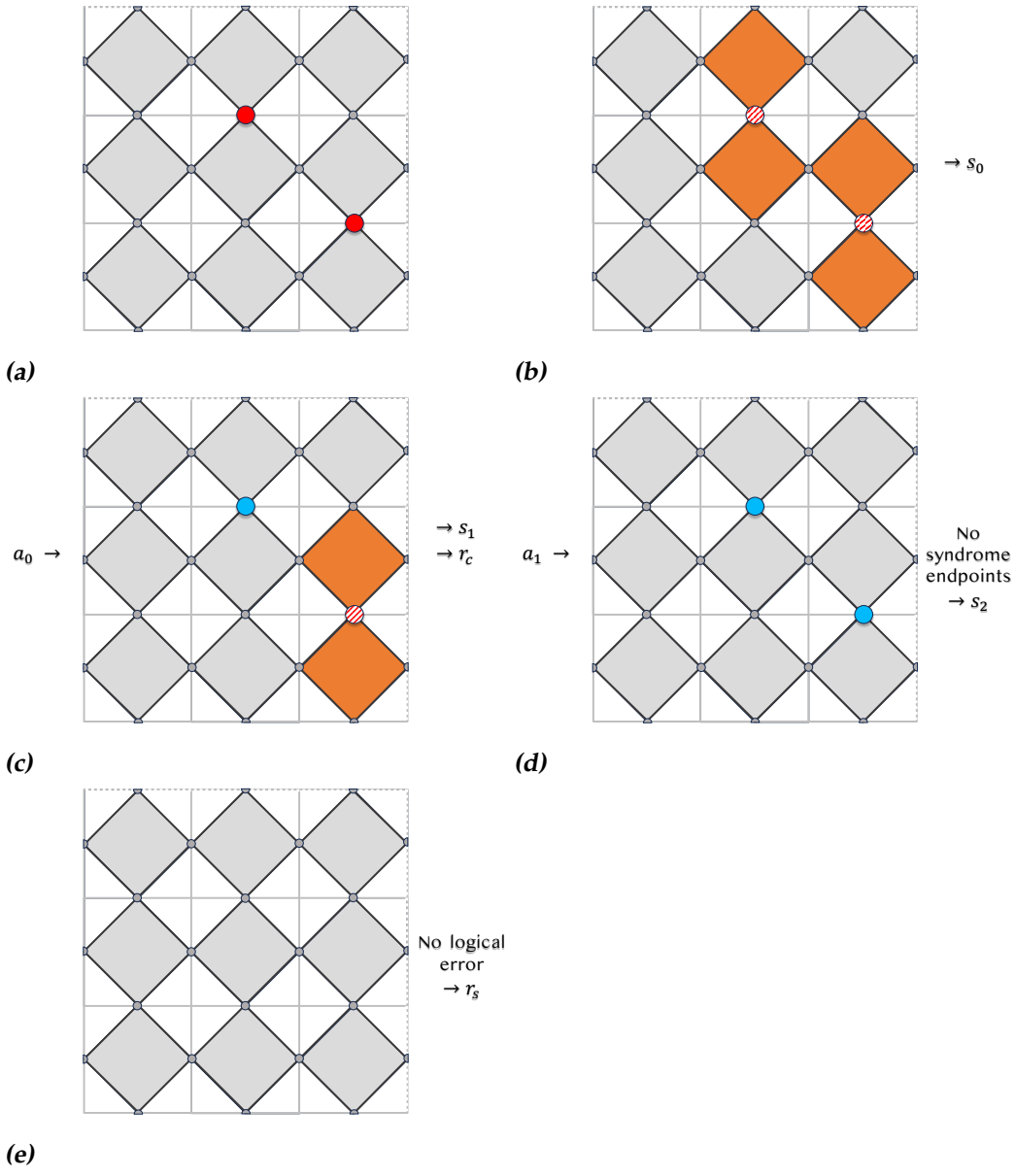


Figure 4.3: *Static framework scenario that results in a successful decoding sequence. 4.2a) Initial errors generated by the environment in red. 4.2b) Syndrome measurement indicated by the orange plaquettes, board state s_0 fed as input to agent. The red striped qubits indicate that these flips are a secret to the agent. 4.3c, 4.3d) Agent performs action a_0, a_1 in blue on environment subsequently, syndromes are measured and agent receives subsequent board states s_1, s_2 with for s_1 its timestep reward r_c. 4.3e) Empty board, logical error check results in termination step reward r_s.*

Algorithm 2 Static Game Framework

Initialize: Environment, p_{error}, board of size $d \times d$, max_moves, rewards r_c, r_s, r_l, Done $\leftarrow$ False
$n \leftarrow 0$
Generate initial errors on board with error rate p_{error}
Measure syndrome
while $n < max\_moves$ **do**
    while not Done **do**
        Agent $\leftarrow$ state observation s
        Policy $\pi \rightarrow$ action a (flip qubit on location a)
        Measure syndrome
        if board has syndrome endpoints **then**
            Agent $\leftarrow r_c$
            Done $\leftarrow$ False
        else
            Check for logical errors (Algorithm 1)
            if logical error **then**
                Agent $\leftarrow r_l$
                Done $\leftarrow$ True
            else
                Agent $\leftarrow r_s$
                Done $\leftarrow$ True
            end if
        end if
        $n+ = 1$
    end while
end while

4.3 Dynamic Framework

For the dynamic game framework, the game rules need adjustment to allow the agent to decode syndrome endpoints on the board for as long as it can survive, that is, without encountering a logical error. The goal of introducing this framework is to build an agent that can endure on the board for prolonged periods, ultimately contributing to fault-tolerant quantum computing where the quantum code needs consistent protection against errors to ensure large quantum computations. Unlike the static framework, where the board is evaluated for logical errors only when brought back to the ground state, the dynamic framework performs this check after each bit-flip operation on the board.

A timestep counter n and an iteration step parameter k are introduced so that new errors are introduced on the board every kth timestep. Note that in this framework, described by Algorithm 3, errors generated by the environment are not dependent on a physical qubit error rate p_{error} as in the static framework. Here, a discrete number of randomly selected qubits are flipped when it is the environment's turn to generate errors, simplifying the training setup and providing a clearer grasp of the agent's strategy. This is done because the agent's performance and policy optimization depend on numerous parameters. For example, as k increases, the agent is allowed to 'survive' on the board for more timesteps automatically. At initialization, N qubits are flipped, and every kth timestep, N_{new} qubits are flipped by the environment, adding complexity to the game setup and influencing the agent's policy. If necessary, however, this framework is suitable for translation to p_{error}. Another rule added to the environment is an extra action in the action space, which for this framework will therefore be of size $(2d \times d) + 1$. The extra action corresponds with the action to be selected when there are no syndrome endpoints present on the board. In this case, and in the absence of a logical error, the agent needs to perform no bit-flip on any of the qubits, as it would only introduce extra unnecessary syndrome endpoints. Therefore, following this rule, the only possible action for the agent when the board is empty is to *do nothing*. Furthermore, the agent can receive three types of timestep rewards, r_c, r_l, and r_e. Like in the static framework, r_c and r_l are the *continuing reward* and *logical error reward*, respectively. If, after performing action a on the board, no logical error occurs, the agent receives r_c. If, however, a bit-flip at qubit location a results in a logical error, the game terminates, and the agent receives r_l. The agent receives an *empty board reward* r_e when the board indicates no syndrome endpoints, and the agent needs to do nothing. This distinction between r_c and r_e ensures that the agent understands the difference be-

tween an empty board and a board with remaining syndrome points. Just as we, as humans, are able to live in a messy room, we would, however, have a much clearer mind when we keep our room tidy and remove all the trash. Fig. 4.4 and 4.5 illustrate two dynamic game scenarios. In Fig. 4.4, the agent fails to keep the board clean, resulting in a messy scenario at future timesteps, increasing the likelihood of encountering a logical error when a new error is introduced. Conversely, in Fig. 4.5, the agent keeps the board clean, receives reward r_e for an empty board state, and has less difficulty keeping track of all syndrome endpoints it needs to decode. One of the rules of the environment for reaching an empty board state is that in this case, the flipped qubit information in both the hidden state and the qubit states known to the agent is reset back to 0. This ensures that when the next error is introduced by the environment, the 'old' memory does not interfere with the agent's ability to decode future problems when performing the check on logical errors.

Algorithm 3 Dynamic Game Framework

Initialize: Environment, N, k, N_{new}, board of size $d \times d$, max_moves, rewards r_c, r_e, r_l, Done $\leftarrow$ False
$n \leftarrow 0$
Generate N randomly selected initial errors on board
Measure syndrome
Check for logical errors (Algorithm 1)
while $n < max\_moves$ **do**
    while not Done **do**
        if $(n \mod k) == 0$ **then**
            Generate N_{new} new randomly selected errors on board
            Measure syndrome
            Check for logical errors (Algorithm 1)
            if logical error **then**
                Agent $\leftarrow r_l$
                Done $\leftarrow$ True
            end if
        end if
        if not Done **then**
            Agent $\leftarrow$ state observation s
            Policy $\pi \rightarrow$ action a (flip qubit on location a)
            if $a ==$ 'do nothing' **then**
                Agent $\leftarrow r_e$
                Done $\leftarrow$ False
            else
                Measure syndrome
                Check for logical errors (Algorithm 1)
                if logical error **then**
                    Agent $\leftarrow r_l$
                    Done $\leftarrow$ True
                else
                    Agent $\leftarrow r_c$
                    Done $\leftarrow$ False
                end if
            end if
            $n+ = 1$
        end if
    end while
end while
Done $\leftarrow$ True

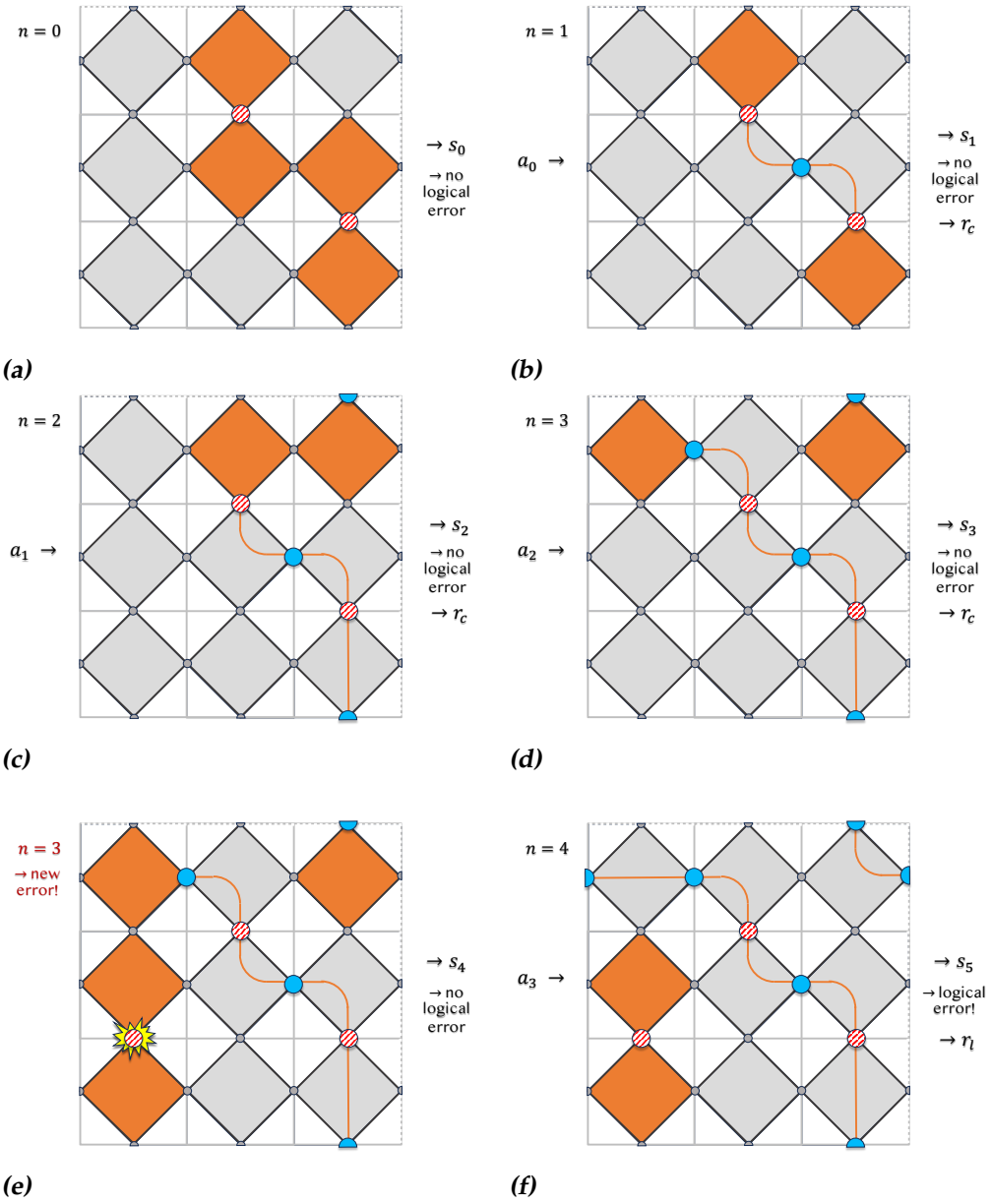


Figure 4.4: *Dynamic framework scenario resulting in a logical error, where* $N = 2$, $k = 3$, $N_{new} = 1$. *4.4a) Environment introduces errors. Syndrome measurement indicated by the orange plaquettes, board is checked on logical errors, board state* s_0 *fed as input to agent. The red striped qubits indicate that these flips are a secret to the agent. 4.4b, 4.4c, 4.4d) Agent performs actions* a_0, a_1, a_2 *in blue on environment subsequently, board is checked on logical errors, and agent receives subsequent board states* s_1, s_2, s_3 *with their timestep rewards* r_c. *4.4e* $(n \mod k) = 0$, *new error introduced by the environment and board is checked on logical errors, agent receives board state* s_4. *4.4f) Agent performs action* a_3 *on the board, resulting in a logical error, agent receives reward* r_l.

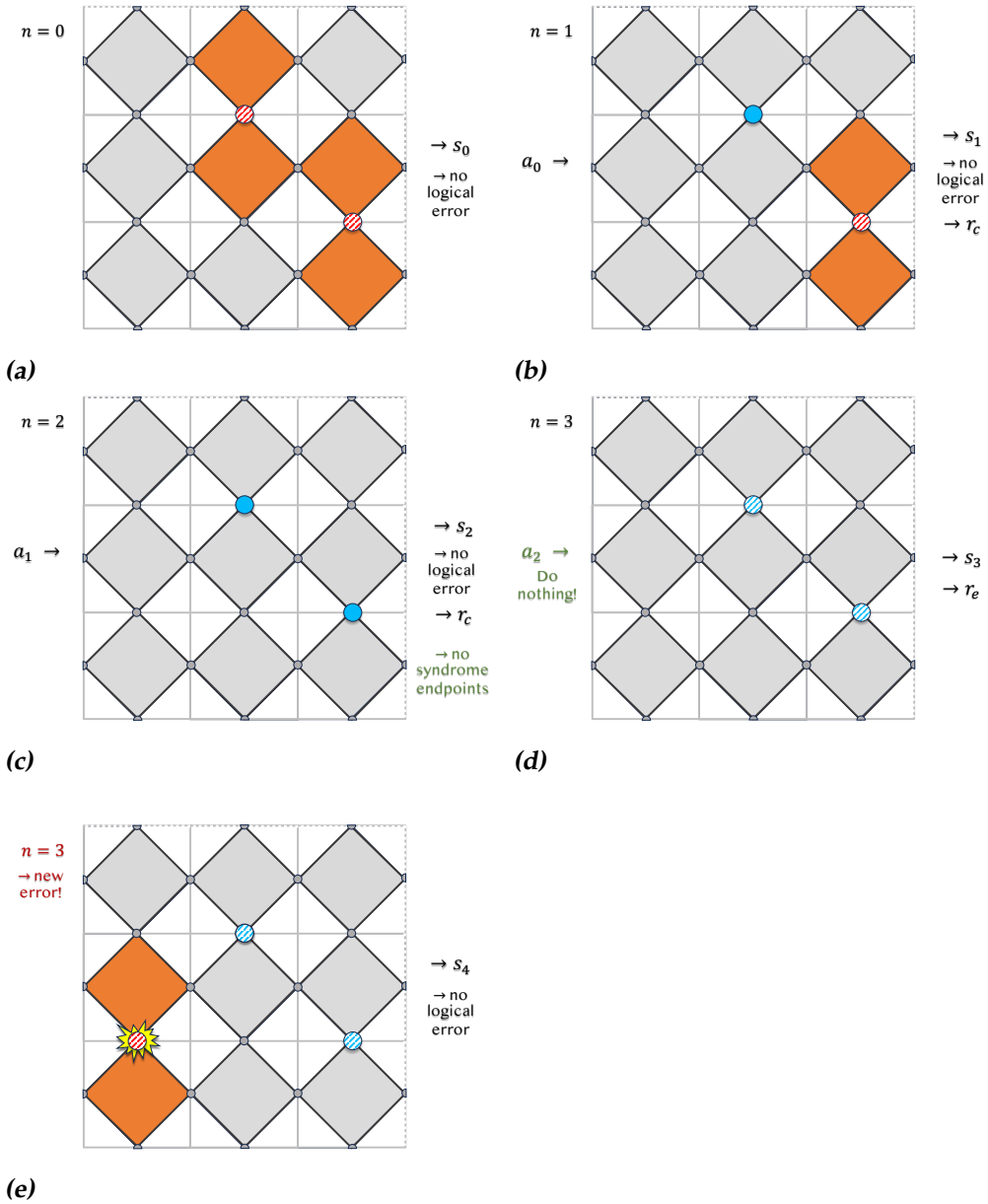


Figure 4.5: *Dynamic framework scenario where agent survives on the board for the shown timesteps, where $N = 2$, $k = 3$, $N_{new} = 1$. 4.5a) Environment introduces errors, board state s_0 fed as input to agent. The red striped qubits indicate that these flips are a secret to the agent. 4.5b, 4.5c) Agent performs action a_0, a_1 in blue on environment subsequently, board is checked on logical errors, and agent receives subsequent board states s_1, s_2 with their timestep rewards r_c. 4.5d) Board is empty, agent does nothing and receives state s_3 with reward r_e. The blue striped qubits indicate that the board is completely 'clean' and ready for the next error to be introduced by the environment. 4.5e $(n \mod k) = 0$, new error introduced by the environment and board is checked on logical errors.*

Chapter 5

Results

The PPO algorithm within the Stable Baselines 3 library will be used as the agent to be trained on the problem [20, 26]. The agent's policies are described by a neural network with a multi-layer perceptron architecture, whose network parameters are to be optimized in order to find the optimal decoding strategy.

5.1 Static Framework

In the static framework training procedure, the agent's primary goal is to decode the board syndromes in the most efficient manner possible. It achieves this by removing all syndrome endpoints in as few moves as possible. Additionally, since the agent must reach the terminal board state, which is the empty board, it should be rewarded upon reaching this state. First, the agent's hyperparameters and the reward scheme of the environment need to be tuned. Afterwards, three different PPO agents are trained on a fixed error rate p_{error} for board examples of sizes $d = 3, 5, 7$. The agents are evaluated and their performances are measured by their success rate p_s, which is defined as the number of cases in which the agent had successfully decoded the syndromes divided by the total number of evaluation examples. The performance of the MWPM decoder on the same set of evaluation examples is measured and used as a benchmark.

5.1.1 Training Setup

The primary goal is to find an agent that removes all syndrome endpoints by flipping as few qubits as possible without encountering a logical error. The agent should receive positive rewards for r_l and r_s upon reaching the terminal board state and should be penalized for each timestep it takes to achieve this state, aiming to align its strategy more closely with that of the MWPM decoder. Therefore, the training convergence and evaluation performance of four different reward schemes are compared with each other in order to find the best settings. Fig. 5.1 shows this training convergence for the first 10.000 training iterations. Note that one training iteration is not the same as one training timestep; the latter corresponds with the environment advancing with one timestep*, whereas one iteration corresponds to the completion of one single game episode, that is, when the agent has reached the terminal board state. The rewards are normalized such that a valid comparison can be made, since otherwise these different reward schemes will result in different orders of magnitude for their maximum rewards. Fig. 5.1 and 5.2 show small differences in reward schemes $(r_c, r_l, r_s) = (-1, 5, 10)$ and $(r_c, r_l, r_s) = (-1, 1, 10)$ for both the training convergence and evaluation performance, and both converge faster than reward schemes $(r_c, r_l, r_s) = (-10, 10, 100)$ and $(r_c, r_l, r_s) = (-1, 1, 2)$. The latter two show a slower convergence, this could be due to the fact that $\frac{r_c}{r_l} = -1$. Therefore, the agent will receive a negative reward at the end of earlier game episodes because it takes many actions to solve the syndrome problem and reach a terminal board state. However, reward scheme $(r_c, r_l, r_s) = (-1, 1, 10)$ also has this ratio $\frac{r_c}{r_l} = -1$, but because r_s is a higher positive number, the few successes in early training stages result in parameter optimization of the agent such that convergence happens faster. Together with $(r_c, r_l, r_s) = (-1, 1, 2)$, $(r_c, r_l, r_s) = (-1, 5, 10)$ shows highest performance after 300.000 training timesteps, but the latter converges fastest compared to the other reward schemes. Therefore, this reward scheme will be set as default for the rest of this study and for training and evaluation on all other board sizes.

*The environment calls `env.step()` and returns a timestep reward and board state observation to the agent.

Version of March 1, 2024– Created March 1, 2024 - 18:58

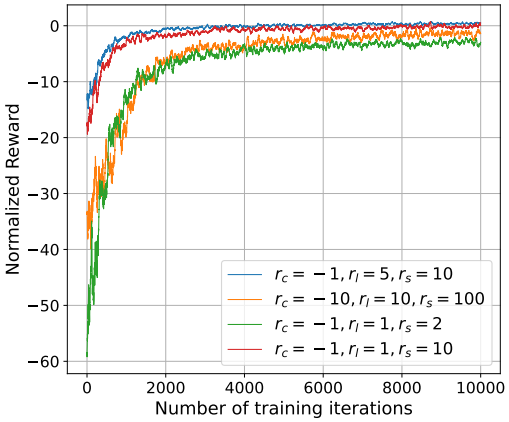


Figure 5.1: *Early training convergence of PPO agents trained on different reward schemes on a $d = 5$ board with error rate $p_{error} = 0.1$ for the first 10.000 training iterations.*

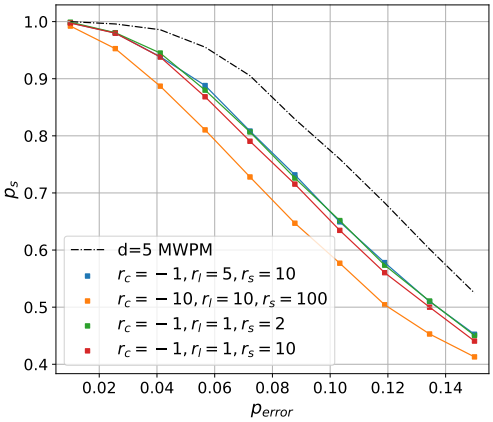


Figure 5.2: *Performance as success rate p_s after 300.000 training timesteps of PPO agents trained on different reward schemes on a $d = 5$ board, compared to a $d = 5$ MWPM decoder, evaluated over 10.000 games.*

The network architecture of the MLP, the learning rate and the entropy coefficient (the clipping parameter ϵ in Eq. 3.8) are tuned. The convergence at early training stages and performance evaluation of each of these hyperparameters can be found in Fig. A.1 - A.6 in the Appendix. The performance after 300.000 training timesteps is the highest for a network with 2 layers, each consisting of 64 nodes. The more complex network architectures consisting of 3 layers, i.e. $64 \times 64 \times 64$ and $20 \times 20 \times 20$, result in a lower success rate during evaluation, indicating that these networks might be overfitting. Furthermore, the learning rate convergence shows slow and unstable convergence for a learning rate of 0.01, resulting in a success rate much lower than for learning rates of 0.0001 and 0.001, shown in Fig. 5.3. From Fig. 5.4 it can be seen that the learning rate of 0.01 results in a very inefficient decoder, having to take many actions to reach the terminal state, and for cases of p_{error} above 0.1 it even reaches the maximum amount of steps allowed to clear a syndrome.

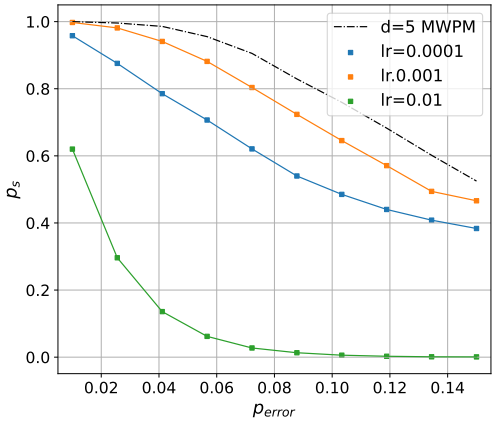


Figure 5.3: *Performance as success rate p_s after 300.000 training timesteps of PPO agents trained on different learning rate values on a $d = 5$ board, compared to a $d = 5$ MWPM decoder, evaluated over 10.000 games.*

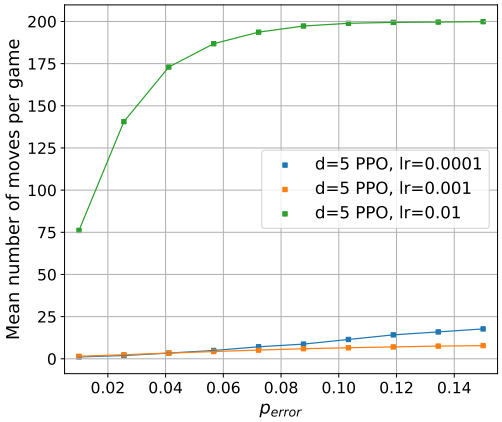


Figure 5.4: *Mean number of moves per evaluation game after 300.000 training timesteps of PPO agents trained on different learning rate values on a $d = 5$ board.*

The settings used to produce each of the figures referenced in this section can be found in Tab. A.1 - A.4 in the Appendix. The hyperparameters for the remainder of the training process can be found in Tab. 5.1.

Parameter	**Value**
Max steps per syndrome	200
Entropy coefficient	0.01
Clipping parameter	0.1
Learning rate	0.001
r_c, r_l, r_s	-1,5,10
Network Layers	2
Network Nodes	64 per layer
Batch Size	64
γ	0.99

Table 5.1: *Hyperparameter settings used for training and evaluation of the PPO agents on board sizes $d = 3, 5, 7$ on a static game framework.*

Using these settings, PPO agents are trained on examples of p_{error} for board sizes $d = 3, 5, 7$. Each agent will train for 10^6 timesteps and show to converge, as is shown in Fig. A.7 - A.9 in the Appendix. It is important to note that, if cases of empty boards or boards with already existing logical errors would be fed to the agent during training, the agent gets rewarded incorrectly. The already existing non-trivial loop is not visible because a logical error has no syndrome endpoints. The agent might reason that it will do a good job by removing the syndrome endpoints, since in all other training examples it was rewarded positively for doing so and it updated its parameters accordingly. However, in these exceptional cases, the agent is going to receive reward r_l instead of r_s, resulting in sub-optimal parameter optimization. Therefore, during the training process, the agent will only be shown game examples that always have syndrome endpoints present on the board, and do not already have a non-trivial error loop when fed to the agent as the initial board state.

5.1.2 Performance Evaluation

After training, the agents are evaluated on 10.000 game examples for each board size, and the success rate p_s is used as a performance measure and is benchmarked against the success rate of the MWPM decoders on the same game examples for each board size. In contrast to the training examples excluding boards without syndrome endpoints or having logical errors, the set of evaluation examples includes these cases, since it is important to evaluate the agent's strategy on those cases as well. Fig. 5.5 shows the success rate p_s for all PPO agents against bit-flip error rates p_{error} between 0.01 and 0.15. For $d = 3$ the PPO agent's success rate corresponds with that of the $d = 3$ MWPM decoder. For the larger board sizes $d = 5, 7$ this is no longer the case and these PPO agents show a success rate far below that of the MWPM decoder. No error threshold is identifiable from these results, whereas the error threshold for the MWPM lies at $p_{\text{th}} \approx 0.11$, corresponding with literature [12]. However, the learning curves from Fig. A.7 - A.9 in the Appendix do show that these agents all have converged during training. Especially for small error rates the MWPM decoders show almost perfect decoding, with success rates approaching or even being equal to 1.

Some of the most frequent cases of syndromes the agent fails to decode, but the MWPM decoder successfully decodes for $d = 5$ are shown in Fig. A.18-A.27 in the Appendix. In most of these cases, the agent chooses to perform more bit-flips than initially arose on the board and more than the MWPM selects as its solution. This contradicts the fact that the reward for

performing an action r_c is set to -1. Although this framework lends itself to a single-shot decoding strategy, in some cases, the first action the agent performs on the board results in the wrong solution, requiring it to correct for its own mistake. This eventually leads to a logical error, causing the agent to take more actions to decode than it could and should have.

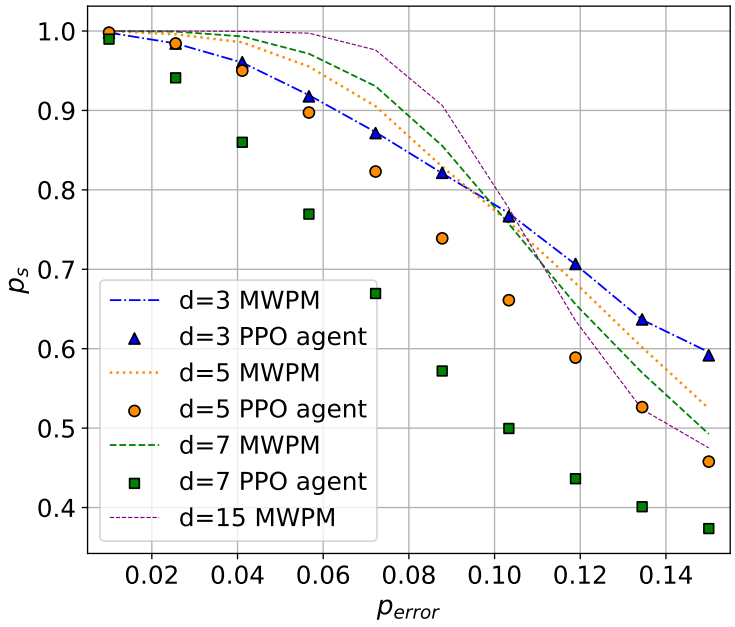


Figure 5.5: *Performance as success rate p_s after 10^6 training timesteps of converged PPO agents for board sizes $d = 3, 5, 7$ versus bit-flip error rate p_{error}. The results are compared to the corresponding MWPM performances. The $d = 15$ MWPM performance shows the approach to larger system sizes.*

Moreover, it is important to mention that for all three board sizes, the agents shown in Fig. 5.5 were trained on a fixed error rate of $p_{\text{error}} = 0.1$. The average number of bit-flip errors introduced by the environment during training are $1.8, 5$ and 9.8 for $d = 3, 5, 7$ respectively. This means that for $d = 5, 7$ agents, significantly fewer training examples have been encountered where only 1 or 2 bit-flip errors are introduced, which can result in an unrepresentative training set, potentially leading to decreased performance. Different sets of bit-flip errors on the board can lead to the same syndrome. For the $d = 5$ case for example, some syndromes can either be caused by 2 bit-flips, or by 3 bit-flips. If the agent has seen more 3 bit-flip scenarios, it will have optimized its policy parameters for 3 bit-flip

scenarios. Choosing to remove the syndrome by flipping 2 qubits on the board will actually result in a logical error, even though this would take less actions to decode. This scenario is shown in Fig. A.18 in the Appendix. Nevertheless, an agent trained using curriculum learning, where training has been done on instances of $p_{\text{error}} = [0.01, 0.038, 0.066, 0.094, 0.122, 0.15]$, each for 10^6 timesteps and using the updated set of parameters from the previous instance, does not show higher performance, as shown in Fig. A.13 in the Appendix. The training process takes six times longer compared to training on a fixed error rate.

Contrary to the result outcomes shown in this section, former studies have shown to succeed in training RL agents up to MWPM performance for the toric code with uncorrelated bit-flip noise using deep Q-learning with the network architecture of a CNN, identifying an error threshold corresponding with that of MWPM [22]. These agents were trained on a timestep reward of -1 until the terminal state was reached, regardless of whether the terminal state had led to a logical error or not. The result for an agent trained on this reward setup of $(r_c, r_l, r_s) = (-1, -1, -1)$ is shown for a $d = 5$ case in Fig. A.12 in the Appendix and does not reach MWPM performance using a MLP architecture with the PPO algorithm from Stable Baselines 3.

A comparison between the results from Andreasson et al. [22] and this research would have been ideal for translating the PPO algorithm to a DQN, preferably also replacing the MLP architecture with a CNN to assess whether the network used in this research might not be feasible to achieve optimal results, or Stable Baselines 3 rendering inaccurately trained policies. However, although Stable Baselines 3 provides a perfect platform for evaluating RL algorithms for customizable problems, translating from PPO to DQN is not straightforward. While the library easily allows for the implementation of an action mask for PPO, which masks out all invalid actions as described by Fig. 4.1 in Chapter 4, the implementation of DQN does not support this method. As a workaround, one could assign a reward with a very large negative value to all actions considered invalid. However, this approach results in a very large search space for the agent during training, where one of the sub-goals becomes learning to avoid choosing an invalid action, thereby slowing down training convergence, especially for larger system sizes [25].

Whereas the main focus of this study lies on finding the performance of PPO agents on uncorrelated bit-flip noise, where the MWPM decoder shows optimal performance for the decoding of uncorrelated bit-flip errors, the results for correlated bit-flip noise are shown in Fig. A.17 in the Appendix. Both MWPM and the PPO agents fail to show a proper er-

ror threshold for correlated bit-flip noise. For $d = 3$, the agent perfectly matches the MWPM decoding performance. For $d = 5, 7$ this is no longer the case.

5.2 Dynamic Framework

The agents to be trained on the dynamic framework have as their goal to maximize their score when playing the decoding game. The score is defined as the number of timesteps it can endure on the board without encountering a logical error. One of the sub-goals for the agents is to learn a strategy such that it knows when to perform no action on any of the qubits when the board is empty. For this framework the agents will only be trained on a $d = 3$ board. First, the scores for different reward scheme setups are evaluated for $d = 3$ and are benchmarked against a non-trained agent, which will indicate a random action selection. Next, the effect of different values for the learning rate are evaluated and the effect of increasing the iteration step parameter k and the amount of (new) errors on the board, N and N_{new} are evaluated and compared with each other, as well as compared with the MWPM decoder.

5.2.1 Training Setup

Since the goal for the agent is to survive for as long as possible on a board that introduces new errors every k timesteps, the rewards should not be negative for any of the actions the agent chooses to perform on the board. The agent will therefore be trained on one of the most basic reward schemes, which is $(r_c, r_e, r_l) = (1, 1, 1)$, implying that it gets +1 on its score for each timestep it manages to survive. However, since this reward setup might not effectively steer the agent towards its sub-goal of learning to keep the board clean and knowing when to 'do nothing', other reward schemes which reward the empty board state higher than other board states are evaluated as well. The agent's scores after playing 10.000 games for evaluation are shown in Fig. 5.6. The training convergence, in which the reward is normalized in order to make a valid comparison between all agents, is shown in Fig. B.1 in the Appendix. In Fig. 5.6, the maximum amount of timesteps the agents can reach per game during both training and evaluation is set to 300. It is visible that the reward scheme $(r_c, r_e, r_l) = (1, 1, 1)$ does not seem to reach performance slightly higher than random action selection. The other reward schemes reach higher scores. With $(N, N_{\text{new}}, k) = (1, 1, 2)$, the agent should be able to decode

the presented syndromes by flipping only one qubit in order to reach an empty board state. The next action it should select is then to 'do nothing'. Higher scores indicate that the agent has managed to learn this sub-goal.

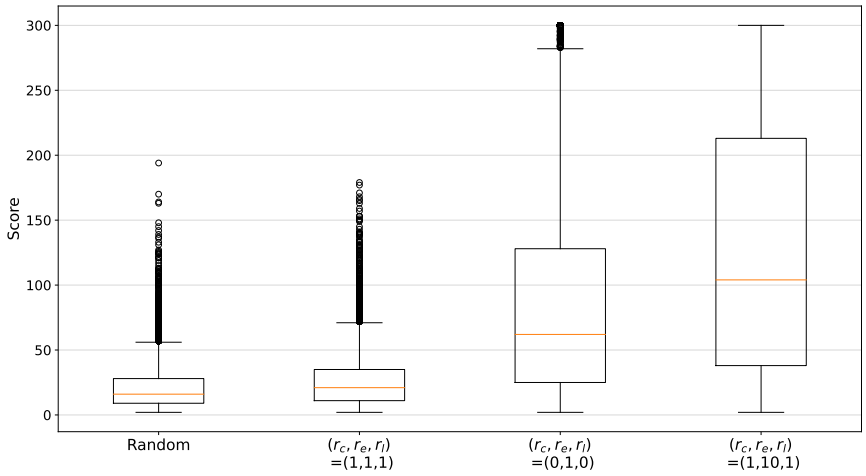


Figure 5.6: *Performance as score after* 10^6 *training timesteps of PPO agents trained on different reward schemes on a* $d = 3$ *board with* $(N, N_{new}, k) = (1, 1, 2)$*, evaluated over 10.000 games.*

The learning rate is investigated for $(r_c, r_e, r_l) = (1, 10, 1)$ and is evaluated for $0.0001, 0.001$ and 0.01, as well as learning rate annealing from 0.001 to 0.0001. Fig. 5.7 shows training convergence for each of the learning rate settings over a training period of $3 \cdot 10^6$ training timesteps, where each plotted training iteration corresponds with one game episode reaching the terminal board state. This is why not all learning curves have the same length; the agents that do well and reach a higher score can play the game for longer and thus include more training timesteps per iteration. For learning rate annealing, the agent reaches highest performance and fastest convergence, which corresponds with it having the shortest learning curve. It is notable that for a learning rate of 0.01 the reward first goes up, and then goes back down again after roughly the 7500th training iteration. This behaviour might be due to the fact that this learning rate is too big to find an optimal solution for the decoder, causing the weight updates to be too large, which results in a decrease in performance. Therefore, learning rate annealing will only be done between 0.001 and 0.0001. Fig. 5.8 shows the scores for each of the agents and shows that learning

rate annealing in this range results in the highest score as compared to training only on fixed learning rate values. The settings used to produce each of the figures referenced in this section can be found in Tab. B.1 - B.2 in the Appendix. The hyperparameters for the remainder of the training process are shown in Tab. 5.2.

Changes in N, N_{new}, k can influence the score in general, and this can be used as a measure to assess the capability of and RL agent to solve the decoding game in the dynamic framework. The RL agents are trained on $N, N_{\text{new}} \in [1,2]$, whereby $k \in [N+1, N+2]$. Because k introduces a measure on how much the agent is allowed to correct for its own mistakes, the agent's performance is investigated both for scenarios where it is allowed to make a mistake and still will be able to fully resolve the syndrome in the next timestep ($k = (N+2)$), and for scenarios in which it is not allowed to do so before new errors are introduced ($k = (N+1)$). Their scores are visualised in Fig. 5.9 for comparison. The training convergence for each of the agents can be found in Fig. B.3 - B.6 in the Appendix.

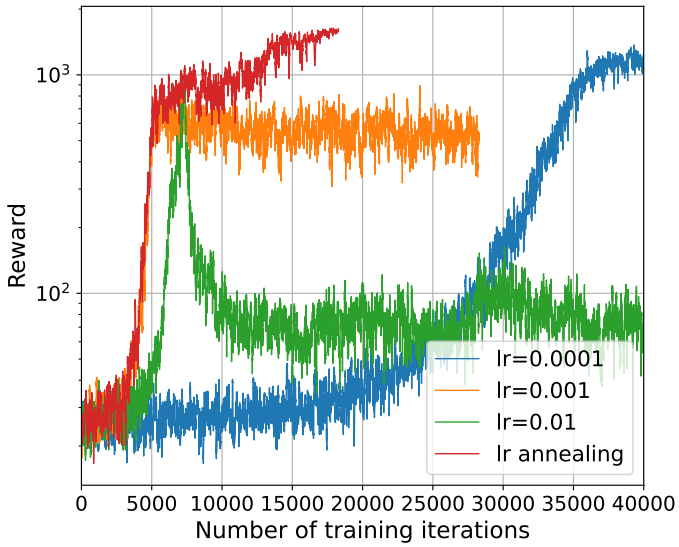


Figure 5.7: *Early training convergence of PPO agents trained on different learning rates on a $d = 3$ board with error rate $(N, N_{new}, k) = (1,1,2)$ for the first 10.000 training iterations.*

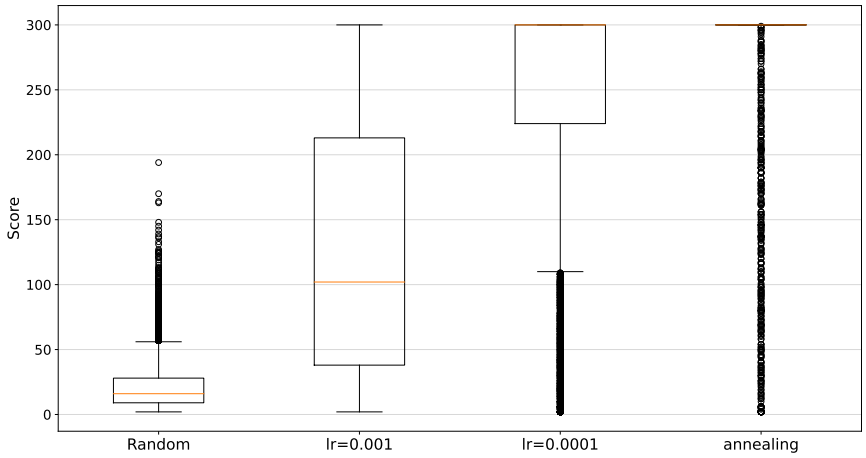


Figure 5.8: *Performance as score after* $3 \cdot 10^6$ *training timesteps of PPO agents trained on different learning rates on a* $d = 3$ *board with* $(N, N_{new}, k) = (1, 1, 2)$*, evaluated over 10.000 games.*

Parameter	**Value**
Max steps per game	300
Entropy coefficient	0.05
Clipping parameter	0.1
Learning rate	annealing from 0.001 to 0.0001
r_c, r_e, r_l	1,10,1
Network Layers	2
Network Nodes	64 per layer
Batch Size	64
γ	0.99

Table 5.2: *Hyperparameter settings used for training and evaluation of the PPO agents on board sizes* $d = 3$ *on a dynamic game framework.*

Fig. 5.9 visualizes that for $d = 3$ all agents except for $(N, N_{\text{new}}, k) = (2, 2, 3)$ show to on average reach the maximum score of 300. The agent fails to find an optimal decoding strategy when N, N_{new} gets larger, but k does not allow for correction of mistakes during training. In this case, the agent fails to escape from following a nearly random strategy as can be seen in Fig. B.5 in the Appendix, probably because it makes too many mis-

takes at the beginning of each game, and the game terminates too quickly for it to escape from those situations.

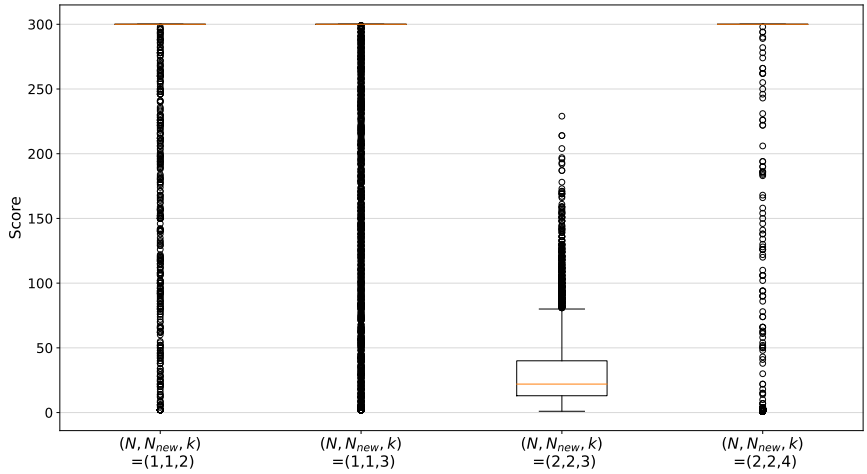


Figure 5.9: *Performance as score of converged PPO agents trained for different settings of* (N, N_{new}, k) *on a* $d = 3$ *board, evaluated over 10.000 games.*

5.2.2 Performance Evaluation

For the three agents that reached training convergence and the maximum possible game score of 300, their trained models are evaluated on examples using the static game environment in order to benchmark their performances against MWPM. Note that in order to carry out this type of evaluation, the action space from the dynamic environment, which is of size $(2d \times d) + 1$, is first translated to the action space of size $2d \times d$ for the static environment. This is done by removing the option for the agent to do nothing; it must therefore always choose a qubit to flip as its action. The results are shown in Fig. 5.10. This plot shows that for $(N, N_{\text{new}}, k) = (2, 2, 4)$ the agent is able to reach MWPM performance. Referring to the learning curve in Fig. B.6, this agent encountered two performance jumps during training. The first jump probably accounts for the agent learning how to survive on the board in general, and the second jump for the fact that it discovers it gets a higher reward for keeping the board as empty as possible. This agent shows to find a strategy similar to that of MWPM, where it decodes syndromes as efficiently as possible. The other two agents do not reach performance up to MWPM as they are trained on single bit-flip cases only, and therefore these examples do not provide the

agent with enough training information about how syndrome endpoints will shift across the board when more than one error has occurred.

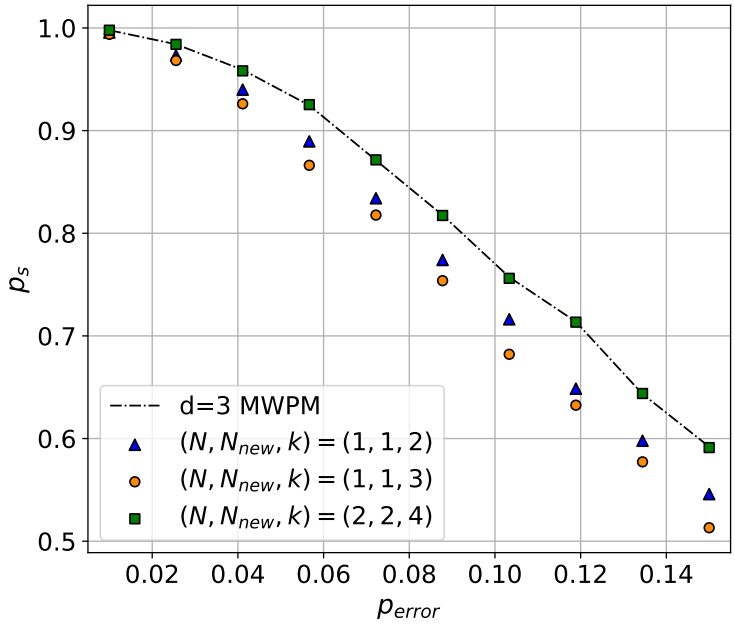


Figure 5.10: *Performance converged PPO agents trained for different settings of* (N, N_{new}, k) *on a* $d = 3$ *board, evaluated over 10.000 games on the static game framework, and benchmarked against a* $d = 3$ *MWPM decoder.*

Chapter 6

Discussion and Outlook

This research investigated the performance of RL agents using Stable Baselines 3's PPO algorithm with an MLP network architecture. Each agent was trained on toric code decoding problems in order to find an optimal error correcting strategy. Two game frameworks are distinguished from each other, whereby this study provides the implementation of a novel dynamic game framework, for which the RL performance can be measured with a survival score and which can eventually be translated to a multi-agent game framework. An implementation for a check on logical errors for the toric code is constructed such that it works for both frameworks, and can be directly used for the training and evaluation of decoding agents.

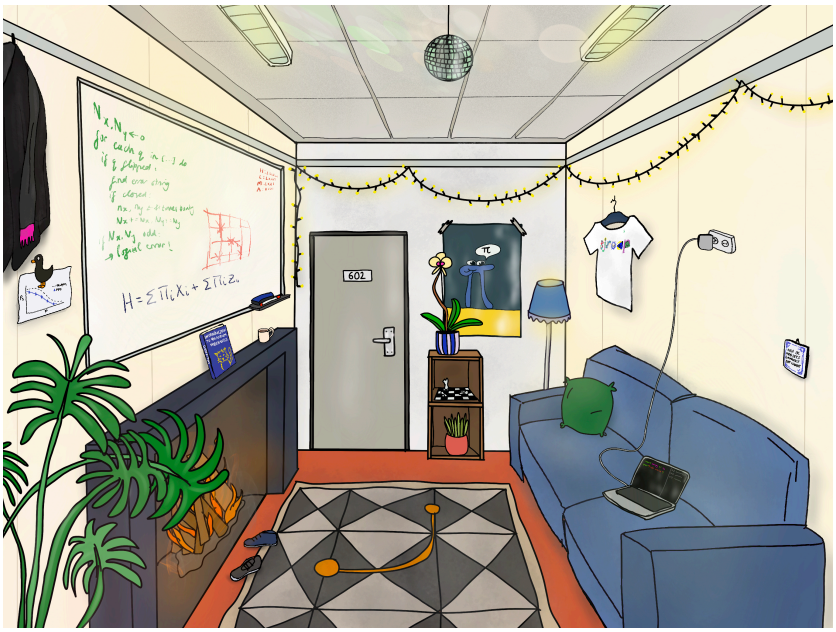
For the static game framework the trained RL agents reach MWPM performance on a $d = 3$ board with uncorrelated bit-flip noise, but for larger system sizes $d = 5, 7$ the agents do not reach this performance. No error threshold could be identified for the PPO decoding agents, whereas other studies show proof of concept that RL is capable of finding an error threshold on this framework [13, 22, 23]. Moreover, Andreasson et al. [22] demonstrate, using more sophisticated network architectures including a CNN and other RL techniques such as deep Q-learning, that RL is able to achieve MWPM performance for uncorrelated bit-flip noise, which indicates that the results presented in this thesis could be successfully improved. However, Stable Baselines 3 does not allow for straightforward translation from PPO to DQN due to differences in the implementation of an action mask. Other libraries should be assayed for this purpose in order to avoid a workaround. Furthermore, both Andreasson et al. [22] and Theveniaut et al. [13] use the relative positions of all syndrome endpoints of a given state observation as an input, called perspectives, in order to

build in rotation invariance. This allows for a decoder trained on a small lattice to transfer to a larger lattice and update weight parameters based on these perspectives rather than all different rotational examples of the same syndrome. Implementing these perspectives into the current training setup for the static framework could therefore be interesting to evaluate. Beyond the scope of this project, but worthwhile to investigate in further research, is the performance of RL agents trained on depolarizing noise.

For the dynamic game framework the RL agents show proof of concept that they are able to learn using the setup of a survival-like game. The agents were trained and evaluated on a $d = 3$ board for different settings of N, N_{new}, k and show that for agent trained on $(N, N_{\text{new}}, k) = (2, 2, 4)$ it was able to reach MWPM performance. Improvements upon this study for the dynamic framework can be done by adjusting the network architecture, training and evaluating on larger system sizes, and tuning other hyperparameters such as the gamma factor; since the sequence of actions is more important in this framework than it is for the static framework, this factor is important to be tuned in order to increase the performance. Furthermore, one could implement frame stacking such that former state observations on a sequence of decoding steps are stored and the agent has access to those former frames when optimizing its policy. This is especially important when extending this framework towards a multi-agent framework, where one agent has to introduce errors on the board, and another agent's goal is to decode the presented syndromes.

Acknowledgements

Being able to familiarize myself with doing research and working on this subject, which struck my interest from the very beginning, has been a real pleasure and therefore I would like to give special thanks Evert van Nieuwenburg for being a very helpful, thoughtful and inspiring supervisor. I really appreciate the effort and guidance he offered me during our meetings, as well as driving me to discuss my project and findings with my peers in our research group. Furthermore, I would like to thank Aske Plaat for being my second supervisor and providing me with fresh insights and motivating discussions. I would also like to express my gratitude towards all members from the Condensed-AI department at the aQa research group, for our discussions, code-meetings, group meetings and knowledge sharing, as well as the fun coffee- and lunch breaks. The group felt like a safe environment in which everyone is encouraged to ask questions and help each other out. Lastly, I would be remiss in not mentioning Alek, Hans and Matthijs, my office buddies at HL602, for striking the best balance ever between seriousness and fun. Doing research in the office wouldn't have been so productive yet so enjoyable without you.



Bibliography

[1] Daniel Gottesman. *An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation*. Apr. 16, 2009. arXiv: 0904.2557 [quant-ph]. URL: http://arxiv.org/abs/0904.2557 (visited on 10/01/2023).

[2] Joschka Roffe. "Quantum Error Correction: An Introductory Guide". In: *Contemporary Physics* 60.3 (July 3, 2019), pp. 226–245. ISSN: 0010-7514, 1366-5812. DOI: 10.1080/00107514.2019.1667078. arXiv: 1907.11157 [quant-ph]. URL: http://arxiv.org/abs/1907.11157 (visited on 10/29/2023).

[3] A. Yu Kitaev. "Fault-tolerant quantum computation by anyons". In: *Annals of Physics* 303.1 (Jan. 2003), pp. 2–30. ISSN: 00034916. DOI: 10.1016/S0003-4916(02)00018-0. arXiv: quant-ph/9707021. URL: http://arxiv.org/abs/quant-ph/9707021 (visited on 02/24/2024).

[4] Vladimir Kolmogorov. "Blossom V: a new implementation of a minimum cost perfect matching algorithm". In: *Mathematical Programming Computation* 1.1 (July 1, 2009), pp. 43–67. ISSN: 1867-2957. DOI: 10.1007/s12532-009-0002-8. URL: https://doi.org/10.1007/s12532-009-0002-8 (visited on 09/13/2023).

[5] Austin G. Fowler. *Minimum weight perfect matching of fault-tolerant topological quantum error correction in average $O(1)$ parallel time*. Oct. 10, 2014. arXiv: 1307.1740 [quant-ph]. URL: http://arxiv.org/abs/1307.1740 (visited on 01/16/2024).

[6] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (Oct. 2017), pp. 354–359. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature24270. URL: https://www.nature.com/articles/nature24270 (visited on 02/24/2024).

[7] Dr James Wootton. *Even You Can Help Build a Quantum Computer*. sci five — University of Basel. Nov. 8, 2016. URL: https://medium.com/sci-five-university-of-basel/even-you-can-help-build-a-quantum-computer-34275b5ab716 (visited on 02/24/2024).

[8] R.W. Hamming. "Error detecting and error correcting codes". In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[9] Emanuel Knill and Raymond Laflamme. "A Theory of Quantum Error-Correcting Codes". In: *Physical Review Letters* 84.11 (Mar. 13, 2000), pp. 2525–2528. ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.84.2525. arXiv: quant-ph/9604034. URL: http://arxiv.org/abs/quant-ph/9604034 (visited on 12/08/2023).

[10] W. K. Wootters and W. H. Zurek. "A single quantum cannot be cloned". In: *Nature* 299.5886 (Oct. 1982), pp. 802–803. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/299802a0. URL: https://www.nature.com/articles/299802a0 (visited on 12/08/2023).

[11] H. Bombin. *An Introduction to Topological Quantum Codes*. Nov. 1, 2013. arXiv: 1311.0277[quant-ph]. URL: http://arxiv.org/abs/1311.0277 (visited on 09/22/2023).

[12] Eric Dennis et al. "Topological quantum memory". In: *Journal of Mathematical Physics* 43.9 (Sept. 1, 2002), pp. 4452–4505. ISSN: 0022-2488, 1089-7658. DOI: 10.1063/1.1499754. URL: https://pubs.aip.org/jmp/article/43/9/4452/230976/Topological-quantum-memory (visited on 09/20/2023).

[13] Hugo Theveniaut and Evert van Nieuwenburg. "A NEAT Quantum Error Decoder". In: *SciPost Physics* 11.1 (July 12, 2021), p. 005. ISSN: 2542-4653. DOI: 10.21468/SciPostPhys.11.1.005. arXiv: 2101.08093[physics,physics:quant-ph]. URL: http://arxiv.org/abs/2101.08093 (visited on 09/12/2023).

[14] P Herringer. "The Toric Code". In: (2020). URL: https://www.physics.rutgers.edu/grad/602/Lectures/JC_Presentations/0419/Intro_Toric_Code.pdf.

[15] Nando Leijenhorst. "Quantum Error Correction: Decoders for the Toric Code". In: (2019). URL: https://repository.tudelft.nl/islandora/object/uuid%3A94823249-e114-4fc0-bae9-4c80d503296f (visited on 02/27/2024).

[16] Jack Edmonds. "Paths, Trees, and Flowers". In: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467. ISSN: 0008-414X, 1496-4279. DOI: 10.4153/CJM-1965-045-4. URL: https://www.cambridge.org/core/product/identifier/S0008414X00039419/type/journal_article (visited on 09/13/2023).

[17] Oscar Higgott. *PyMatching: A Python package for decoding quantum codes with minimum-weight perfect matching*. July 12, 2021. DOI: 10.48550/arXiv.2105.13082. arXiv: 2105.13082[quant-ph]. URL: http://arxiv.org/abs/2105.13082 (visited on 09/12/2023).

[18] Sergey Bravyi, Martin Suchara, and Alexander Vargo. "Efficient algorithms for maximum likelihood decoding in the surface code". In: *Physical Review A* 90.3 (Sept. 25, 2014), p. 032326. ISSN: 1050-2947, 1094-1622. DOI: 10.1103/PhysRevA.90.032326. URL: https://link.aps.org/doi/10.1103/PhysRevA.90.032326 (visited on 01/15/2024).

[19] Richard S. Sutton and Andrew Barto. *Reinforcement learning: an introduction*. Nachdruck. Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2014. 322 pp. ISBN: 978-0-262-19398-6.

[20] John Schulman et al. *Proximal Policy Optimization Algorithms*. Aug. 28, 2017. arXiv: 1707.06347[cs]. URL: http://arxiv.org/abs/1707.06347 (visited on 10/29/2023).

[21] Ryan Sweke et al. "Reinforcement Learning Decoders for Fault-Tolerant Quantum Computation". In: *Machine Learning: Science and Technology* 2.2 (June 1, 2021), p. 025005. ISSN: 2632-2153. DOI: 10.1088/2632-2153/abc609. arXiv: 1810.07207[quant-ph]. URL: http://arxiv.org/abs/1810.07207 (visited on 10/29/2023).

[22] Philip Andreasson et al. "Quantum error correction for the toric code using deep reinforcement learning". In: *Quantum* 3 (Sept. 2, 2019), p. 183. ISSN: 2521-327X. DOI: 10.22331/q-2019-09-02-183. arXiv: 1811.12338[cond-mat,physics:quant-ph]. URL: http://arxiv.org/abs/1811.12338 (visited on 09/13/2023).

[23] Laia Domingo Colomer, Michalis Skotiniotis, and Ramon MuÃ±oz Tapia. "Reinforcement learning for optimal error correction of toric codes". In: *Physics Letters A* 384.17 (June 15, 2020), p. 126353. ISSN: 0375-9601. DOI: 10.1016/j.physleta.2020.126353. URL: https://www.sciencedirect.com/science/article/pii/S0375960120301638 (visited on 09/13/2023).

[24] Greg Brockman et al. *OpenAI Gym*. June 5, 2016. arXiv: 1606.01540[cs]. URL: http://arxiv.org/abs/1606.01540 (visited on 02/04/2024).

[25] Shengyi Huang and Santiago Ontanon. "A Closer Look at Invalid Action Masking in Policy Gradient Algorithms". In: *The International FLAIRS Conference Proceedings* 35 (May 4, 2022). ISSN: 2334-0762. DOI: 10.32473/flairs.v35i.130584. arXiv: 2006.14171[cs,stat]. URL: http://arxiv.org/abs/2006.14171 (visited on 10/10/2023).

[26] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: ().

[27] L. Spoor. "RL on the toric code". In: (). URL: https://github.com/lindsayspoor/RL_on_toric_code.

Appendix A

Static Framework

A.1 Hyperparameter Settings

Parameter	Value
Max steps per syndrome	200
Entropy coefficient	0.01
Clipping parameter	0.1
Learning rate	0.001
Network Layers	2
Network Nodes	64 per layer
Batch Size	64
γ	0.99

Table A.1: *Hyperparameter settings for the ablation study of the reward schemes for the $d = 5$ PPO agent.*

Parameter	Value
Max steps per syndrome	200
Entropy coefficient	0.05
Clipping parameter	0.1
r_c, r_l, r_s	-1,5,10
Learning rate	0.001
Batch Size	64
γ	0.99

Table A.2: *Hyperparameter settings for the ablation study of the network architecture for the $d = 5$ PPO agent.*

Version of March 1, 2024– Created March 1, 2024 - 18:58

Parameter	Value
Max steps per syndrome	200
Entropy coefficient	0.05
Clipping parameter	0.1
r_c, r_l, r_s	-1,5,10
Network Layers	2
Network Nodes	64 per layer
Batch Size	64
γ	0.99

Table A.3: *Hyperparameter settings for the ablation study of the learning rates for the $d = 5$ PPO agent.*

Parameter	Value
Max steps per syndrome	200
Clipping parameter	0.1
r_c, r_l, r_s	-1,5,10
Network Layers	2
Network Nodes	64 per layer
Learning rate	0.001
Batch Size	64
γ	0.99

Table A.4: *Hyperparameter settings for the ablation study of the entropy coefficient for the $d = 5$ PPO agent.*

Parameter	Value
Max steps per syndrome	200
Entropy coefficient	0.01
Clipping parameter	0.1
Learning rate	0.001
r_c, r_l, r_s	-1,5,10
Network Layers	2
Network Nodes	64 per layer
Batch Size	64
γ	0.99

Table A.5: *Hyperparameter settings used for training and evaluation of the PPO agents on board sizes $d = 3, 5, 7$ on a static game framework, both for correlated and uncorrelated bit-flip noise, and for curriculum learning.*

Parameter	Value
Max steps per syndrome	200
Entropy coefficient	0.01
Clipping parameter	0.1
Learning rate	0.001
r_c, r_l, r_s	-1,-1,-1
Network Layers	2
Network Nodes	64 per layer
Batch Size	64
γ	0.99

Table A.6: *Hyperparameter settings used for training and evaluation of the PPO agents on board sizes $d = 3, 5, 7$ on a static game framework on a reward scheme of $(r_c, r_s, r_l) = (-1, -1, -1)$.*

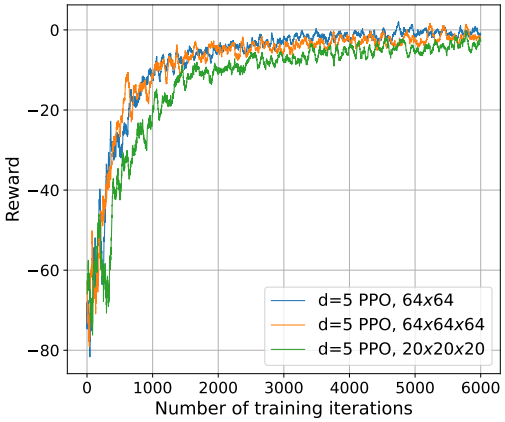


Figure A.1: *Early training convergence of PPO agents trained on different network architectures on a $d = 5$ board with error rate $p_{\text{error}} = 0.1$ for the first 6000 training iterations.*

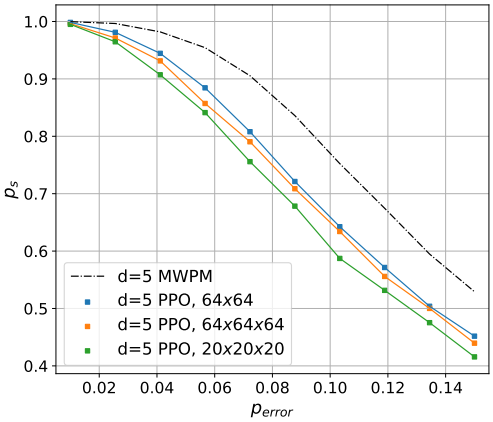


Figure A.2: *Performance as success rate p_s after 300.000 training timesteps of PPO agents trained on different network architectures on a $d = 5$ board, compared to a $d = 5$ MWPM decoder, evaluated over 10.000 games.*

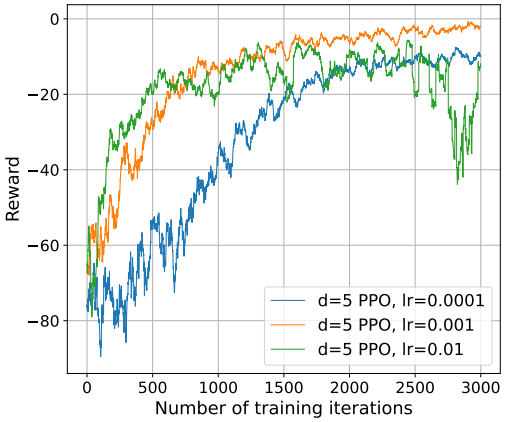


Figure A.3: *Early training convergence of PPO agents trained on different learning rate values on a $d = 5$ board with error rate $p_{\text{error}} = 0.1$ for the first 3000 training iterations.*

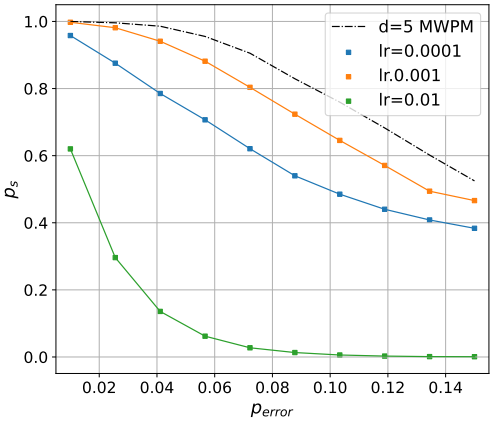


Figure A.4: *Performance as success rate p_s after 300.000 training timesteps of PPO agents trained on different learning rate values on a $d = 5$ board, compared to a $d = 5$ MWPM decoder, evaluated over 10.000 games.*

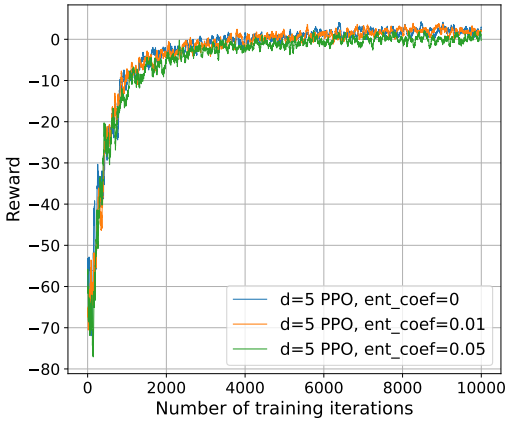


Figure A.5: *Early training convergence of PPO agents trained on different entropy coefficient values on a $d = 5$ board with error rate $p_{\text{error}} = 0.1$ for the first 20000 training iterations.*

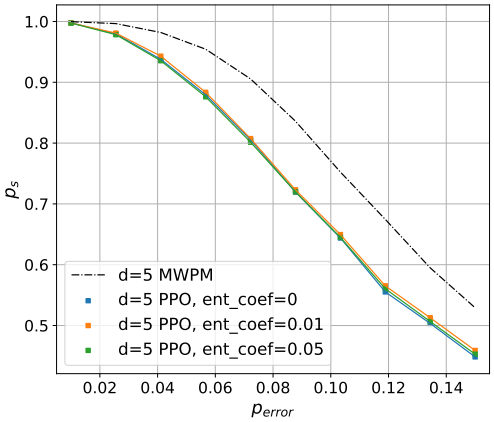


Figure A.6: *Performance as success rate p_s after 300.000 training timesteps of PPO agents trained on different entropy coefficient values on a $d = 5$ board, compared to a $d = 5$ MWPM decoder, evaluated over 10.000 games.*

A.2 Experiments on PPO

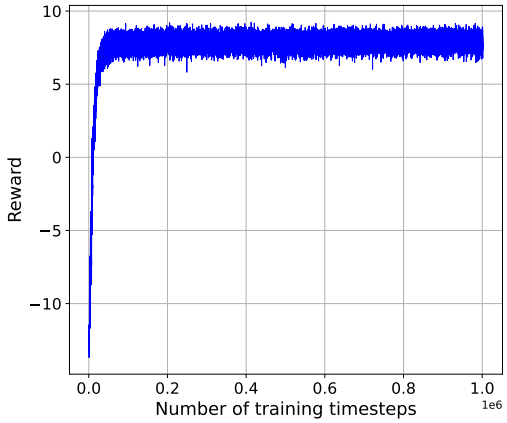


Figure A.7: *Training convergence for a PPO agent over 10^6 training timesteps on a $d = 3$ board. The line is smoothed by showing the moving average of the timestep reward over a window of 50 timesteps.*

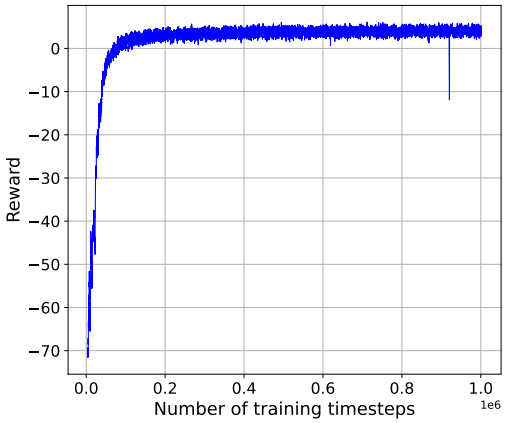


Figure A.8: *Training convergence for a PPO agent over 10^6 training timesteps on a $d = 5$ board. The line is smoothed by showing the moving average of the timestep reward over a window of 50 timesteps.*

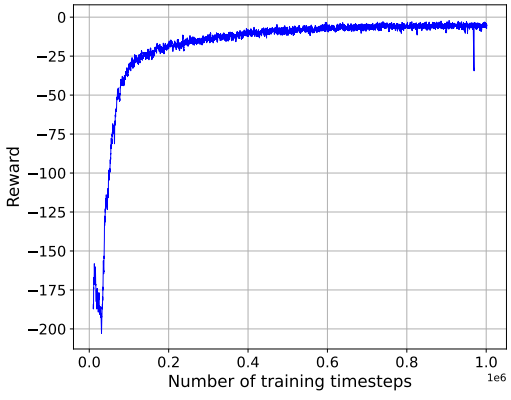


Figure A.9: *Training convergence for a PPO agent over 10^6 training timesteps on a $d = 7$ board. The line is smoothed by showing the moving average of the timestep reward over a window of 50 timesteps.*

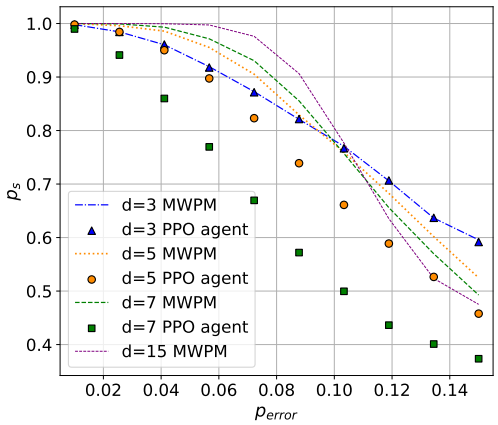


Figure A.10: *Performance as success rate p_s after 10^6 training timesteps of converged PPO agents for board sizes $d = 3, 5, 7$ versus bit-flip error rate p_{error}.*

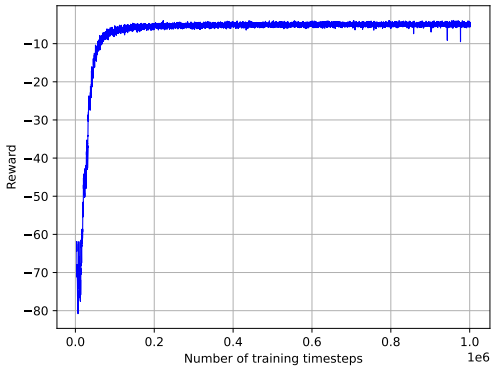


Figure A.11: *Training convergence for a PPO agent over 10^6 training timesteps on a $d = 5$ board, using reward scheme $(r_c, r_l, r_s) = (-1, -1, -1)$. The line is smoothed by showing the moving average of the timestep reward over a window of 50 timesteps.*

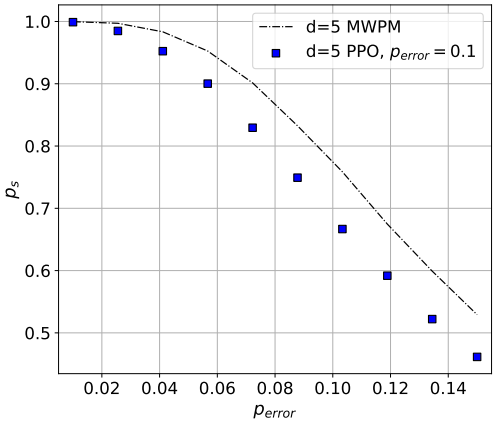


Figure A.12: *Performance as success rate p_s after 10^6 training timesteps for a converged PPO agent on board size $d = 5$ versus bit-flip error rate p_{error}, using reward scheme $(r_c, r_l, r_s) = (-1, -1, -1)$.*

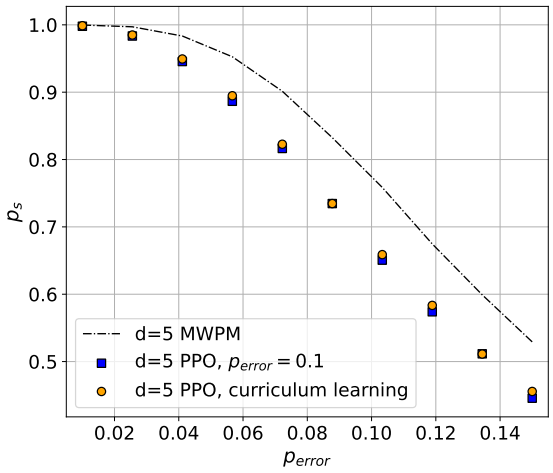


Figure A.13: *Performance as success rate p_s for converged PPO agents on board size $d = 5$ versus bit-flip error rate p_{error}. The agent trained with curriculum learning has trained on error rates $[0.01, 0.038, 0.066, 0.094, 0.122, 0.15]$, each for 10^6 training timesteps.*

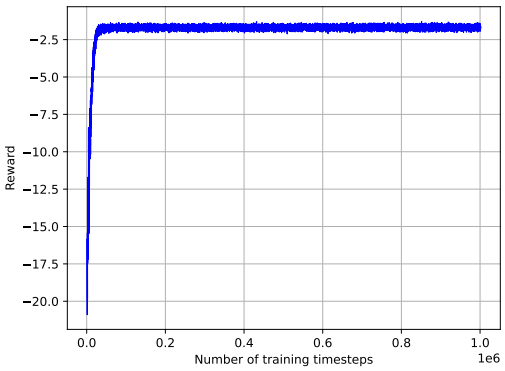


Figure A.14: *Training convergence for a PPO agent over 10^6 training timesteps on a $d = 3$ board for correlated bit-flip errors. The line is smoothed by showing the moving average of the timestep reward over a window of 50 timesteps.*

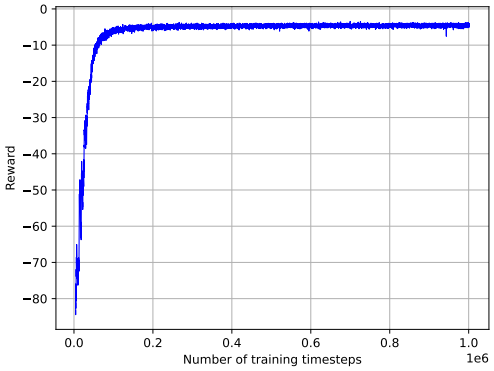


Figure A.15: *Training convergence for a PPO agent over 10^6 training timesteps on a $d = 5$ board for correlated bit-flip errors. The line is smoothed by showing the moving average of the timestep reward over a window of 50 timesteps.*

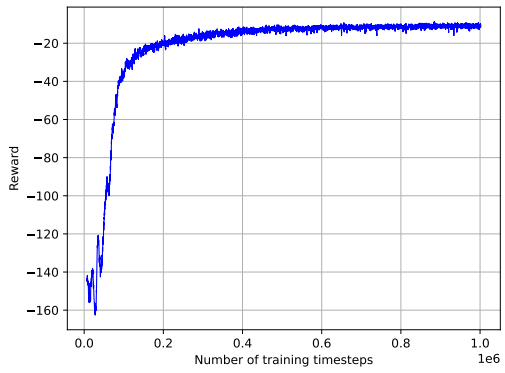


Figure A.16: *Training convergence for a PPO agent over 10^6 training timesteps on a $d = 7$ board for correlated bit-flip errors. The line is smoothed by showing the moving average of the timestep reward over a window of 50 timesteps.*

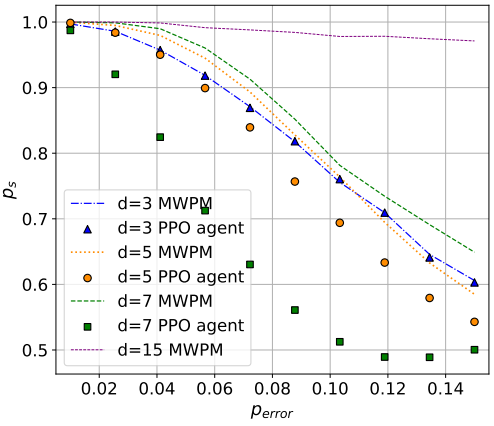


Figure A.17: *Performance as success rate p_s after 10^6 training timesteps of converged PPO agents for board sizes $d = 3, 5, 7$ versus bit-flip error rate p_{error} for correlated bit-flip errors. The results are compared to the corresponding MWPM performances. The $d = 15$ MWPM performance shows the approach to larger system sizes.*

A.3 Examples of failed decoding cases

Fig. A.18-A.27 show rendered examples of cases in which a $d = 5$ PPO agent with the performance shown in Fig. 5.5 fails to decode the presented syndrome. Each plot shows the initial qubits flipped by the environment, the flips from the agent, and the flips from the MWPM decoder. In each of the shown examples, the MWPM decoding actions led to a success, whereas the agent failed.

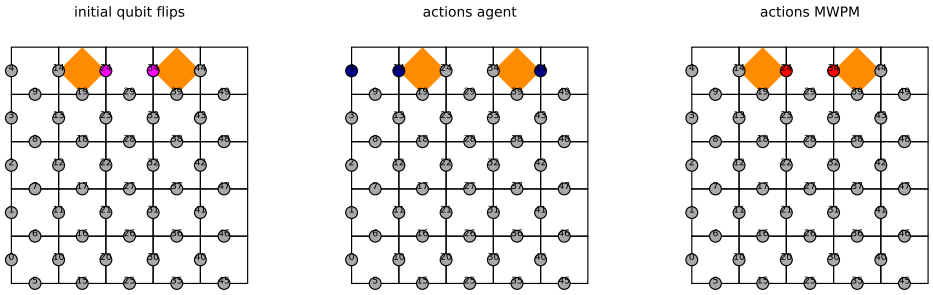


Figure A.18

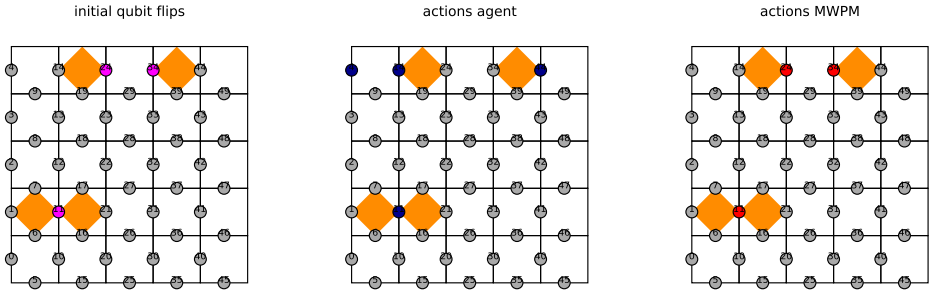


Figure A.19

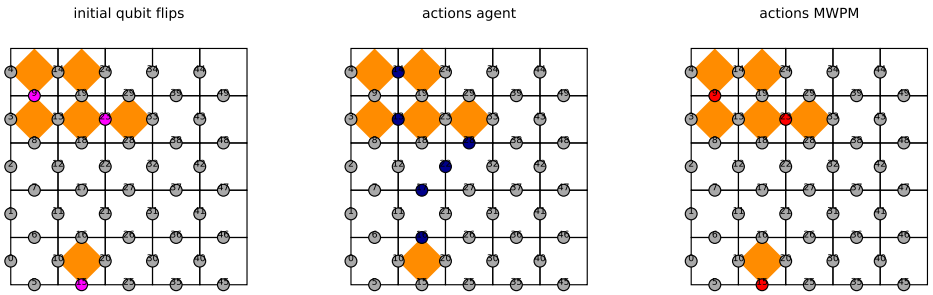


Figure A.20

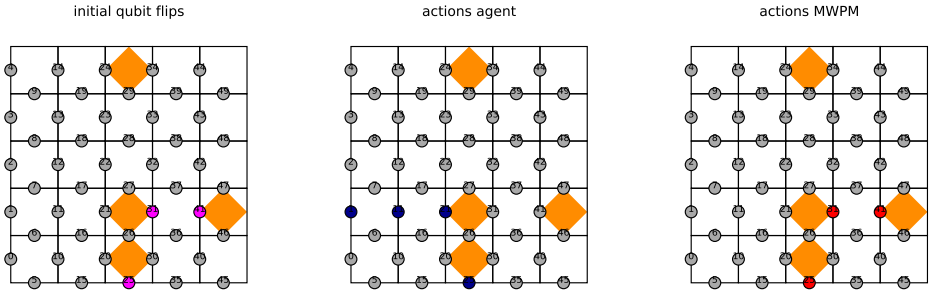


Figure A.21

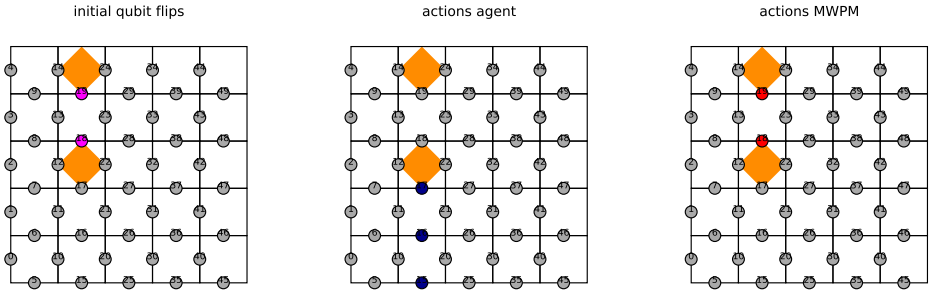


Figure A.22

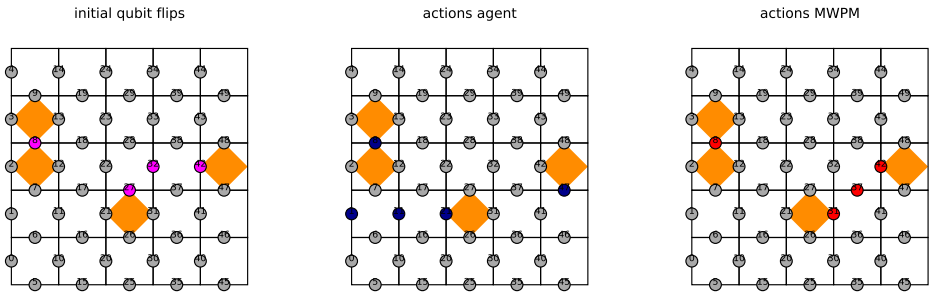


Figure A.23

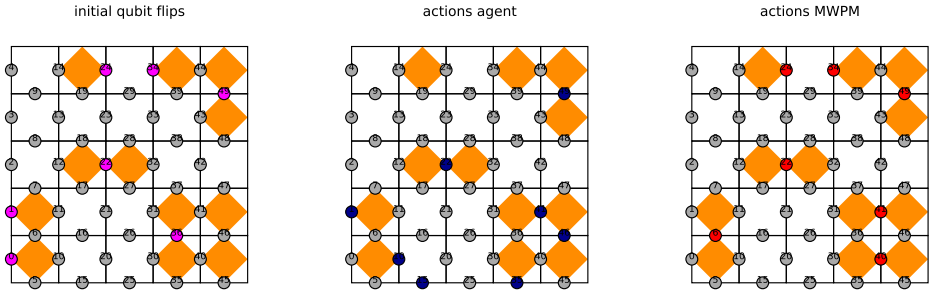


Figure A.24

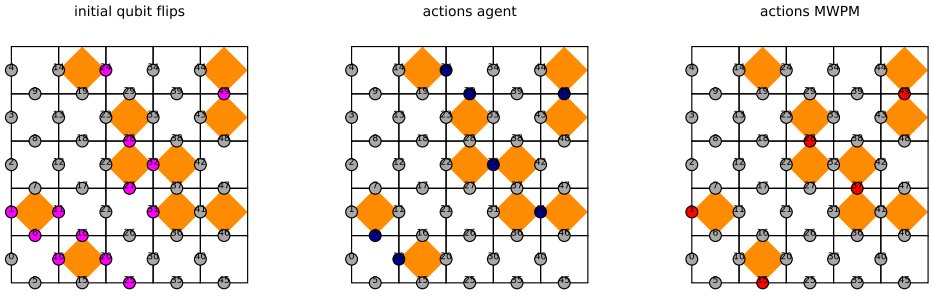


Figure A.25

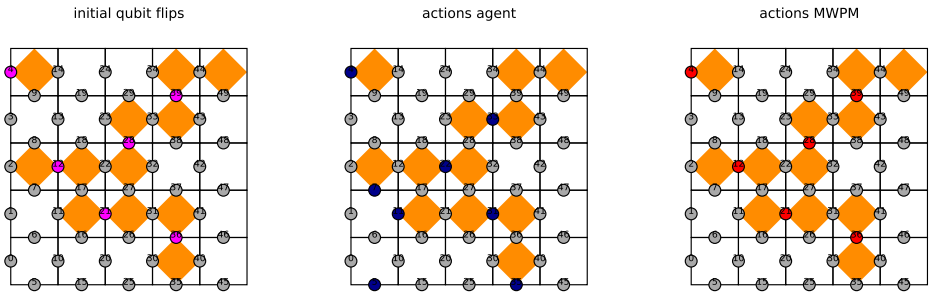


Figure A.26

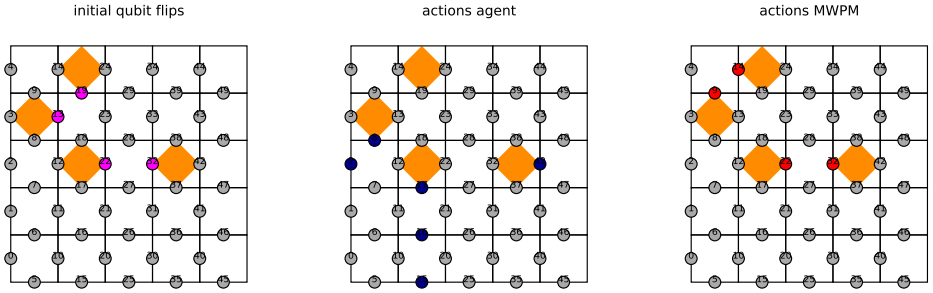


Figure A.27

Appendix B

Dynamic Framework

B.1 Hyperparameter Settings

Parameter	Value
Max steps per syndrome	300
Entropy coefficient	0.05
Clipping parameter	0.1
Learning rate	0.001
N, N_{new}, k	1,1,2
Network Layers	2
Network Nodes	64 per layer
Batch Size	64
γ	0.99

Table B.1: *Hyperparameter settings for the ablation study of the reward schemes for the $d = 3$ PPO agent.*

Parameter	Value
Max steps per syndrome	300
Entropy coefficient	0.05
Clipping parameter	0.1
r_c, r_e, r_l	1,10,1
N, N_{new}, k	1,1,2
Network Layers	2
Network Nodes	64 per layer
Batch Size	64
γ	0.99

Table B.2: *Hyperparameter settings for the ablation study of the learning rate for the $d = 3$ PPO agent.*

Version of March 1, 2024– Created March 1, 2024 - 18:58

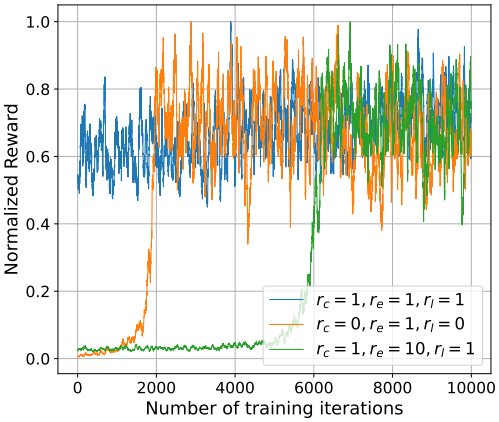


Figure B.1: *Early training convergence of PPO agents trained on different reward schemes on a $d = 3$ board with error rate $(N, N_{new}, k) = (1, 1, 2)$ for the first 10.000 training iterations.*

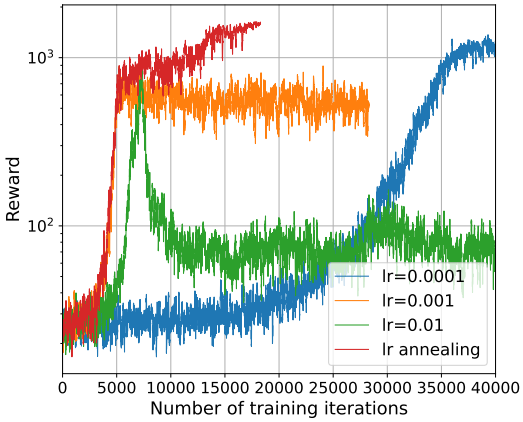


Figure B.2: *Early training convergence of PPO agents trained on different reward schemes on a $d = 3$ board with error rate $(N, N_{new}, k) = (1, 1, 2)$ for the first 40.000 training iterations.*

B.2 Experiments on different values of $N, N_{\mathbf{new}}, k$

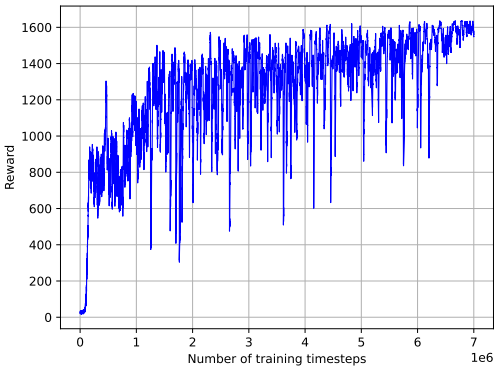


Figure B.3: *Training convergence for a PPO agent over $7 \cdot 10^6$ training timesteps on a $d = 3$ board for $(N, N_{new}, k) = (1, 1, 2)$. The line is smoothed by showing the moving average of the timestep reward over a window of 50 timesteps.*

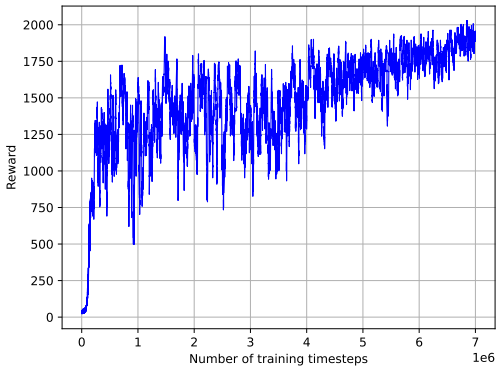


Figure B.4: *Training convergence for a PPO agent over $7 \cdot 10^6$ training timesteps on a $d = 3$ board for $(N, N_{new}, k) = (1, 1, 3)$ The line is smoothed by showing the moving average of the timestep reward over a window of 50 timesteps.*

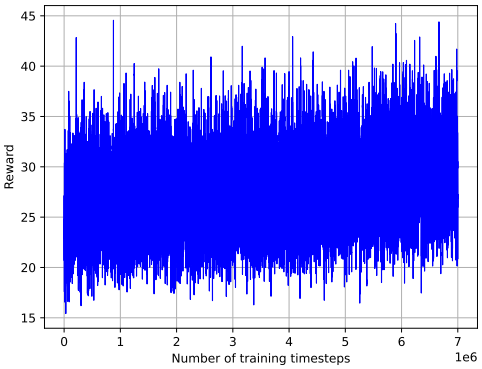


Figure B.5: *Training convergence for a PPO agent over $7 \cdot 10^6$ training timesteps on a $d = 3$ board for $(N, N_{new}, k) = (2, 2, 3)$. The line is smoothed by showing the moving average of the timestep reward over a window of 50 timesteps.*

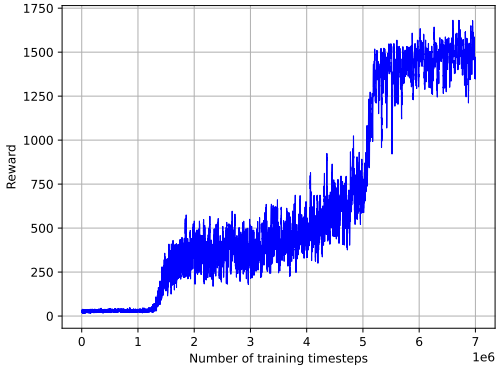


Figure B.6: *Training convergence for a PPO agent over $7 \cdot 10^6$ training timesteps on a $d = 3$ board for $(N, N_{new}, k) = (2, 2, 4)$. The line is smoothed by showing the moving average of the timestep reward over a window of 50 timesteps.*

Appendix C

Code

The code used for this project is written in Python 3 and is publicly available at GitHub [27].

C.1 Description

This repository is constructed to run experiments for the thesis ”Quantum error correction on the toric code using two distinct reinforcement learning game frameworks”, and was originally made as a deliverable for the MSc Thesis of Lindsay Spoor. The research contains the training and evaluation of RL agents on the toric code decoding problem.

Requirements

The code for this project requires Python 3 and the following packages:

- Stable baselines 3
- OpenAI Gymnasium
- Numpy
- Matplotlib
- Pytorch

Environments

This repository contains different environment files, all wrapped in OpenAI Gym, and were originally created for "A NEAT Quantum Error Decoder" (`https://github.com/condensedAI/neat-qec`), and are tailored to the purpose of this research. The following files contain environments:

- `toric_game_static_env.py` - Contains the environment used for the static game framework using a PPO or DQN agent with an MLP structure. The file contains the following environments:
  - ToricGameEnv: introduces uncorrelated bit-flip errors with an error rate
  - ToricGameFixedErrs: introduces uncorrelated bit-flip errors with a fixed amount of errors
  - ToricGameLocalErrs: introduces correlated bit-flip errors with an error rate
  - Board: Implementation of toric code on a board
- `toric_game_static_env_cnn.py` - Contains the environment used for the static game framework using a PPO or DQN agent with a CNN structure. The file contains the following environments:
  - ToricGameEnvCNN: introduces uncorrelated bit-flip errors with an error rate
  - ToricGameFixedErrsCNN: introduces uncorrelated bit-flip errors with a fixed amount of errors
  - Board: Implementation of toric code on a board
- `toric_game_static_env_extra_action.py` - Contains the translation of action space using an agent trained on a dynamic environment to be evaluated on a static environment.
- `toric_game_dynamic_env.py` - Contains the environment used for the dynamic game framework using a PPO agent with an MLP structure. The file contains the following environments:
  - ToricGameDynamicEnv: introduces uncorrelated bit-flip errors with an error rate
  - ToricGameDynamicEnvFixedErrs: introduces uncorrelated bit-flip errors with a fixed amount of errors
  - Board: Implementation of toric code on a board

Agents

The repository contains different files to initialise RL agents with, using Stable Baselines 3 as a library to train and evaluate the agents with. The following files are called to initialise agents with:

- `PPO_static_agent.py` - Contains a class that constructs, trains, loads a PPO agent with an MLP network architecture and is dependent on the environments stored in `toric_game_static_env.py`
- `PPO_CNN_static_agent.py` - Contains a class that constructs, trains, loads a PPO agent with a CNN network architecture and is dependent on the environments stored in `toric_game_static_env_cnn.py`
- `DQN_static_agent.py` - Contains a class that constructs, trains, loads a DQN agent with an MLP network architecture and is dependent on the environments stored in `toric_game_static_env.py`
- `PPO_dynamic_agent.py` - Contains a class that constructs, trains, loads a PPO agent with an MLP network architecture and is dependent on the environments stored in `toric_game_dynamic_env.py`

Training and evaluating the agents

The repository contains different files to train and evaluate RL agents on the toric code. The following files can be executed from the command prompt:

- `run_PPO_static_agent.py` - Specify all settings in this file in order to either train or load and evaluate a PPO model on the static environment, and make sure the correct storing location is specified in `PPO_static_agent.py`.
- `run_PPO_CNN_static_agent.py` - Specify all settings in this file in order to either train or load and evaluate a PPO model with a CNN network architecture on the static environment, and make sure the correct storing location is specified in `PPO_CNN_static_agent.py`.
- `run_DQN_static_agent.py` - Specify all settings in this file in order to either train or load and evaluate a DQN model on the static environment, and make sure the correct storing location is specified in `DQN_static_agent.py`.

- `run_PPO_dynamic_agent.py` - Specify all settings in this file in order to either train or load and evaluate a PPO model on the dynamic environment, and make sure the correct storing location is specified in `PPO_dynamic_agent.py`.
- `evaluate_dynamic_on_static.py` - Specify all settings in this file in order to either train or load a PPO model on the dynamic environment, and evaluate on the static environment, and make sure the correct storing location is specified in `PPO_dynamic_agent.py`.

Version of March 1, 2024– Created March 1, 2024 - 18:58