Approximating 2-Qubit Entanglers in the Fibonacci Model using Reinforcement Learning

Conor Sheridan

April 10, 2025

1 Abstract

Topological quantum computation is inherently fault tolerant, but synthesizing reliable two-qubit entangling gates is challenging due to leakage into noncomputational states. In the Fibonacci anyon model, qubits are encoded in anyonic fusion—single qubits as morphisms $(\tau \otimes \tau) \otimes \tau \to \tau$ and two qubits via their composition, with leakage arising from $\tau \otimes \tau \to 1$. This project employs a recurrent variant of the Proximal Policy Optimization algorithm to generate braid group representations approximating the CNOT gate. Our reinforcement learning framework uses a composite reward function that balances CNOT fidelity against penalties for leakage and nonunitarity, while Makhlin invariants optimize for gates locally equivalent to CNOT. The results point to a promising approach for synthesizing robust quantum gates under physical constraints in topological quantum computing.

Contents

2 Introduction

Topological quantum computation (TQC) offers a novel approach to fault-tolerant quantum information processing by encoding quantum information in the global, topological properties of a physical system. This method leverages anyonic excitations whose braiding naturally implements robust unitary transformations, thereby providing resilience against local sources of error. In this work, we focus on the Fibonacci anyon model, which is especially appealing because it is universal for quantum computation. That is, any desired quantum gate can be approximated to arbitrary accuracy through an appropriate sequence of braiding operations.

In what follows, we provide a detailed exposition of the background and motivation for our study, outline the key challenges in synthesizing entangling gates in the Fibonacci model, and describe our reinforcement learning (RL) approach aimed at discovering effective braiding sequences that approximate the Controlled-NOT (CNOT) gate.

2.1 Background and Motivation

2.1.1 Quantum Computation and Entangling Gates

Quantum computation harnesses the principles of quantum mechanics to perform operations on qubits—quantum bits that can exist in superpositions of states. A universal set of quantum gates typically includes both single-qubit rotations and at least one entangling two-qubit gate, such as the CNOT gate. The CNOT gate, in particular, is pivotal because, together with single-qubit gates, it enables the construction of any quantum algorithm. However, the practical realization of high-fidelity entangling gates remains one of the major challenges in quantum computing.

2.1.2 Fibonacci Anyons and Topological Protection

In TQC, information is stored in non-local degrees of freedom, making it inherently robust to local perturbations. The Fibonacci anyon model is a prominent example due to its simplicity and universality. In this model, particles known as anyons obey non-Abelian statistics, meaning that their braiding results in transformations that depend on the order of operations. A single qubit can be encoded in the fusion space of anyons, for example via the morphism

$$(\tau \otimes \tau) \otimes \tau \to \tau,$$

where the intermediate fusion outcomes form a two-dimensional Hilbert space corresponding to the computational basis. When constructing multi-qubit systems, additional fusion outcomes may introduce noncomputational subspaces—referred to as leakage—which can detract from the fidelity of quantum gate operations.

2.1.3 Braiding Operations and Their Role

Braiding anyons corresponds to performing unitary operations on the encoded qubits. The sequence of braiding operations determines the overall quantum gate implemented by the system. However, the space of possible braid sequences grows exponentially with the length of the sequence, and not every sequence results in a gate that stays within the computational subspace. Therefore, identifying braiding sequences that approximate the target gate (such as the CNOT) with minimal leakage and high unitarity is a nontrivial problem.

2.2 Problem Statement

Designing a reliable entangling gate in the Fibonacci anyon model involves two major challenges. First, the braid sequences must approximate the desired CNOT gate with high fidelity. Second, these sequences must minimize leakage into the noncomputational subspace and maintain the unitarity of the operation. Leakage not only degrades the quality of the quantum operation but also introduces errors that could compromise the fault-tolerance advantages of topological quantum computation.

The core research question addressed in this work is: *How can reinforcement learning be used to discover braiding sequences in the Fibonacci anyon model that approximate the CNOT gate while simultaneously minimizing leakage and preserving unitarity?*

2.3 Objectives of the Study

To address the problem, the objectives of this study are threefold:

1. **Develop a Reinforcement Learning Framework:** We design and implement a recurrent variant of the Proximal Policy Optimization (PPO) algorithm tailored to the sequential decision-making problem of generating braid sequences.

2. **Explore Trade-offs in Gate Synthesis:** By formulating a composite reward function that accounts for gate fidelity, leakage, and unitarity, we investigate the inherent trade-offs involved in approximating the CNOT gate.

3. **Extend Optimization to Locally Equivalent Gates:** Following the approach in [6] we utilize Makhlin invariants[26], to approximate gates that are locally equivalent to the CNOT, thereby potentially increasing the set of acceptable braiding sequences and providing additional insights into the optimization landscape.

2.4 Overview of the Approach and Report Structure

We begin by modeling the Fibonacci anyon system and the associated braiding operations following [10], explaining how qubits are encoded and how leakage arises. This mathematical framework lays the foundation for our RL-based methodology, where the agent's actions correspond to elementary braiding operations and the state represents the current partial sequence. The reward function is carefully designed to balance fidelity against leakage and unitarity constraints.

The remainder of the paper is structured as follows:

- **Section 3** provides a detailed background on the Fibonacci anyon model, the theory of topological quantum gates, and the basics of reinforcement learning, setting the stage for our work.

- **Section 4** formally describes the main metric we use to approximate our gate sequences, Makhlin Invariants.

- **Section 5** describes our methodology in detail, including the design of the RL environment, the architecture of the recurrent PPO agent, and the specifics of the reward function.
- **Section 6** presents the results obtained from our simulations, discussing the performance of the learned braiding sequences and analyzing the trade-offs between gate fidelity, leakage, and unitarity. This section also contains our concludion with a summary of our findings, their implications for topological quantum computing, and potential avenues for future research.

By combining the theoretical underpinnings of TQC with cutting-edge reinforcement learning techniques, this work aims to contribute a robust framework for synthesizing quantum gates in a leakage-prone, non-standard computational setting. The insights gained from this study not only enhance our understanding of the interplay between machine learning and quantum control but also pave the way for more reliable implementations of topological quantum computation.

The code for our model can be found at github.

3 Theoretical Framework

In this section, we detail the methods and tools used in our project. We first describe the underlying theoretical framework based on the Fibonacci anyon model and braid group representations. Next, we explain our reinforcement learning (RL) approach, including the architecture of the recurrent Proximal Policy Optimization (PPO) agent and the design of the reward function. Finally, we outline our optimization and evaluation strategy, including the use of Makhlin invariants and simulation metrics.

3.1 The Fibonacci Anyons

3.1.1 Fibonacci Anyons as a Unitary Modular Tensor Category:

The Fibonacci anyon model is one of the simplest non-Abelian anyon theories and is defined as a modular tensor category (MTC) ([2], pg. 48, Definition 3.1.1). In this category, there are exactly two isomorphism classes of simple objects:

$$\mathbf{1} \text{ (the vacuum)}, \qquad \tau \text{ (the nontrivial anyon)}.$$

The fusion rules are given by

- $\tau \otimes \tau \cong \mathbf{1} \oplus \tau$,
- $\tau \otimes \mathbf{1} \cong \mathbf{1} \otimes \tau \cong \tau$,
- $\mathbf{1} \otimes \mathbf{1} \cong \mathbf{1}$.

As a braided monoidal category ([2], pg.17, Definition 1.2.3), the Fibonacci category is equipped with:

- **Associativity Isomorphisms (F-matrices):** The associator

  $$\alpha_{X,Y,Z} : (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z)$$

  more commonly referred to as the F-matrix when a concrete model is chosen. This operation is part of the underlying data of the monoidal category.

- **Braiding Isomorphisms (R-matrices):** The braidor isomorphism

$$c_{X,Y} : X \otimes Y \to Y \otimes X.$$

more commonly referred to as the R-matrix when a concrete model is chosen. The braiding satisfies the hexagon equations ([11], pg.195, Definition 8.1.1)), ensuring consistency with the associator.[1]

Example 3.1 (**F-matrix**). *Consider the fusion space $V_\tau^{\tau\tau\tau} := Hom(\tau \otimes \tau \otimes \tau, \tau)$ of three τ anyons fusing to an overall outcome τ. We can choose a basis in one of two equivalent ways:*

1. ***Left-Associated Basis:*** *First fuse the first two anyons and then fuse the result with the third. Label the intermediate outcome by a. In this basis, the state is denoted*

$$|((\tau \otimes \tau)_a \otimes \tau)\rangle,$$

where $a \in \{\mathbf{1}, \tau\}$.

2. ***Right-Associated Basis:*** *Alternatively, first fuse the last two anyons and then fuse the first τ with the intermediate outcome c. In this basis, the state is written as*

$$|(\tau \otimes (\tau \otimes \tau)_b)\rangle,$$

where $b \in \{\mathbf{1}, \tau\}$.

The F-matrix is expressed as a linear transformation between the two bases of $V_\tau^{\tau\tau\tau}$. Specifically, we define the F-matrix elements by the relation

$$|(\tau \otimes (\tau \otimes \tau)_b)\rangle = \sum_{a \in \{\mathbf{1}, \tau\}} [F_\tau^{\tau\tau\tau}]_{b,a} \; |((\tau \otimes \tau)_a \otimes \tau)\rangle.$$

Here the F-matrix is typically given by

$$F_\tau^{\tau\tau\tau} = \begin{pmatrix} \phi^{-1} & \sqrt{\phi^{-1}} \\ \sqrt{\phi^{-1}} & -\phi^{-1} \end{pmatrix},$$

where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Once can check that these F-symbols satisfy the pentagon equations required for monoidal coherence.

□

Example 3.2 (**R-matrix**). *Consider the fusion space*

$$V_c^{\tau\tau} := \mathrm{Hom}(\tau \otimes \tau, c)$$

[1] Let $\mathcal{C}$ be a monoidal category. A *braiding* in $\mathcal{C}$ is given by a natural isomorphism $c_{X,Y} : X \otimes Y \to Y \otimes X$ for all objects $X, Y \in \mathcal{C}$, which satisfies the hexagon axioms. In general, the braiding is not required to be symmetric, meaning that the composition $c_{Y,X} \circ c_{X,Y}$ need not equal the identity on $X \otimes Y$, or equivalently $c_{X,Y} \neq c_{Y,X}^{-1}$. In contrast, a *symmetric monoidal category* is a monoidal category equipped with a braiding $\sigma_{X,Y}$ that is involutive; that is, for all objects X, Y, one has $\sigma_{Y,X} \circ \sigma_{X,Y} = \mathrm{id}_{X \otimes Y}$. Thus, while every symmetry is a braiding, not every braiding is symmetric.

for the two-τ fusion channel, where $c \in \{\mathbf{1}, \tau\}$. In this space, the braiding is captured by the isomorphism

$$c_{\tau,\tau} : \tau \otimes \tau \to \tau \otimes \tau,$$

which, when restricted to the fusion channel c, acts by a phase given by the R-matrix element $R_c^{\tau\tau}$. In a chosen basis in which the fusion outcome is labeled by c, we have:

$$c_{\tau,\tau} \left| (\tau \otimes \tau)_c \right\rangle \;=\; R_c^{\tau\tau} \left| (\tau \otimes \tau)_c \right\rangle.$$

In the Fibonacci anyon model the braiding eigenvalues are explicitly given by

$$R_{\mathbf{1}}^{\tau\tau} \;=\; e^{-4\pi i/5}, \qquad R_{\tau}^{\tau\tau} \;=\; e^{3\pi i/5}.$$

Thus, if we denote by $\left| (\tau \otimes \tau)_{\mathbf{1}} \right\rangle$ the state in which two τ anyons fuse to the vacuum, then braiding the two anyons yields

$$c_{\tau,\tau} \left| (\tau \otimes \tau)_{\mathbf{1}} \right\rangle \;=\; e^{-4\pi i/5} \left| (\tau \otimes \tau)_{\mathbf{1}} \right\rangle.$$

Similarly, for the fusion channel where $\tau \otimes \tau$ yields τ, we have

$$c_{\tau,\tau} \left| (\tau \otimes \tau)_{\tau} \right\rangle \;=\; e^{3\pi i/5} \left| (\tau \otimes \tau)_{\tau} \right\rangle.$$

One may check that these R-symbols satisfy the hexagon equations required for the coherence of a braided monoidal category. □

3.1.2 Alternative Description as a 2-Category:

A modern perspective views a UMTC as a fully dualizable object in a higher categorical setting. In particular, one may consider the Fibonacci category as a 2-category with a single object $*$, where:

- The 1-morphisms are given by the objects of the UMTC (i.e., $\mathbf{1}$ and τ).
- The 2-morphisms are the morphisms between these objects generated by the F-matrix and R-matrix.

This 2-categorical view is particularly natural in the context of extended topological quantum field theories (TQFTs), where a UMTC appears as the value of the TQFT on a point. Such an approach is discussed, for example, in [3], which describes modular categories as representations of the 3-dimensional bordism 2-category.

3.1.3 Connection to $\mathbf{SU(2)_3}$ Chern–Simons Theory:

The Fibonacci anyon model has a well-known physical realization in $\mathrm{SU}(2)_k$ Chern–Simons theory for $k = 3$. In $\mathrm{SU}(2)_3$ Chern–Simons theory, the allowed representations (or anyon types) are labeled by spins $j = 0, \frac{1}{2}, 1, \frac{3}{2}$. The Fibonacci theory arises as a subtheory by restricting to the integer spin sector:

$$\text{Objects: } 0 \quad \text{and} \quad 1.$$

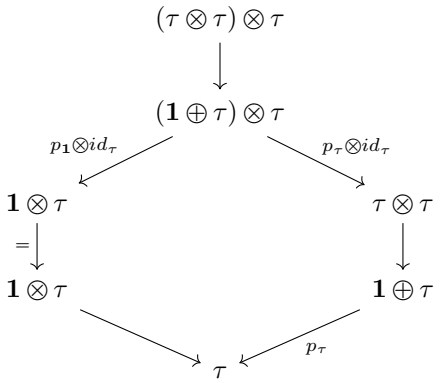
relabeling $\mathbf{1} := 0$ and $\tau := 1$ we recover the Fibonacci model. In the language of quantum groups, the Fibonacci category is equivalent to the representation category of $U_q(\mathfrak{sl}_2)$ with the parameter

$$q = \exp\left(\frac{2\pi i}{5}\right),$$

which is directly related to $SU(2)_3$ Chern–Simons theory. This connection explains the emergence of the golden ratio in the F-symbols and underpins the topological properties used in fault-tolerant quantum computation [12, 26].

3.1.4 Encoding and Fusion Rules

As discussed above, in the Fibonacci anyon model quantum information can be (spasely) encoded in the fusion spaces of the UMTC. A single qubit is represented by the fusion outcome of three anyons, formalized as the diagram:



Here, the intermediate fusion outcome given by the projectors

- $p_{\mathbf{1}} \otimes id_\tau : (\mathbf{1} \oplus \tau) \otimes \tau \to \mathbf{1} \otimes \tau$,
- $p_\tau \otimes id_\tau : (\mathbf{1} \oplus \tau) \otimes \tau \to \tau \otimes \tau$

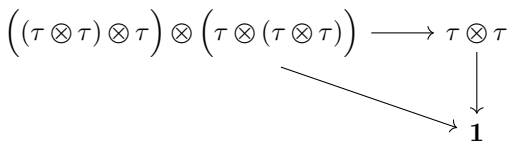
form a basis for a two-dimensional Hilbert space providing an isomorphism

$$Hom((\tau \otimes \tau) \otimes \tau, \tau) \simeq \mathbb{C}^2.$$

When constructing two-qubit systems, we compose two copies of the above encoding and postcompose with the fusion $\tau \otimes \tau \to 1$. Since the representable functors preserve the monoidal product, that is, they are lax monoidal functors, we have:

$$\begin{aligned} Hom(\Big((\tau \otimes \tau) \otimes \tau\Big) \otimes \Big(\tau \otimes (\tau \otimes \tau)\Big), \tau \otimes \tau) \\ \simeq Hom((\tau \otimes \tau) \otimes \tau, \tau) \otimes Hom(\tau \otimes (\tau \otimes \tau), \tau) \\ \simeq \mathbb{C}^2 \otimes \mathbb{C}^2 \end{aligned}$$

Thus the collection of morphisms that factor through $\tau \otimes \tau$ as in the diagram



form a basis for a four-dimensional Hilbert space. We denote the basis elements of this Hilbert space as

- $|00\rangle$ corresponding to $\cdots \circ \Big(p_{\mathbf{1}} \otimes \cdots\Big) \otimes \Big(p_{\mathbf{1}} \otimes \cdots\Big)$,
- $|01\rangle$ corresponding to $\cdots \circ \Big(p_{\mathbf{1}} \otimes \cdots\Big) \otimes \Big(p_{\tau} \otimes \cdots\Big)$,
- $|10\rangle$ corresponding to $\cdots \circ \Big(p_{\tau} \otimes \cdots\Big) \otimes \Big(p_{\mathbf{1}} \otimes \cdots\Big)$,
- $|11\rangle$ corresponding to $\cdots \circ \Big(p_{\tau} \otimes \cdots\Big) \otimes \Big(p_{\tau} \otimes \cdots\Big)$.

Unfortunately there is a morphism that does not factor through $\tau \otimes \tau$, namely

$$\begin{array}{c} \Big((\tau \otimes \tau) \otimes \tau\Big) \otimes \Big(\tau \otimes (\tau \otimes \tau)\Big) \\ \downarrow \\ \Big(\tau \otimes \tau\Big) \otimes \Big(\tau \otimes \tau\Big) \\ \downarrow \\ \mathbf{1} \otimes \mathbf{1} \\ \downarrow \\ \mathbf{1}. \end{array}$$

This introduces a redundent dimension into our encoding and means that we have an isomorphism

$$Hom(\Big((\tau \otimes \tau) \otimes \tau\Big) \otimes \Big(\tau \otimes (\tau \otimes \tau)\Big), \mathbf{1}) \simeq \mathbb{C}^5.$$

We denote the basis for this redundency as $|NC\rangle$ and refer to it as the *noncomputational basis* [10, 6].

3.1.5 Braid Group Representations

Braiding anyons is the core physical process that implements quantum gates in topological quantum computing. As mentioned above, the braiding isomorphism (or *braidor*) of a braided monoidal category induces a unitary representation of the three-strand braid group on the fusion spaces of the UMTC [10]

$$\rho_3 : B_3 \to \mathbf{U}(V_\tau^{\tau\tau\tau}).$$

Here, the n-strand braid group is defined as

$$B_n = \Big\langle \sigma_1, \sigma_2, \ldots, \sigma_{n-1} \;\Big|\; \begin{array}{ll} \sigma_i\sigma_j = \sigma_j\sigma_i & \text{if } |i-j| \geqslant 2, \\ \sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1} & \text{for } 1 \leqslant i \leqslant n-2 \end{array} \Big\rangle.$$

Intuitively, the generators σ_i correspond to elementary braiding operations—specifically, the counterclockwise exchange of the i-th and $(i+1)$-th anyons. These operations satisfy the Yang–Baxter equation,

$$\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1},$$

which guarantees the consistency of braiding three or more anyons. The representation ρ_3 assigns to each braid a unitary operator acting on the fusion space $V_\tau^{\tau\tau\tau}$, with the unitary transformations determined by the R- and F-matrices, representing the phase shifts acquired during the exchange

of anyons, and encoding the change of basis between different fusion outcomes, respectively. In this way, the Yang–Baxter equation ensures that the non-Abelian statistics encoded in these matrices are coherently maintained, thereby implementing quantum gates in a topologically protected manner.

For our 2-qubit encoding we follow [10] and use the following braid group generators

$$\begin{aligned}
\rho_6(\sigma_1) &= (R_\tau^{\tau\tau}) \oplus (R \otimes I_2) \\
\rho_6(\sigma_2) &= (R_\tau^{\tau\tau}) \oplus (FRF \otimes I_2) \\
\rho_6(\sigma_3) &= P_{14}((R_\tau^{\tau\tau}) \oplus R \oplus FRF)P_{14} \\
\rho_6(\sigma_4) &= (R_\tau^{\tau\tau}) \oplus (I_2 \otimes FRF) \\
\rho_6(\sigma_5) &= (R_\tau^{\tau\tau}) \oplus (I_2 \otimes R).
\end{aligned}$$

Note the fact that these generators have been checked by the authors of [6].

3.2 Reinforcement Learning

Reinforcement Learning [25] is a paradigm in machine learning focused on sequential decision-making, where an agent learns to interact with an environment by taking actions and receiving feedback in the form of rewards. The primary goal of RL is to develop a policy—a mapping from states to actions—that maximizes the expected cumulative reward over time. Unlike supervised learning, where the correct output is provided for each input, RL relies on trial-and-error exploration and a reward signal to guide learning.

In our work, RL provides a framework for navigating the complex space of braid sequences in topological quantum computing. By framing the synthesis of quantum gates as a sequential decision process, our RL agent iteratively refines its policy to minimize errors such as leakage and non-unitarity while approximating target operations like the CNOT gate. This is achieved through advanced techniques such as Proximal Policy Optimization (PPO), which ensures that policy updates remain stable and that learning progresses incrementally. The following sections detail the core concepts of reinforcement learning, describe our agent architecture, and explain how the learning algorithms are integrated into our compositional framework.

In the following we break this exposition into two sections

- **classical perspective:** Where we recall the usual perspective of reinforcement learning in terms of Markov decision processes. This section is material that can be found in [25],
- **original perspective** Where we attempt to model the modern cybernetic perspective in an elementary language.

3.2.1 Classical Perspective

Reinforcement learning is commonly formulated in the language of Markov decision processes (MDPs). An MDP is defined as a tuple

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \gamma),$$

where:

- **State Space, $\mathcal{S}$:** This is a (typically measurable) set of states that the system can occupy.
- **Action Space, $\mathcal{A}$:** This is a set of actions available to the agent. Depending on the application, $\mathcal{A}$ may be finite, countably infinite, or even continuous.

- **Transition Kernel, P:** For $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$, the function

$$P(s' \mid s, a) = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}$$

defines the probability of transitioning to state s' from state s when action a is taken. The Markov property asserts that this transition probability depends only on the current state and action, and not on any earlier history.

- **Reward Function, r:** The function

$$r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$$

assigns an instantaneous reward $r(s, a, s')$ to the transition from state s to state s' when action a is executed.

- **Discount Factor, γ:** A constant $\gamma \in [0, 1]$ that determines the present value of future rewards. When $\gamma = 1$, one typically considers finite-horizon problems or average-reward criteria to ensure well-definedness.

Policies and Objective An agent in an MDP aims to choose actions in a way that maximizes some cumulative measure of reward. A policy π is a mapping that specifies a probability measure over actions given a state:

$$\pi(a \mid s) = \Pr\{a_t = a \mid s_t = s\}.$$

A policy is said to be deterministic if it assigns probability one to a single action for each state, and stochastic otherwise.

The quality of a policy is measured via its value functions. The *state-value function* under policy π is defined as:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t, s_{t+1}) \,\middle|\, s_0 = s \right],$$

where the expectation $\mathbb{E}_\pi$ is taken over trajectories generated by following the policy π and the stochastic transitions determined by P.

Similarly, the *action-value function* (or Q-function) is given by:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t, s_{t+1}) \,\middle|\, s_0 = s,\, a_0 = a \right].$$

The Optimality Criterion The ultimate objective in RL is to find an optimal policy π^* that maximizes the expected discounted cumulative reward for all states:

$$V^{\pi^*}(s) = \sup_\pi V^\pi(s), \quad \forall s \in \mathcal{S}.$$

Consequently, the *optimal state-value function* $V^*(s)$ satisfies:

$$V^*(s) = \sup_\pi \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t, s_{t+1}) \,\middle|\, s_0 = s \right],$$

and, similarly, the *optimal action-value function* is:

$$Q^*(s, a) = \sup_\pi Q^\pi(s, a).$$

Dynamic Programming and the Bellman Equations A cornerstone of the RL framework is the dynamic programming principle, which manifests in the Bellman equations.

For any policy π, the state-value function satisfies the *Bellman expectation equation*:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[r(s, a, s') + \gamma V^{\pi}(s') \right].$$

Similarly, for the action-value function:

$$Q^{\pi}(s, a) = \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[r(s, a, s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a' \mid s') Q^{\pi}(s', a') \right].$$

Bellman Optimality Equations For the optimal value functions, the Bellman equations take the form:

$$V^{*}(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[r(s, a, s') + \gamma V^{*}(s') \right],$$

and

$$Q^{*}(s, a) = \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \left[r(s, a, s') + \gamma \max_{a' \in \mathcal{A}} Q^{*}(s', a') \right].$$

These equations are central because they provide a recursive characterization of the optimal value functions.

3.2.2 Classical Actor-Critic Model

The Actor-Critic framework [5] is a class of algorithms that unifies policy-based and value-based reinforcement learning methods.

Suppose the policy is parameterized by a vector θ; that is, the actor is represented by the function $\pi_{\theta} : \mathcal{S} \times \mathcal{A} \to [0, 1]$ such that

$$\pi_{\theta}(a \mid s) = \Pr\{a_t = a \mid s_t = s; \theta\}.$$

The critic, on the other hand, is designed to provide an estimate of a value function. For instance, one may introduce a parameterized value function $V_{\omega}(s)$, with parameters ω, to serve as a baseline, or alternatively, a parameterized action-value function $Q_{\omega}(s, a)$.

The central idea is to use the critic's estimate to guide the policy updates of the actor. Specifically, if one uses the state-value function, an advantage function is defined as:

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s),$$

or estimated directly, which corrects for the variability in the returns and helps reduce the variance of the policy gradient.

Policy Gradient and Actor Update The objective of the actor is to maximize the expected cumulative discounted reward, defined as:

$$J(\theta) = \mathbb{E}_{\pi_{\theta}} \left[\sum_{t=0}^{\infty} \gamma^{t} r(s_t, a_t, s_{t+1}) \right].$$

By the policy gradient theorem, the gradient of $J(\theta)$ can be written as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a \mid s) \, A^\pi(s, a) \right].$$

In practice, the actor's parameters are updated along the gradient ascent direction:

$$\theta \leftarrow \theta + \alpha_\theta \, \nabla_\theta \log \pi_\theta(a \mid s) \, \hat{A}(s, a),$$

where $\hat{A}(s, a)$ is an estimator of the true advantage $A^\pi(s, a)$ and $\alpha_\theta > 0$ is the learning rate.

Critic Update The critic's role is to provide a reliable estimate of either $V^\pi(s)$ or $Q^\pi(s, a)$. A common approach is to minimize a temporal-difference (TD) error that compares the current estimate with a bootstrapped target. For example, when approximating $V^\pi(s)$, one might update the critic by minimizing the mean-squared TD error:

$$L(\omega) = \mathbb{E}_{\pi_\theta} \left[\big(r(s, a, s') + \gamma V_\omega(s') - V_\omega(s) \big)^2 \right],$$

and updating ω in the direction of the gradient descent:

$$\omega \leftarrow \omega - \alpha_\omega \, \nabla_\omega L(\omega),$$

High-Level Summary of the Actor-Critic Loop In simpler language, in the actor-critic framework, two key components—the actor and the critic—operate in tandem to improve the agent's performance:

- **Actor:** The actor is responsible for selecting actions according to a parameterized policy $\pi_\theta(a \mid s)$. It directly interacts with the environment by choosing actions based on the current state, aiming to maximize the expected cumulative reward.
- **Critic:** The critic evaluates the actions taken by the actor by estimating a value function (either the state-value $V_\omega(s)$ or action-value $Q_\omega(s, a)$). It provides an estimate of the expected future rewards, which is used to compute an advantage signal (i.e., the difference between the observed return and the current estimate).

The agent-environment loops is as follows:

1. The actor selects an action based on the current state using $\pi_\theta(a \mid s)$.
2. The environment responds with a new state and a reward.
3. The critic evaluates the new state and computes (for example) a temporal-difference (TD) error, which reflects the discrepancy between the predicted value and the observed outcome.
4. This TD error is used to form an advantage estimate, guiding the actor's update by scaling the gradient of the log-probability of the selected action.
5. Simultaneously, the critic updates its parameters to better predict the values corresponding to the current policy.

This iterative process, where the critic's rapid evaluation informs the actor's slower policy adjustments, embodies the actor-critic loop. The continuous feedback loop facilitates more stable and efficient learning by combining direct policy optimization with value-based evaluation.

3.2.3 Original Perspective

Below is an attempt at an original perspective inspired by contemporary work in systems theory [8][17][21], and cybernetics [15] [13] [7]. Any overlap with the literature is due to my own ignorance.

Cybernetics In a broad sense, a *cybernetic system* (or open game) is characterized by three fundamental components:

- **Input:** A source of signals or data that the system consumes.
- **Parameter Object:** Internal degrees of freedom or parameters that govern how the system processes inputs.
- **Output:** The signals or data produced by the system in response to its inputs and internal parameters.

These components are arranged in a feedback loop, allowing the system's outputs to influence future inputs (and potentially update the parameters).

Formally, one can model such a system as a profunctor optic[8]—a structure that captures both how information flows "forward" from inputs to outputs and how it flows "backward" from outputs (and external objectives) to update the internal parameters. In category theory, profunctor optics provide a compositional framework: smaller open games (cybernetic systems) can be combined into larger ones, preserving their input-output interfaces and internal parameterizations.

Reinforcement Learning as a Cybernetic System A reinforcement learning setup naturally fits into this cybernetic picture. From this viewpoint:

- **Input:** The RL agent receives an observation (or state) from the environment.
- **Parameter Object:** The agent has internal parameters—often represented by weights in a function approximator (e.g., a neural network policy)—that dictate how it maps states to actions.
- **Output:** The agent selects an action based on its current policy parameters and the observed state.

Meanwhile, the *environment* can itself be treated as a complementary "agent" in the open game formalism, taking the RL agent's action as input and producing the next state and reward as output.

When the environment satisfies the *Markov property*—i.e., the probability of transitioning to the next state depends only on the current state and action—the entire RL loop can be regarded as two interacting open games:

- **The Learning Agent:** A cybernetic system whose parameters are iteratively adjusted (e.g., via gradient updates) based on reward signals.
- **The Environment:** Another cybernetic system that updates states with Markovian dynamics and generates rewards.

In this arrangement, the composite (agent + environment) is an open game exchanging actions and states: each of the two constituent open games has its own inputs, parameters, and outputs, and they are wired together to form a feedback loop. The profunctor optic viewpoint ensures that any local changes (e.g., a gradient update in the agent's parameters) can be propagated coherently throughout the larger system.

3.2.4 Statics

We assume the following types:

- $\mathcal{S}$: the type of observations (or states) of the environment;
- $\mathcal{A}$: the type of actions of the agent. Actions are environment transition functions, more on this in the next section;
- $\mathcal{H}$: the type of hidden states (or short-term memory) of the agent;
- Θ: the type of learnable parameters (e.g., the weights of a neural network). Intuitively these are to be thought of as a compressed (lossy) representation of the agents entire history.

We bundle the hidden state and the parameter set into a single type:

$$\mathcal{M} := \mathcal{H} \times \Theta.$$

Additionally, we assume we have two monads ([1], pg. 265, Chapter 10):

- the probability monad [4][2] Δ, and
- the state monad [20] $T(-) := \mathcal{M} \to (- \times \mathcal{M})$.

Intuitively a computation $T(X)$ is a function

$$\begin{aligned} \mathcal{H} \times \Theta &\to (X \times (\mathcal{H} \times \Theta))) \\ (h, \theta) &\mapsto (x, h', \theta') \end{aligned}$$

that, given a "state of short-term memory" $h \in \mathcal{H}$ and a "state of long-term memory" $\theta \in \Theta$, produces a value $x \in X$ and an updated "state of short-term memory" $h' \in \mathcal{H}$ and a "state of long-term memory" $\theta' \in \Theta$.

We define an **agent** as a morphism

$$f : \mathcal{S} \to T\left(\Delta(\mathcal{A})\right),$$

that is, given an observation $s \in \mathcal{S}$ and an internal memory state $m = (h, \theta) \in \mathcal{M}$ the agent produces a pair:

$$f(s, m) = (\pi(s, m), m'),$$

where $\pi(s, m) \in \Delta(\mathcal{A})$ is a probability distribution over actions and $m' \in \mathcal{M}$ is the updated internal state.

We can decompose f into three composable components, each represented as a (pure) function:

1. The **policy** function, which extracts the action distribution from the current observation and internal memory state:

$$\begin{aligned} \pi : \mathcal{S} \times \mathcal{M} &\to \Delta(\mathcal{A}) \\ (s, m) &\mapsto \pi(s, m) \end{aligned}$$

[2]There are many monads that go by the name probability monad, here we mean a probability monad with finite support.

2. The **memory update** function, which updates the short-term internal memory state based on the current observation:
$$t : \mathcal{S} \times \mathcal{M} \to \mathcal{M}$$
$$(s, (h, \theta)) \mapsto (h', \theta)$$

3. The **learner** function, which refines the parameter set in response to a reward signal.

   Let
$$r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$$
$$(s, a) \mapsto x$$
   denote some **reward function** that assigns a scores based on taking action $a \in \mathcal{A}$ in environment state $s \in \mathcal{S}$. Let
$$U : \mathbb{R} \times \mathcal{M} \to \mathcal{M}$$
$$(x, (h, \theta)) \mapsto (h, \theta')$$
   be some **parameter update** function. Then the learner is given by the composite
$$\text{learn} : \mathcal{S} \times \mathcal{A} \times \mathcal{M} \xrightarrow{r \times id_{\mathcal{M}}} \mathbb{R} \times \mathcal{M} \xrightarrow{\quad U \quad} \mathcal{M}.$$
$$\begin{aligned} \text{learn}(s, a, (h, \theta)) &= U(r(s, a), (h, \theta)) \\ &= (h, \theta'). \end{aligned}$$

Example 3.3 (Value function)**.** *Assume the agent has a value function*
$$V : \mathcal{S} \times \mathcal{H} \times \Theta \to \mathbb{R},$$
which estimates the expected cumulative reward starting from an observation $s \in \mathcal{S}$, with working memory $h \in \mathcal{H}$ and parameters $\theta \in \Theta$. We define a loss function L_V for the value prediction as follows:
$$L_V : \Theta \times \mathcal{S} \times \mathcal{A} \times \mathbb{R} \to \mathbb{R}, \quad L_V(\theta; s, a, r(s, a)) \;=\; \frac{1}{2}\Big(V(s, h, \theta) - r(s, a)\Big)^2.$$
Here, $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function.

The parameter update function is then given by
$$U_V\big(r(s, a), (h, \theta)\big) \;=\; \Big(h,\, \theta - \eta\, \nabla_\theta L_V(\theta; s, a, r(s, a))\Big),$$
where $\eta > 0$ is the learning rate. □

Example 3.4 (Action-Value function)**.** *Suppose we define an action-value function (Q-function)*
$$Q : \mathcal{S} \times \mathcal{A} \times \mathcal{H} \times \Theta \to \mathbb{R},$$
which estimates the expected return of taking action a in state s under the current parameters θ. In a Q-learning setting, one typically defines a loss function L_Q as
$$L_Q : \Theta \times \mathcal{S} \times \mathcal{A} \times \mathbb{R} \to \mathbb{R}, \quad L_Q(\theta; s, a, y) \;=\; \frac{1}{2}\Big(Q(s, a, \theta) - y\Big)^2,$$
where the target y is computed, for example, as
$$y = r(s, a) + \gamma \max_{a' \in \mathcal{A}} Q(s', a', \theta),$$

with s' being the next state and $\gamma \in [0,1)$ the discount factor.

The corresponding parameter update function is

$$U_Q\big(r(s,a),(h,\theta)\big) \;=\; \Big(h,\; \theta - \eta\,\nabla_\theta L_Q(\theta; s, a, y)\Big).$$

□

In a "do"-notation style, a single iteration of the agent's operation is written as follows:

$f(s,m) =$ **do**
$a \leftarrow \pi(s,(h,\theta))$; (sample an action from the policy)
$h' \leftarrow t(s,(h,\theta))$; (update the internal state, e.g., working memory)
$\theta' \leftarrow U\big(r(s,a),(h,\theta)\big)$; (update the parameters based on the reward)
return $(a,(h',\theta'))$.

Example 3.5. *More explicitly, assume at time t the agent's state is $m_t = (h_t, \theta_t)$ and it observes s_t. The steps are as follows:*

1. Action Selection: *Compute the policy output:*

$$\pi(s_t, m_t) = \pi(s_t, (h_t, \theta_t)) \in \Delta(\mathcal{A}),$$

and sample an action a_t.

2. Memory Update: *Update the working memory:*

$$m_t^{(1)} = t(s_t, m_t) = (h_{t+1}, \theta_t).$$

3. Parameter Update: *After executing a_t, the environment returns a reward $r(s_t, a_t)$. The learning function then updates the parameters:*

$$\theta_{t+1} = \theta_t - \eta\,\nabla_\theta L(\theta_t; s_t, a_t, r(s_t, a_t)),$$

and we set

$$m_{t+1} = (h_{t+1}, \theta_{t+1}).$$

Thus, the entire operation of the agent is given by:

$$f(s_t, m_t) = \big(a_t,\; m_{t+1}\big).$$

□

3.2.5 Dynamics

Now, we formalize the interface between the environment and the agent. We begin by defining the environment transition function and then show how to combine it with the agent's behavior.

1. **The Environment Transition Function:** We assume that the environment is governed by a stochastic transition function

$$\text{step} : \mathcal{S} \to \Delta(\mathcal{S}),$$

which, given an environment state $s \in \mathcal{S}$, produces a finitely supported probability distribution over the next states in $\mathcal{S}$. In our framework, step is a Kleisli arrow for the probability monad Δ. This means that when composing multiple stochastic transition functions, we use the Kleisli composition. For instance, if $f, g : \mathcal{S} \to \Delta(\mathcal{S})$ are two such transition functions, their composition is defined as

$$\mathcal{S} \xrightarrow{f} \Delta(\mathcal{S}) \xrightarrow{\Delta(g)} \Delta(\Delta(\mathcal{S})) \xrightarrow{\mu} \Delta(\mathcal{S})$$

$$g \circ f = \mu \circ \Delta(g) \circ f,$$

This compositional structure ensures that the probabilistic effects are correctly propagated through successive transitions.

2. **Environment–Agent Transition Function:** Consider an arbitrary map $\phi : \mathcal{M} \to \Delta(\mathcal{M})$. We define an **environment-agent transition** function as a map

$$step_\phi = \bullet \circ (step \times \phi) : \mathcal{S} \times \mathcal{M} \to \Delta(\mathcal{S}) \times \Delta(M) \to \Delta(\mathcal{S} \times \mathcal{M}).$$

As a special case we have the lift

$$\mathcal{S} \times \mathcal{M} \xrightarrow{step \times g_{\mathcal{M}}} \Delta(S) \times \Delta(M) \xrightarrow{\bullet} \Delta(\mathcal{S} \times \mathcal{M})$$

$$\begin{aligned} &\text{step}_{\mathcal{M}} : \mathcal{S} \times \mathcal{M} \to \Delta(\mathcal{S} \times \mathcal{M}) \\ &\text{step}_{\mathcal{M}} := \bullet \circ (step \times g_{\mathcal{M}}) \end{aligned}$$

which applies the environment transition to the environment component and leaves the internal state unchanged.

3. **The Agent and Its Adjoint Equivalence:** Recall, our agent is modeled by a function

$$f : \mathcal{S} \to T\big(\Delta(\mathcal{A})\big),$$

where $T(-)$ is a state monad (with $T(X) = \mathcal{M} \to (X \times \mathcal{M})$) and $\Delta(\mathcal{A})$ denotes the probability distribution over actions. If we make the reasonable assumption that we are working in a cartesian closed category, then there is an adjoint equivalence

$$\text{Hom}\big(\mathcal{S}, T(\Delta(\mathcal{A}))\big) \simeq \text{Hom}\big(\mathcal{S} \times \mathcal{M}, \Delta(\mathcal{A}) \times \mathcal{M}\big),$$

where the transformation $f \mapsto \ulcorner f \urcorner$ "lifts" f to explicitly take the agent's internal state into account.

4. **Environment–Agent Interaction:** We define an **environment–agent interaction** as an environment-agent transition function $\mathcal{S} \times \mathcal{M} \to \Delta(\mathcal{S} \times \mathcal{M})$ that factors through an action $\mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$.

   More precisely, we assume that the action space $\mathcal{A}$ is a subobject of the type of state-transition functions $[\mathcal{S}, \Delta(\mathcal{S})]$; that is, each action $a \in \mathcal{A}$ is a function $a : \mathcal{S} \to \Delta(\mathcal{S})$. Using this fact, the (stochastic) **action-evaluation** map $\text{ev}_{\text{act}} : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ applies an action to a state.

   In complete detail the overall environment–agent interaction can be captured by the following composite function

$$
\begin{array}{c}
\mathcal{S} \times \mathcal{M} \\
\downarrow{\scriptstyle id_{\mathcal{S}} \times \ulcorner f \urcorner} \\
\mathcal{S} \times \Delta(\mathcal{A} \times \mathcal{M}) \\
\downarrow{\scriptstyle g_{\mathcal{S}} \times id_{\Delta(\mathcal{A} \times \mathcal{M})}} \\
\Delta(\mathcal{S}) \times \Delta(\mathcal{A} \times \mathcal{M}) \\
\downarrow{\scriptstyle \bullet} \\
\Delta(\mathcal{S} \times (\mathcal{A} \times \mathcal{M})) \\
\downarrow{\scriptstyle \alpha_{\mathcal{S}\mathcal{A}\mathcal{M}}} \\
\Delta((\mathcal{S} \times \mathcal{A}) \times \mathcal{M}) \\
\downarrow{\scriptstyle \Delta(ev_{\mathcal{A}} \times id_{\mathcal{M}})} \\
\Delta(\Delta(\mathcal{S}) \times \mathcal{M}) \\
\downarrow{\scriptstyle marg} \\
\Delta(\Delta(\mathcal{S})) \times \Delta(\mathcal{M}) \\
\downarrow{\scriptstyle \mu \times id_{\mathcal{M}}} \\
\Delta(\mathcal{S}) \times \Delta(\mathcal{M}) \\
\downarrow{\scriptstyle \bullet} \\
\Delta(\mathcal{S} \times \mathcal{M})
\end{array}
$$

5. Although the above composite is quite complicated it essentiall boils down to:
   (a) start with an environment-agent state $(s, (h, \theta))$
   (b) run a cycle of the agent to get the action-brain pair $(a, (h', \theta'))$
   (c) apply the action to change the state $ev_{\mathcal{A}}(s, a) = s'$ giving us a new environment-agent pair $(s', (h', \theta'))$

This formalism allows us to view the entire (Recurrent) RL interaction cycle as a compositional, monadic process. The agent's behaviour is combined with the environment's stochastic dynamics and the overall system produces a new distribution over environment states and internal agent states.

3.2.6 Original: Actor-Critic Model

In the actor–critic models [18], we partition the parameter space into two parts: one for the actor (the policy) and one for the critic (the value function). For clarity, we can write

$$\Theta = \Theta_{\text{actor}} \times \Theta_{\text{critic}},$$

so that the overall internal state becomes

$$\mathcal{M} = \mathcal{H} \times (\Theta_{\text{actor}} \times \Theta_{\text{critic}}).$$

- **The Actor:** The actor is responsible for decision making. In our framework, it is given by the modified policy function

$$\pi_{actor} : \mathcal{S} \times (\mathcal{H} \times \Theta_{actor}) \to \Delta(\mathcal{A}),$$

- **The Critic:** The critic estimates the expected (discounted) return. It is modeled by a map

$$V_{critic} : \mathcal{S} \times (\mathcal{H} \times \Theta_{critic}) \to \mathbb{R},$$

where $V(s, m)$ provides a real-valued evaluation of state s given the current internal state m.

We now define an actor-critic model as an agent of the form

$$f : \mathcal{S} \to T\big(\Delta(\mathcal{A} \times \mathbb{R})\big).$$

Equivalently, the curried form is

$$f : \mathcal{S} \times \mathcal{M} \to \Delta(\mathcal{A} \times \mathbb{R}) \times \mathcal{M},$$

and it is defined by

$$f(s, m) \;=\; \Big(\big((\pi(s, m), V(s, m))\big),\, m'\Big).$$

Here, the pair $(\pi(s, m), V(s, m))$ comprises the actor's stochastic policy and the critic's evaluation, and m' is the updated internal state (resulting from a separate memory update function, as discussed below).

In an actor–critic model, both the actor and critic parameters are updated using information about the reward. Let

$$r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$$

denote the immediate reward function. The learning update is performed via a function

$$U : \mathbb{R} \times \mathcal{M} \to \mathcal{M},$$

which typically updates the parameter components Θ_{actor} and Θ_{critic} based on a loss function that incorporates both the actor loss (often via a clipped surrogate objective as in PPO) and the critic loss (typically a mean-squared error between the critic's value and a temporal-difference target). In other words, for a given reward signal $r(s, a)$ and internal state $m = (h, \theta_{\text{actor}}, \theta_{\text{critic}})$, we define

$$U\big(r(s, a), m\big) \;=\; \big(h,\, \theta'_{\text{actor}},\, \theta'_{\text{critic}}\big),$$

where the updates θ'_{actor} and θ'_{critic} are computed typically by gradient descent on their respective loss functions.

Example 3.6.

- $\mathcal{S} = \mathbb{R}^n$ *is the space of observations (or environment states).*
- $\mathcal{A} = \{0, 1\}$ *is a binary action space.*
- *We assume a trivial hidden state (or working memory), so that* $\mathcal{H} = \{*\}$ *(a singleton), and hence*

$$\mathcal{M} := \mathcal{H} \times \Theta,$$

where $\Theta = \Theta_{actor} \times \Theta_{critic}$.

- *Let* $\Theta_{actor} = \mathbb{R}^{1\times n}$ *and* $\Theta_{critic} = \mathbb{R}^{1\times n}$.

The agent's internal state is represented by

$$m = (*, (\theta_{actor}, \theta_{critic})) \in \mathcal{M}.$$

1. The Actor (Policy) Function *The actor produces a probability distribution over actions given an observation $s \in \mathcal{S}$ and the current parameters. Define*

$$\pi : \mathcal{S} \times \mathcal{M} \to \Delta(\mathcal{A})$$

by

$$\pi(s, (*, (\theta_{actor}, \theta_{critic}))) = \mathrm{softmax}(\theta_{actor} \cdot s),$$

where the softmax is taken over the two outcomes 0 and 1. Concretely, if $\theta_{actor} \cdot s = x \in \mathbb{R}$, we define

$$\pi(s, m)(0) = \frac{e^{-x}}{e^{-x} + e^{x}}, \quad \pi(s, m)(1) = \frac{e^{x}}{e^{-x} + e^{x}}.$$

2. The Critic (Value) Function *The critic estimates the expected return (value) of a state. Define*

$$V : \mathcal{S} \times \mathcal{M} \to \mathbb{R}$$

by

$$V(s, (*, (\theta_{actor}, \theta_{critic}))) = \theta_{critic} \cdot s.$$

Thus, given s and the current critic parameters, $V(s, m)$ is a linear function that approximates the expected return.

3. The Learner (Parameter Updater) Function *For the critic, we define the temporal-difference target*

$$y = r(s, a) + \gamma\, V(s', (*, (\theta_{actor}, \theta_{critic}))),$$

where s' is the next state, and $\gamma \in [0, 1)$ is the discount factor. Then the critic loss is

$$L_{critic}(\theta_{critic}; s, a, r) = \frac{1}{2}\Big(V(s, (*, (\theta_{actor}, \theta_{critic}))) - y\Big)^2,$$

and we update

$$\theta'_{critic} = \theta_{critic} - \eta_{critic}\, \nabla_{\theta_{critic}} L_{critic}(\theta_{critic}; s, a, r).$$

For the actor, we compute an ***advantage estimate***

$$A(s, a) = y - V(s, (*, (\theta_{actor}, \theta_{critic}))),$$

and define the actor loss via the policy gradient. A typical update is

$$\theta'_{actor} = \theta_{actor} + \eta_{actor}\, \nabla_{\theta_{actor}} \log \pi(s, m)(a)\, A(s, a).$$

Thus, the learner function updates the internal state as

$$learn(s, a, m) = \Big(*,\ \big(\theta_{actor} - \eta_{actor}\, \nabla_{\theta_{actor}} L_{actor}(s, a),\ \theta_{critic} - \eta_{critic}\, \nabla_{\theta_{critic}} L_{critic}(s, a, r)\big)\Big).$$

Using the above components, we define the overall actor–critic function as:

$$f : \mathcal{S} \times \mathcal{M} \to \Delta(\mathcal{A} \times \mathbb{R}) \times \mathcal{M},$$

where

$$f(s, m) \ = \ \Big(\big((\pi(s, m), V(s, m))\big),\ learn(s, a, m_1)\Big).$$

Here, m_1 is the intermediate internal state after applying any memory update $t(s, m)$ (if necessary), and a is sampled from the distribution $\pi(s, m)$.

After sampling an action a and obtaining a new state s' from the environment, the entire cycle updates the internal state m to m' and returns the pair $(a, V(s, m))$ in the probabilistic context. □

In summary, while our formulation of the actor–critic model captures the core conceptual structure—partitioning parameters into actor and critic, generating stochastic policies via π, estimating values via V, and updating parameters through gradient-based learning—it abstracts away several practical considerations that are vital in real-world implementations. In practice, additional techniques such as sophisticated advantage estimation (e.g., generalized advantage estimation [23]), variance reduction methods[5], and strategies for efficient exploration (including experience replay[22] or trust-region methods) play a crucial role in stabilizing and accelerating learning. These techniques can be integrated into our framework by refining the loss functions, adjusting the learning updates, or extending the monadic structure to account for more complex side effects. Ultimately, while our abstract model provides a rigorous and compositional basis, incorporating these practical refinements is essential for achieving robust performance in real-world reinforcement learning applications.

3.2.7 Recurrent Proximal Policy Optimization

PPO as a Modified Actor–Critic Model. Recall that an actor–critic model consists of two primary functions:

$$\pi : \mathcal{S} \times \mathcal{M} \to \Delta(\mathcal{A}), \quad \text{(the actor or policy function)},$$
$$V : \mathcal{S} \times \mathcal{M} \to \mathbb{R}, \quad \text{(the critic or value function)},$$

Given an observation $s \in \mathcal{S}$ and an internal state $m \in \mathcal{M}$, the actor produces a stochastic policy $\pi(s, m) \in \Delta(\mathcal{A})$, while the critic provides a value estimate $V(s, m) \in \mathbb{R}$.

In Proximal Policy Optimization (PPO) [24], the policy is updated by maximizing a surrogate objective designed to limit the deviation between the new policy and the old policy. Formally, if we denote the previous (old) policy by $\pi_{\theta_{\text{old}}}$, the PPO surrogate objective is given by

$$L^{\text{PPO}}(\theta) = \mathbb{E}_{(s,a)\sim\pi_{\theta_{\text{old}}}}\left[\min\left(\frac{\pi_\theta(a \mid s, m)}{\pi_{\theta_{\text{old}}}(a \mid s, m)}\,\hat{A}(s, a),\ \text{clip}\left(\frac{\pi_\theta(a \mid s, m)}{\pi_{\theta_{\text{old}}}(a \mid s, m)},\ 1-\epsilon,\ 1+\epsilon\right)\,\hat{A}(s, a)\right)\right],$$

where:

1. $\hat{A}(s, a)$ is an advantage estimate computed from the critic, and
2. $\epsilon > 0$ is a hyperparameter that controls the allowed change in the policy.

This surrogate objective $L^{\text{PPO}}(\theta)$ is maximized with respect to the actor parameters. In our framework, we capture the gradient-based update for the actor using a learner function. In particular, we define a learner (or parameter updater) function

$$U : \mathbb{R} \times \mathcal{M} \to \mathcal{M},$$

which applies a gradient step based on a reward signal or, more generally, gradient information derived from the surrogate objective.

More precisely, suppose that for a given sample (s, a), the loss for the actor is

$$L^{\text{PPO}}(\theta; s, a) \quad \text{(to be maximized)},$$

and the critic's loss is

$$L_{\text{critic}}(\theta; s, a, r) = \frac{1}{2}\Big(V(s, m) - y\Big)^2,$$

where the target y is typically defined as

$$y = r(s,a) + \gamma\, V(s', m)$$

with s' being the next state and $\gamma \in [0,1)$ the discount factor.

Then the learner function is given by

$$\begin{aligned} U\big(r(s,a),(h,(\theta_{\text{actor}},\theta_{\text{critic}}))\big) \;=\; \Big(h,\; & \\ & \Big(\theta_{\text{actor}} + \eta_{\text{actor}}\, \nabla_{\theta_{\text{actor}}} L^{\text{PPO}}(\theta; s, a), \\ & \quad \theta_{\text{critic}} - \eta_{\text{critic}}\, \nabla_{\theta_{\text{critic}}} L_{\text{critic}}(\theta; s, a, r(s,a))\Big)\Big), \end{aligned}$$

where:

(i) $L^{\text{PPO}}(\theta; s, a)$ is the surrogate objective for the actor (whose gradient is used for a gradient ascent update),

(ii) $L_{\text{critic}}(\theta; s, a, r)$ is the critic's loss (minimized via gradient descent),

(iii) η_{actor} and η_{critic} are the learning rates for the actor and critic, respectively.

Below is a revised version of the subsection that faithfully reflects the LSTM architecture and allows for the memory update function to couple with the parameter update function. This version uses a categorical and type-theoretic perspective without referring to measure-theoretic issues, and it makes explicit the roles of the hidden state and cell state in the LSTM recurrence.

Recurrent Memory Update via LSTM. In partially observable environments, a single observation

$$s \in \mathcal{S}$$

often fails to capture all relevant information about the underlying environment. To overcome this limitation, an agent is typically endowed with a recurrent mechanism—most notably, a Long Short-Term Memory (LSTM) network [16] which maintains a dynamic representation of history.

In our framework, we extend the internal state to include both a hidden state and a cell state, and we combine these with the learnable parameters. Concretely, we set

$$\mathcal{M} = \mathcal{H} \times \mathcal{C} \times \Theta,$$

where:

- $\mathcal{H}$ is the space of hidden states,
- $\mathcal{C}$ is the space of cell states, and
- Θ is the space of parameters.

The recurrent memory update function is then defined as

$$t : \mathcal{S} \times (\mathcal{H} \times \mathcal{C} \times \Theta) \to (\mathcal{H} \times \mathcal{C} \times \Theta).$$

For an input $(s, (h, c, \theta))$, the function t is instantiated as an LSTM cell update. That is, using the standard LSTM recurrence, we compute:

$$\begin{aligned} i &= \sigma\big(W_i\, s + U_i\, h + b_i\big), \\ f &= \sigma\big(W_f\, s + U_f\, h + b_f\big), \\ o &= \sigma\big(W_o\, s + U_o\, h + b_o\big), \\ \tilde{c} &= \tanh\big(W_c\, s + U_c\, h + b_c\big), \end{aligned}$$

where σ denotes the logistic sigmoid function, and W_*, U_*, and b_* are matrices and biases that form part of the parameter set θ. Then, the cell and hidden states are updated as follows:

$$\begin{aligned} c' &= f \odot c + i \odot \tilde{c}, \\ h' &= o \odot \tanh(c'). \end{aligned}$$

Thus, we define

$$t(s, (h, c, \theta)) = \big(h', c', \theta'\big),$$

where θ' may be equal to θ or be further updated by coupling with a learner function U. In practice, one often updates the parameters θ (which include the LSTM weights) simultaneously with the recurrent update, so that the dynamics of the memory and the long-term policy are interdependent.

This recurrent update t is then integrated into the overall agent function. If the overall agent is given by

$$f : \mathcal{S} \times \mathcal{M} \to \Delta(\mathcal{A}) \times \mathcal{M},$$

we can decompose f as follows:

$$f(s, m) = \Big(\pi(s, m),\, U\big(r(s, a), t(s, m)\big)\Big),$$

where:

(i) $\pi : \mathcal{S} \times \mathcal{M} \to \Delta(\mathcal{A})$ is the policy (actor) function that selects actions stochastically,

(ii) $r(s, a)$ is the reward function,

(iii) $t(s, m)$ updates the internal state using the LSTM recurrence, and

(iv) $U : \mathbb{R} \times \mathcal{M} \to \mathcal{M}$ is the learner (parameter updater) function that incorporates the reward signal into a further update of θ.

3.3 Agent Architecture

Our agent is built upon a recurrent Proximal Policy Optimization (PPO) framework, utilizing an LSTM specifically tailored for the synthesis of braid sequences in topological quantum computing.

State and Action Spaces. The agent's *state space* $\mathcal{S}$ is defined as the set of all ordered sequences of braid gates, where the available braid gates are those introduced in Section 3.1.5. The *action space* $\mathcal{A}$ consists of the individual braid gates that can be postcomposed to extend the current sequence. At each decision point, the agent selects an action $a \in \mathcal{A}$, thereby appending the corresponding braid gate to the existing sequence. In our framework, the agent's observation at time t is an element $s_t \in \mathcal{S}$, representing the current braid sequence.

Reward Function. The agent's performance is evaluated using a reward function that is defined as a weighted sum of three error measures:

1. **Unitary Error:** Measures the deviation of the synthesized gate from unitarity.
2. **Leakage:** Penalizes the interaction with noncomputational subspaces.
3. **Closeness Error:** Quantifies the similarity between the synthesized gate and the target CNOT gate. For this term, we use either the Schatten p-norm to measure direct numerical closeness or Makhlin invariants to assess local equivalence.

The overall reward is computed as a weighted combination of these errors, with the objective of minimizing error (or equivalently, maximizing reward).

The agent is tasked with generating a random sequence from 20 to 40 that minimizes this error, and therefore constructing an approximation to a 2-qubit entangling gate for the Fibonacci model.

4 Local Equivalence Classes

In [10], it was conjectured that directly approximating entangling two-qubit gates (such as CNOT) via Fibonacci braids might be computationally infeasible without additional insight. This claim was further investigated in [6], where numerical evidence suggested the difficulty of such direct approximations. As a result, [6] introduced the idea of approximating not the CNOT *itself*, but rather any gate in the *local equivalence class* of CNOT. In other words, we allow for single-qubit transformations to precede or follow the main entangling operation, thereby broadening the range of acceptable target gates while still retaining the same nonlocal content as the CNOT. This approach is the primary target for our agent.

In this section, we present two methods for classifying two-qubit gates up to their local equivalence classes: the *Cartan decomposition* of SU(4) and the *Makhlin invariants*. Note that our model only uses the latter as a target, but for completeness we include the Cartan decomposition.

4.1 Motivation

When working with two qubits, any unitary gate $U \in \mathrm{SU}(4)$ can be factored into local operations (one-qubit unitaries) and a nonlocal component that genuinely entangles the qubits. Symbolically:

$$U \;=\; (V_1 \otimes W_1)\, U_{\mathrm{nl}}\, (V_2 \otimes W_2),$$

where V_i and W_i act on individual qubits, and U_{nl} is the nonlocal part. Two gates U and U' are said to be *locally equivalent* if they share the same nonlocal part up to local unitaries:

$$U' \;=\; (V_1' \otimes W_1')\, U_{\mathrm{nl}}\, (V_2' \otimes W_2').$$

Local gates do not alter the nature or amount of entanglement that a two-qubit gate can produce. Hence, when looking for an entangling gate, two gates in the same local equivalence class are essentially the same.

4.2 Cartan Decomposition of SU(4)

Let $G = \mathrm{SU}(4)$ denote the group of 4×4 special unitary matrices. Our aim is to derive a canonical factorization of any $U \in G$ into local and nonlocal components via the Cartan (or KAK) decomposition. We now present a derivation.

4.2.1 Lie Algebra Decomposition

Let $\mathfrak{g} = \mathfrak{su}(4)$ denote the Lie algebra of G; that is, the set of 4×4 complex matrices X satisfying

$$X^\dagger = -X \quad \text{and} \quad \operatorname{tr}(X) = 0.$$

Equip $\mathfrak{g}$ with the Hilbert–Schmidt inner product defined by

$$\langle X, Y \rangle = \operatorname{tr}(X^\dagger Y).$$

In the context of two-qubit systems, a natural subalgebra is given by the *local* operations, namely those that act independently on each qubit. Define

$$\mathfrak{k} \cong \mathfrak{su}(2) \oplus \mathfrak{su}(2)$$

as the set of block-diagonal matrices in $\mathfrak{su}(4)$ of the form

$$X = X_1 \otimes I_2 + I_2 \otimes X_2, \quad X_1, X_2 \in \mathfrak{su}(2).$$

Then, we define the orthogonal complement of $\mathfrak{k}$ by

$$\mathfrak{p} = \{Y \in \mathfrak{su}(4) \mid \langle Y, X \rangle = 0 \text{ for all } X \in \mathfrak{k}\}.$$

Thus, we obtain the vector space decomposition

$$\mathfrak{su}(4) = \mathfrak{k} \oplus \mathfrak{p}.$$

This is a *symmetric decomposition* in the sense that the Lie brackets satisfy

$$[\mathfrak{k}, \mathfrak{k}] \subset \mathfrak{k}, \quad [\mathfrak{k}, \mathfrak{p}] \subset \mathfrak{p}, \quad [\mathfrak{p}, \mathfrak{p}] \subset \mathfrak{k}.$$

4.2.2 Maximal Abelian Subalgebra of $\mathfrak{p}$

Within $\mathfrak{p}$, we select a maximal Abelian subalgebra $\mathfrak{a}$, in the two-qubit setting we choose

$$\mathfrak{a} = \operatorname{span}\Big\{ i\,\sigma_x \otimes \sigma_x,\ i\,\sigma_y \otimes \sigma_y,\ i\,\sigma_z \otimes \sigma_z \Big\},$$

where $\sigma_x, \sigma_y, \sigma_z$ are the Pauli matrices. It is straightforward to verify that these elements are skew-Hermitian and that

$$[i\,\sigma_j \otimes \sigma_j,\, i\,\sigma_k \otimes \sigma_k] = 0 \quad \text{for all } j, k \in \{x, y, z\},$$

so that $\mathfrak{a}$ is Abelian.

An arbitrary element $A \in \mathfrak{a}$ can be uniquely written as

$$A = -\frac{i}{2}\Big(c_1\,\sigma_x \otimes \sigma_x + c_2\,\sigma_y \otimes \sigma_y + c_3\,\sigma_z \otimes \sigma_z \Big),$$

with real coefficients c_1, c_2, c_3. The choice of the prefactor $-\frac{i}{2}$ is conventional and ensures that the exponential map

$$\exp : \mathfrak{su}(4) \to \mathrm{SU}(4)$$

yields unitaries in G without additional scaling.

The Cartan (KAK) Decomposition

For a compact Lie groups every $U \in G$ can be expressed as

$$U = K_1 \, \exp(A) \, K_2,$$

where K_1 and K_2 belong to the Lie subgroup $K \subset G$ corresponding to $\mathfrak{k}$ (namely, $K \cong \mathrm{SU}(2) \times \mathrm{SU}(2)$), and $A \in \mathfrak{a}$.

Since elements of K are of the form $k_1 \otimes k_2$ with $k_1, k_2 \in \mathrm{SU}(2)$, the decomposition can be written explicitly as

$$U = (k_1 \otimes k_2) \, \exp\Big[-\frac{i}{2}\Big(c_1 \, \sigma_x \otimes \sigma_x + c_2 \, \sigma_y \otimes \sigma_y + c_3 \, \sigma_z \otimes \sigma_z\Big)\Big] \, (k_3 \otimes k_4),$$

with $k_1, k_2, k_3, k_4 \in \mathrm{SU}(2)$. In this expression:

- The factors $k_1 \otimes k_2$ and $k_3 \otimes k_4$ are *local* unitaries acting on the individual qubits.
- The central term,

  $$U_{\mathrm{nl}} = \exp\Big[-\frac{i}{2}\Big(c_1 \, \sigma_x \otimes \sigma_x + c_2 \, \sigma_y \otimes \sigma_y + c_3 \, \sigma_z \otimes \sigma_z\Big)\Big],$$

  constitutes the *nonlocal* or entangling part of U.

Uniqueness via the Weyl Chamber

The coefficients (c_1, c_2, c_3) are determined only up to the action of the Weyl group of the Lie algebra $\mathfrak{a}$. To obtain a unique representation, one restricts (c_1, c_2, c_3) to a fundamental domain called the *Weyl chamber*. A standard choice is

$$\mathcal{W} = \Big\{(c_1, c_2, c_3) \in \mathbb{R}^3 \;\Big|\; 0 \leqslant c_3 \leqslant c_2 \leqslant c_1 \leqslant \frac{\pi}{2}\Big\}.$$

Once the parameters are chosen in $\mathcal{W}$, the decomposition

$$U = (k_1 \otimes k_2) \, U_{\mathrm{nl}} \, (k_3 \otimes k_4)$$

is unique up to the ambiguities of the local factors.

4.2.3 Determining Local Equivalence to CNOT

Given a two-qubit unitary $U \in \mathrm{SU}(4)$ with Cartan decomposition

$$U = (k_1 \otimes k_2) \, \exp\Big[-\frac{i}{2}\Big(c_1 \, \sigma_x \otimes \sigma_x + c_2 \, \sigma_y \otimes \sigma_y + c_3 \, \sigma_z \otimes \sigma_z\Big)\Big] \, (k_3 \otimes k_4),$$

where $k_i \in \mathrm{SU}(2)$ are local unitaries and $(c_1, c_2, c_3) \in \mathcal{W}$ (the Weyl chamber), the nonlocal component

$$U_{\mathrm{nl}} = \exp\Big[-\frac{i}{2}\Big(c_1 \, \sigma_x \otimes \sigma_x + c_2 \, \sigma_y \otimes \sigma_y + c_3 \, \sigma_z \otimes \sigma_z\Big)\Big]$$

fully determines the entangling power of U.

For the CNOT gate, the canonical parameters are (under our conventions)

$$(c_1, c_2, c_3) = \Big(\frac{\pi}{2}, 0, 0\Big).$$

To assess whether U is approximately locally equivalent to CNOT, we compute the distance between the extracted parameters and the canonical CNOT parameters:

$$d\Big((c_1, c_2, c_3), \Big(\frac{\pi}{2}, 0, 0\Big)\Big) = \sqrt{\Big(c_1 - \frac{\pi}{2}\Big)^2 + c_2^2 + c_3^2}.$$

If

$$d\Big((c_1, c_2, c_3), \Big(\frac{\pi}{2}, 0, 0\Big)\Big) < \epsilon,$$

for a prescribed tolerance $\epsilon > 0$, then the nonlocal component of U is approximately equal to that of CNOT. Since local unitaries do not affect the entangling properties, this criterion guarantees that U is approximately locally equivalent to the CNOT gate.

4.3 Makhlin Invariants

Let $U \in \mathrm{SU}(4)$ be a two-qubit unitary operator. Makhlin [19] demonstrated that the local equivalence class of U can be completely characterized by a small number of scalar invariants. We now describe a derivation of these invariants.

4.3.1 The Magic Basis and Conjugation

The construction begins with the introduction of a fixed unitary matrix $Q \in \mathrm{U}(4)$ which transforms U into a particular basis commonly known as the *magic basis*. Although the choice of Q is not unique, any two valid choices differ only by local unitaries. Define

$$U_B = Q^\dagger \, U \, Q.$$

In the magic basis, the structure of local operations is simplified; in particular, the action of local unitaries becomes either block-diagonal or real orthogonal.

4.3.2 Definition of m_U

Once U is expressed in the magic basis, we define the matrix

$$m_U = U_B^T \, U_B.$$

The construction of m_U is motivated by the following observation: if one replaces U by a locally equivalent gate, the corresponding U_B undergoes a transformation by block-diagonal unitaries, and the product $U_B^T U_B$ is conjugated by a real orthogonal matrix. Consequently, the eigenvalues of m_U, and hence functions of its trace, remain invariant under local conjugations.

4.3.3 Extraction of the Invariants

From the matrix m_U one defines three scalar quantities that serve as complete invariants under local operations. The choice we use is [6]

$$g_1 = \mathrm{Re}\left\{ \frac{\mathrm{tr}^2(m_U)}{16 \det(U)} \right\} \quad g_2 = \mathrm{Im}\left\{ \frac{\mathrm{tr}^2(m_U)}{16 \det(U)} \right\} \quad g_3 = \frac{\mathrm{tr}^2(m_U) - \mathrm{tr}\big(m_U^2\big)}{4 \det(U)}$$

4.3.4 Invariance under Local Operations

Suppose U' is obtained from U by local operations:

$$U' = (U_X \otimes U_Y)\, U\, (V_X \otimes V_Y).$$

Under the corresponding change in the magic basis, one obtains

$$U'_B = Q^\dagger\, U'\, Q = (Q^\dagger (U_X \otimes U_Y) Q)\, U_B\, (Q^\dagger (V_X \otimes V_Y) Q).$$

The matrices $Q^\dagger(U_X \otimes U_Y)Q$ and $Q^\dagger(V_X \otimes V_Y)Q$ are real orthogonal, so that $m_{U'} = {U'_B}^T\, U'_B$ is similar to m_U via a real orthogonal conjugation. Since the trace and determinant are invariant under similarity transformations, it follows that

$$\mathrm{tr}(m_{U'}) = \mathrm{tr}(m_U) \quad \text{and} \quad \mathrm{tr}(m_{U'}^2) = \mathrm{tr}(m_U^2).$$

Therefore, the invariants g_1, g_2, and g_3 are unchanged under local operations, and the local equivalence class of U is completely determined by the triple (g_1, g_2, g_3). Similar to the Cartan decomposition we can use these three numbers to check closesness to the CNOT gate.

4.4 Implications for Gate Synthesis in TQC

From a topological quantum computing (TQC) perspective, targeting the local equivalence class of CNOT—rather than a particular instance of CNOT itself—offers three major benefits:

1. **Approximation:** Even if the actual gate is difficult to approximate, we might still find gates that are locally equivalent to it. This is especially important for entangling gates (such as CNOT) which are invariant under local equivalence transforms.

2. **Reduced Search Space:** By allowing for single-qubit rotations before or after the nonlocal entangling braid, we can accommodate a broader family of acceptable solutions.

3. **Resource Accounting:** In many quantum algorithms, local operations are cheaper or easier to implement fault-tolerantly, while the two-qubit entangling step is more resource-intensive. By explicitly separating out the nonlocal portion, we clarify the primary resource cost of the entangling gate.

5 Implementation

After a thorough exposition of the theoretical foundations underlying our work, we now describe the practical implementation and experimental results. In our approach, we address the problem of approximating a target quantum gate through the sequential application of braid generators. To this end, we design a custom environment that simulates the evolution of a quantum gate composition and deploy a recurrent reinforcement learning agent to explore and optimize the sequence of braid operations.

In what follows, we first detail the **Environment-Agent Interaction**. This description covers the formulation of the state space—where the evolving gate composition is represented as a 5×5 complex matrix and encoded as a 50-dimensional real vector—the discrete action space that corresponds to the application of specific braid generators (or their inverses), and the state transitions that occur upon each action. We then explain the construction of our multi-term reward function,

which is designed to penalize errors arising from leakage, non-unitarity, and deviation from the target gate.

The subsequent sections provide a two-tiered presentation. The *High-Level Overview* section presents a concise summary of the overall program flow, while the *In-Depth Details* section offers a comprehensive description of the environment design, training setup, algorithmic interactions, and evaluation procedures. Together, these sections illustrate how our implementation leverages recurrent reinforcement learning and parallelized training to efficiently drive the quantum gate approximation process.

5.0.1 The Environment-Agent Interaction

State Space. The state space starts as a 5×5 identity matrix I. At every instance after that the environment is a composition of braid generators. We let $M \in \mathbb{C}^{5\times 5}$ be the matrix representing the composite braid operator at any stage in the episode. Ideally, the physical process should act on a four-dimensional subspace (corresponding to the intended two-qubit gate); however, as mentioned before, braiding can introduce leakage into an ancillary dimension. To capture this phenomenon, we allow M to be a full 5×5 complex operator.

To be compatible with the API interface used, it is necessary to represent the state as a finite-dimensional real vector. We achieve this by flattening both the real and imaginary components of M, yielding a 50-dimensional observation:

$$\underbrace{\big(\Re(M_{1,1}), \Re(M_{1,2}), \ldots, \Re(M_{5,5})\big)}_{\text{25 real components}}, \quad \underbrace{\big(\Im(M_{1,1}), \Im(M_{1,2}), \ldots, \Im(M_{5,5})\big)}_{\text{25 imaginary components}}.$$

Action Space. In each decision step, the agent selects from a finite set of ten discrete actions. These actions correspond to the application of one of five topological braid generators $\rho_1, \rho_2, \rho_3, \rho_4, \rho_5$ or their respective inverses, i.e.,

$$\{\rho_1,\ \rho_2,\ \rho_3,\ \rho_4,\ \rho_5,\ \rho_1^{-1},\ \rho_2^{-1},\ \rho_3^{-1},\ \rho_4^{-1},\ \rho_5^{-1}\}.$$

Mathematically, each generator ρ_j (or inverse ρ_j^{-1}) is represented by a 5×5 matrix derived via a combination of tensor products, direct sums, and swaps on the R-matrix and F-matrix. For convenience we recall the generators here

$$\begin{aligned}
\rho_6(\sigma_1) &= (R_\tau^{\tau\tau}) \oplus (R \otimes I_2) \\
\rho_6(\sigma_2) &= (R_\tau^{\tau\tau}) \oplus (FRF \otimes I_2) \\
\rho_6(\sigma_3) &= P_{14}((R_\tau^{\tau\tau}) \oplus R \oplus FRF)P_{14} \\
\rho_6(\sigma_4) &= (R_\tau^{\tau\tau}) \oplus (I_2 \otimes FRF) \\
\rho_6(\sigma_5) &= (R_\tau^{\tau\tau}) \oplus (I_2 \otimes R).
\end{aligned}$$

During agent initialization, these symbolic expressions undergo numeric substitution (e.g., $\rho_j \leftarrow \rho_j\big|_{\text{subs}}$) and are ultimately cast to NumPy arrays for efficient matrix multiplications.

Transitions. Given the current composite gate M_t at time t, selecting an action

$$a \in \{\rho_1, \ldots, \rho_5, \rho_1^{-1}, \ldots, \rho_5^{-1}\}$$

updates the system to

$$M_{t+1} \;=\; A_a \circ M_t,$$

where A_a is the numeric 5×5 array corresponding to the chosen action. After the prespecified chosen number of steps, the environment ends the episode. This stochastic variation encourages the agent to find approximations to the target gate independent of the composition length.

This design ensures that each episode is finite-horizon, while still providing enough opportunity for the agent to explore various compositions and refine its strategy over time.

5.0.2 Reward Function

To guide the agent toward constructing braid compositions that emulate a target quantum gate $T \in \mathbb{C}^{4\times4}$, we introduce a reward mechanism based on three key criteria: leakage, unitarity, and closeness to the target operation. This multi-term reward function explicitly balances distinct error sources that influence overall performance.

Metrics. Denoting the agent's current composite operator by $M \in \mathbb{C}^{5\times5}$. We use the error metrics as used in [6]:

1. **Leakage Error:** We inspect the element $M[0,0]$ to gauge how much amplitude leaks into or out of the primary 4×4 sub-block. The leakage magnitude is

$$\text{leakage} \;=\; \big|M[0,0]\big|.$$

   Because a perfect embedding of the two-qubit gate would entail $M[0,0] = 1$, we define leakage_error $= 1 -$ leakage. A lower $\big|M[0,0]\big|$ implies a higher penalty, as it suggests the system fails to remain confined to the intended subspace.

2. **Unitarity Error:** We extract the four-dimensional sub-block $U = M[1:5, 1:5] \in \mathbb{C}^{4\times4}$. In a fully ideal situation, U should be unitary. Let

$$U^\dagger U - I_4$$

   denote the deviation from unitarity. We measure this deviation with the Schatten 1-norm

$$\big\|U^\dagger U - I_4\big\|_1 = \operatorname{tr}|U^\dagger U - I_4|.$$

   A perfectly unitary submatrix yields zero for this metric; as unitarity degrades, the norm increases.

3. **Closeness to Target:** Here we have a switch `local_equivalence_class : Bool`, whose value corresponds to either directly approximating the target CNOT gate or approximating the CNOT gate up to local gate transformations respectively. We compare the sub-block U to the desired target gate T. Two variants are available:

   - *Schatten 2-norm:* If `local_equivalence_class` is disabled, we compute

$$\left\| \frac{U}{\|U\|_2} \;-\; \frac{T}{\|T\|_2} \right\|_2,$$

     ensuring that overall magnitude and global phase factors are normalized away.
   - *Makhlin invariants*: If `local_equivalence_class` is enabled, we compare the Makhlin invariants of U and T. This method is robust to local unitary transformations and effectively identifies whether U and T belong to the same equivalence class up to single-qubit operations.

   A lower closeness error corresponds to a more faithful realization of T.

Weighted Combination. Let `leakage_error`, `closeness_error`, `unitarity_error` be the aforementioned terms. Then the reward R incurred is given by a linear combination of these errors:

$$R \;=\; -\Big(\alpha \text{ leakage\_error} \;+\; \beta \text{ closeness\_error} \;+\; \gamma \text{ unitarity\_error}\Big),$$

where α, β, γ are nonnegative weighting coefficients such that $\alpha + \beta + \gamma = 1$. In the current implementation, $\alpha = 0.7$, $\beta = 0.1$, and $\gamma = 0.2$. The negative sign is due to the fact that we are penalizing the agent with respect to the errors but dually we could have provide positive feedback instead.

Incremental Rewards and Final Bonus. To encourage continual improvement, after each gate composition the agent obtains a small incremental reward (see reward shaping [14]) reflecting any reduction in the combined error from one step to the next. More formally, if E_t denotes the sum of weighted errors at time t, then upon transitioning to state $t+1$, the agent gains an additional scaled reward $\Delta = E_t - E_{t+1}$. This incremental reward structure drives the agent to refine its composition incrementally.

Once an episode terminates, an unscaled terminal bonus is added based on the agent's final composition metrics. If the final operator M yields a sub-block U that is both nearly unitary and close to T, the bonus provides a substantial positive signal. Conversely, if M is far from the target, the bonus remains negative (or minimal), reflecting poor performance.

Overall, this reward design punishes all primary errors simultaneously, while still providing the agent with a stepwise incentive to make small but consistent gains. By balancing continuous feedback (through incremental rewards) with a final evaluation (through the terminal bonus), the agent is more likely to find short, high-fidelity braid sequences that converge to the desired target gate in a stable and sample-efficient manner.

5.1 Implementation Details

5.1.1 Reinforcement Learning Algorithm

In order to manage the complexities inherent in braiding-based gate composition, we employ a recurrent variant of the Proximal Policy Optimization (PPO) algorithm, namely *RecurrentPPO*, as provided by the `sb3_contrib` repository [9]. The core motivation behind opting for a recurrent approach is the sequential nature of the problem: an agent may need to recall past actions or partial operator states to effectively refine the sequence toward the targeted quantum gate. By incorporating Long Short-Term Memory (LSTM) layers the policy network gains a form of "memory," potentially allowing it to capture long-range dependencies and identify patterns that emerge only after a series of braiding actions.

Hyperparameter Configuration. We initialize the policy with an MLP-LSTM architecture specified via:

$$\text{net\_arch} = [512, 256, 128, 64],$$

indicating a feedforward layer of 512 units, followed by smaller intermediate layers of 256, 128, and 64 neurons, respectively. A recurrent LSTM layer [16] sits alongside these layers to handle hidden states across time steps. We use a learning rate of 1×10^{-4} and set the discount factor γ to 0.9 to focus on relatively shorter-horizon rewards while still retaining some long-term credit assignment. We tested training for several timesteps (for instance $100,000$ timesteps for composition lengths between 20 and 40) to accumulate sufficient trajectories for convergence in this high-dimensional environment.

Why Recurrent? Unlike standard feedforward policies, which observe the environment state in isolation, a recurrent architecture naturally integrates past observations with the current one. This is particularly relevant for gate composition, where the operator evolves cumulatively, and subtle patterns—such as how a specific braid generator might invert or reinforce a previously applied gate—can emerge only after multiple sequential actions. Employing LSTM memory cells thus aids in capturing these dependencies, potentially leading to more efficient exploration and less tendency to rediscover trivial or redundant sequences.

5.1.2 Parallelization Strategy

To expedite training, we harness the `SubprocVecEnv` approach provided by Stable Baselines3. This interface spawns multiple parallel environment instances, allowing the agent to collect experience data simultaneously from a variety of initial conditions and random seeds. In this work, eight concurrent processes are instantiated (num_envs = 8), striking a balance between computational load and improved sample diversity. By parallelizing experience collection, the effective throughput of training episodes dramatically increases, often yielding faster convergence and improved stability.

5.2 High-Level Overview

The overall program can be divided into several main stages. In essence, our framework involves:

1. **Environment Initialization:** We create a custom OpenAI Gym environment (`GateApproxEnv`) that processes symbolic braid gates and a target gate. This environment defines the action space (each braid gate is an action) and produces a 50-dimensional observation (a flattened 5×5 complex matrix). It also initializes the gate composition (starting as the identity matrix).

2. **Training with Recurrent PPO:** To accommodate the sequential dependencies inherent in the braiding process, we use a recurrent variant of Proximal Policy Optimization (PPO). Our policy network, which includes LSTM layers alongside feedforward layers, leverages memory to recall past actions. This recurrent approach helps the agent refine the gate composition over time.

3. **Parallel Data Collection:** To speed up training, we use vectorized environments (via `SubprocVecEnv`) that run multiple instances of the environment in parallel. This increases sample diversity and throughput.

4. **Recording and Logging Metrics:** A custom callback class (`InfoCallback`) extracts environment-specific metrics (e.g., leakage, closeness error, unitarity error, and episode rewards) after each episode. These metrics are recorded for later analysis.

5. **Policy Evaluation and Visualization:** Once training is complete, the trained policy is evaluated in the vectorized environment via the `evaluate_policy_vec` function. Additionally, helper functions produce plots of training metrics and cumulative performance scores, providing insights into agent performance over time.

The flow of the program is summarized in the following pseudocode.

Algorithm 1 High-Level Flow of the Program (Formal Version)

procedure MainFlow

Let $\mathcal{G} \leftarrow$ `GateApproxEnv`($\mathcal{B}$, T, Σ, $L_{\max}$, α, β, γ, ϵ)

Define the action space $\mathcal{A}$ and the observation space $\mathcal{O} \subset \mathbb{R}^{50}$.

Set the initial gate composition:

$$G_0 \leftarrow I_{5\times 5}$$

where $I_{5\times 5}$ is the 5×5 identity matrix.

Construct a vectorized environment $\mathcal{E}_v$ via `SubprocVecEnv` with $n = 8$ parallel instances.

Initialize the recurrent PPO policy π_θ, using an MLP-LSTM architecture.

Instantiate the logging callback $C \leftarrow$ `InfoCallback`().

while stopping criterion not satisfied **do**

Observe current state $s \in \mathcal{O}$.

Choose action $a \sim \pi_\theta(s)$.

Update gate composition:

$$G \leftarrow G \cdot B_a,$$

where $B_a \in \mathcal{B}$ is the braid gate corresponding to action a.

Compute reward:

$$r \leftarrow -\Big(\alpha\, \mathrm{err}_{\mathrm{leak}} + \beta\, \mathrm{err}_{\mathrm{close}} + \gamma\, \mathrm{err}_{\mathrm{unit}}\Big).$$

Log the metrics (e.g., $\mathrm{err}_{\mathrm{leak}}$, $\mathrm{err}_{\mathrm{close}}$, $\mathrm{err}_{\mathrm{unit}}$, r) via C.

end while

Evaluate the trained policy on $\mathcal{E}_v$ using `evaluate_policy_vec`:

$$(\{G_f\}, \{\delta\}) \leftarrow \texttt{evaluate\_policy\_vec}(\mathcal{E}_v, \pi_\theta, N)$$

where $\{G_f\}$ denotes the final gate compositions and $\{\delta\}$ the associated error metrics for N episodes.

Plot the following:

- The sequence of final rewards r versus episode number.
- Leakage ℓ, closeness error c, and unitarity error u per episode.
- Cumulative scores that reflect the frequency of episodes attaining low error.

end procedure

where:

- $\mathcal{B}$ is the set of symbolic braid gates,
- T is the target gate,
- Σ is the substitution dictionary,
- $L_{\max}$ is the maximum sequence length, and
- α, β, γ, ϵ are hyperparameters.

5.3 In-Depth Description

The core of our implementation is the `GateApproxEnv` class, a custom OpenAI Gym environment designed for our goal of composing a series of braid gates to approximate the local equivalence class of a target quantum gate.

5.3.1 Initialization and Environment Setup

Upon instantiation, the environment processes symbolic braid gates and a target gate by substituting numerical values and evaluating them. If any symbol remains unresolved, an error is raised. The class then defines an action space—assigning one action per braid gate—and an observation space consisting of a flattened (real and imaginary) 50-dimensional vector derived from a 5×5 complex matrix. Finally, the initial gate composition is set to the identity matrix.

Algorithm 2 Environment Initialization

procedure InitializeEnv(*braid_gates*, *target_gate*, *subs*, *max_length*, α, β, γ, *local_equivalence_class*)
  for each gate g in *braid_gates* **do**
    $g_{num} \leftarrow g$.subs(*subs*).evalf()
    if g_{num} contains unresolved symbols **then**
      Error: "Not all symbols substituted in a braid gate."
    end if
    Append numeric array of g_{num} to `braid_gates`
  end for
  $t_{num} \leftarrow$ *target_gate*.subs(*subs*).evalf()
  if t_{num} contains unresolved symbols **then**
    Error: "Not all symbols substituted in target gate."
  end if
  `target_gate` ← numeric array of t_{num}
  `action_space` ← Discrete(number of braid gates)
  `observation_space` ← Box(low=$-\infty$, high=∞, shape=$(50,)$)
  ResetComposition
end procedure

5.3.2 Resetting the Composition and Episode Preparation

Before each episode, the environment resets its state. The current gate composition is reinitialized as the 5×5 identity matrix, the action history (gate stack) is cleared, and the episode length is randomized between a specified range. Initial error metrics—for leakage, unitarity, and closeness to target—are computed and stored for later reward comparisons.

Algorithm 3 Reset and State Initialization

procedure RESET
    Call parent reset (with optional seed) RESETCOMPOSITION
    Set `current_step` $\leftarrow 0$
    Randomly select `random_episode_length` in [20, max_length]
    ($leak$, uni, $close$, _) $\leftarrow$ COMPUTEREWARD
    Store `prev_total_error` $\leftarrow \alpha \cdot leak + \beta \cdot close + \gamma \cdot uni$
    return (Current observation, empty info)
end procedure
1ex
procedure RESETCOMPOSITION
    `current_composition` $\leftarrow$ 5×5 identity matrix (complex)
    `current_length` $\leftarrow 0$
    `gate_stack` $\leftarrow$ empty list
end procedure

5.3.3 Executing an Action and Updating the Composition

When an action is taken, the environment verifies that the maximum sequence length has not been exceeded. If allowed, the corresponding braid gate is multiplied to update the current composition. The action is then recorded in the gate stack.

Algorithm 4 Action Execution

procedure TAKEACTION(*action*)
    if `current_length` $<$ `max_length` **then**
        `current_composition` $\leftarrow$ `current_composition` $\times$ `braid_gates`[action]
        Append string representation of *action* to `gate_stack`
        Increment `current_length`
    else
        Print: "Warning: Maximum composition length reached."
    end if
end procedure

5.3.4 Computing the Reward

The reward function evaluates the three key errors:

- `leakage_error`,
- `closeness_error`,
- `unitarity_error`

These errors are then combined with user-defined weights, and the reward is defined as their negative weighted sum as above

$$R \;=\; -\Big(\alpha\ \text{leakage\_error} \;+\; \beta\ \text{closeness\_error} \;+\; \gamma\ \text{unitarity\_error}\Big),$$

Algorithm 5 Reward Computation

procedure ComputeReward
  Let $M \leftarrow$ `current_composition`, $T \leftarrow$ `target_gate`
  `leakage` $\leftarrow |M[0,0]|$; apply safeguard if `leakage` $< 1e{-}12$
  `leakage_error` $\leftarrow 1.0 -$ `leakage`
  Define $M_{4\times4} \leftarrow M[1:5, 1:5]$
  `unitarity_error` $\leftarrow$ Schatten-1 norm of $(M_{4\times4}^{\dagger} M_{4\times4} - I)$
  if `local_equivalence_class` is enabled **then**
    `closeness_error` $\leftarrow$ LocalEquivalenceDistance(T, $M_{4\times4}$)
  else
    Normalize $M_{4\times4}$ and T by their Schatten-2 norms
    `closeness_error` $\leftarrow$ Schatten-2 norm of (normalized $M_{4\times4} - T$)
  end if
  `reward` $\leftarrow -\big(\alpha \cdot$ `leakage_error` $+ \beta \cdot$ `closeness_error` $+ \gamma \cdot$ `unitarity_error`$\big)$
  return (`leakage`, `closeness_error`, `unitarity_error`, `reward`)
end procedure

5.3.5 The Step Function: Integrating Action and Reward

The `step` function ties together the action execution and reward updates. It first records the current total error, applies the chosen action, and then recomputes the error metrics. The improvement in total error provides the immediate reward. If the episode length has been reached, a final bonus reward is added, and additional diagnostic information (such as gate invariants) is assembled.

Algorithm 6 Step Function

procedure Step(*action*)
  old_error $\leftarrow$ `prev_total_error`
  TakeAction(*action*)
  Increment `current_step`
  (*leak*, *close*, *unit*, *combined_reward*) $\leftarrow$ ComputeReward
  new_error $\leftarrow \alpha \cdot (1.0 - leak) + \beta \cdot close + \gamma \cdot unit$
  improvement $\leftarrow$ *old_error* $-$ *new_error*
  Set *step_reward* $\leftarrow$ *improvement*
  Update stored error metrics with current values
  if `current_step` $\geqslant$ `random_episode_length` **then**
    terminated $\leftarrow$ *true*
    step_reward $\leftarrow$ *step_reward* $+$ *combined_reward*
    Compute final invariants (e.g., g_1, g_2, g_3) from the final submatrix
    Populate an info structure with step metrics and gate history
  else
    terminated $\leftarrow$ *false*
  end if
  obs $\leftarrow$ GetObservation
  return (*obs*, *step_reward*, *terminated*, *truncated* = false, *info*)
end procedure

5.3.6 Extracting the Observation

Finally, the `GetObservation` method converts the current 5×5 complex gate matrix into a 50-dimensional vector by concatenating its real and imaginary parts.

Algorithm 7 Observation Extraction

procedure GetObservation
    Flatten real and imaginary parts of `current_composition` into vector *obs*
    return *obs*
end procedure

5.4 Helper Functions for `GateApproxEnv`

The above class makes use of several key functions which we will now describe.

5.4.1 Schatten p-Norm Calculation

The function `schatten_p_norm` computes the Schatten p-norm of a matrix T using its singular values. It first ensures that T is treated as a complex NumPy array, then uses singular value decomposition to extract the singular values. Depending on the value of p, it returns:

- The maximum singular value if $p = \infty$,
- The sum of the singular values if $p = 1$, or
- $(\sum \sigma^p)^{1/p}$ for other values of p.

Algorithm 8 Schatten p-Norm Calculation

procedure SchattenPNorm(T, p)
    Convert T to a complex NumPy array
    $\sigma \leftarrow$ Singular values of T via SVD
    if $p = \infty$ **then**
        return $\max(\sigma)$
    else if $p = 1$ **then**
        return $\sum \sigma$
    else
        return $(\sum \sigma^p)^{\frac{1}{p}}$
    end if
end procedure

5.4.2 Makhlin Invariants Computation

The function `compute_makhlin_invariants` calculates three invariants (g_1, g_2, and g_3) for a unitary matrix U. To achieve this:

- A fixed transformation matrix Q is used to obtain $U_B = Q^\dagger U Q$.
- The Makhlin matrix is formed as $m_U = U_B^T U_B$.

- The traces of m_U and m_U^2 and the determinant of U are computed.
- A complex value is then calculated from these traces and the determinant; its real and imaginary parts form g_1 and g_2, while g_3 is determined by the difference between the square of the trace and the trace of the square, normalized appropriately.

Algorithm 9 Makhlin Invariants Computation

procedure ComputeMakhlinInvariants(U)
  $i \leftarrow 1j$ ▷ Complex unit
  Define

$$Q \leftarrow \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & i \\ 0 & i & 1 & 0 \\ 0 & i & -1 & 0 \\ 1 & 0 & 0 & -i \end{bmatrix}$$

  $U_B \leftarrow Q^\dagger U Q$
  $m_U \leftarrow U_B^T U_B$
  $tr\_mU \leftarrow \text{trace}(m_U)$
  $tr\_mU2 \leftarrow \text{trace}(m_U \cdot m_U)$
  $det\_U \leftarrow \det(U)$
  if $|det\_U| < 10^{-12}$ **then**
    Print: "Warning: Determinant is very small, adding regularization."
    Set $det\_U \leftarrow det\_U + 10^{-12}$
  end if
  $complex\_val \leftarrow \frac{(tr\_mU)^2}{16\, det\_U}$
  $g_1 \leftarrow \text{Re}(complex\_val)$
  $g_2 \leftarrow \text{Im}(complex\_val)$
  $g_3 \leftarrow \frac{(tr\_mU)^2 - tr\_mU2}{4\, det\_U}$
  return (g_1, g_2, g_3)
end procedure

5.4.3 Local Equivalence Distance

The function `local_equivalence_distance` measures the similarity between two gates (or their submatrices) E and U by comparing their Makhlin invariants. It computes the invariants for both matrices and then calculates the sum of the squared differences between corresponding invariants. This summation serves as a metric of how locally equivalent the two matrices are.

Algorithm 10 Local Equivalence Distance

procedure LocalEquivalenceDistance(E, U)
    $gE \leftarrow$ ComputeMakhlinInvariants(E)
    $gU \leftarrow$ ComputeMakhlinInvariants(U)
    $d \leftarrow 0$
    for each pair (e, u) in corresponding entries of gE and gU **do**
        $d \leftarrow d + |e - u|^2$
    end for
    return d
end procedure

5.4.4 Gate Composition Extraction

The function `get_gate_composition` constructs the final gate composition by processing a string of digits. Each digit in the string corresponds to the index of a braid gate. The function:

- Initializes a 5×5 identity matrix.
- Iterates over each character in the string, converting it to an integer.
- Validates the index and multiplies the current composition by the corresponding braid gate.

The resulting matrix represents the composite gate.

Algorithm 11 Gate Composition Extraction

procedure GetGateComposition(*gate_string*)
    composition ← 5×5 identity matrix (complex)
    for each character c in *gate_string* **do**
        Attempt to convert c to integer *gate_index*
        if conversion fails **then**
            Error: "Invalid character in gate string."
        end if
        if *gate_index* is out of bounds **then**
            Error: "Gate index out of bounds."
        end if
        Update: *composition* ← *composition*× `braid_gates`[*gate_index*]
    end for
    return *composition*
end procedure

5.5 Recording, Logging, and Data Visualization

We use a custom callback class (`InfoCallback`) to record environment-specific data during training. This includes logging leakage, closeness error, unitarity error, episode rewards, and episode lengths. The callback extracts these metrics from the `infos` dictionary at every step of the training and appends them to dedicated log lists. In addition to the callback class, several helper functions are provided to visualize these recorded metrics and to evaluate the policy performance.

5.5.1 Custom Callback: `InfoCallback`

The `InfoCallback` class inherits from `BaseCallback` and is used to log key metrics across episodes when running vectorized environments. The main points are:

- **Initialization:** Set up empty lists for logging leakage, closeness error, unitarity error, rewards, and episode lengths.
- **_on_step:** At each environment step, if an episode terminates (i.e., `done = True`), the callback extracts metrics from the `infos` dictionary of that environment instance and appends the values to the logs.
- **Data Saving/Loading:** The callback provides functions (`save_callback_data` and `load_callback_data`) to persist or restore the logged data using pickle.

Below is the pseudocode representation for the core methods of `InfoCallback`:

Algorithm 12 InfoCallback: Data Recording at Each Step

```
procedure OnStep
    dones ← list of done flags from locals['dones']
    infos ← list of info dictionaries from locals['infos']
    for each index env_idx in the range of number of environments do
        if dones[env_idx] is True then
            info ← infos[env_idx]                    ▷ Extract environment-specific metrics
            leakage ← info.get("leakage", None)
            closeness_error ← info.get("closeness_error", None)
            unitarity_error ← info.get("unitarity_error", None)
            final_reward ← info.get("final_reward", None)
            current_step ← info.get("current_step", None)
            if leakage ≠ None then
                Append leakage to leakage_log
            end if
            if closeness_error ≠ None then
                Append closeness_error to closeness_log
            end if
            if unitarity_error ≠ None then
                Append unitarity_error to unitarity_log
            end if
            if final_reward ≠ None then
                Append final_reward to episode_rewards
            end if
            if current_step ≠ None then
                Append current_step to episode_lengths
            end if
        end if
    end for
    return True                                      ▷ Continue training
end procedure
```

The saving and loading functions use Python's `pickle` module to persist the logged data to disk.

5.5.2 Plotting Functions for Training Metrics

Several helper functions are provided to visualize the metrics logged by the callback:

- `plot_training_metrics`: Creates subplots for the final episode reward, leakage, closeness error, and unitarity error.
- `plot_low_unitary_acceleration`: Converts unitarity errors into a cumulative score for episodes with low error and plots the progression.
- `plot_low_leakage_acceleration`: Similarly, computes and plots a cumulative score for achieving low leakage.

Below is a high-level pseudocode outline for `plot_training_metrics`:

Algorithm 13 Plot Training Metrics

```
procedure PlotTrainingMetrics(callback)
    episodes ← sequence from 1 to number of episodes (length of episode_rewards)
    Create a 2x2 subplot grid
    Plot final episode reward vs. episodes
    Plot leakage vs. episodes
    Plot closeness error vs. episodes
    Plot unitarity error vs. episodes
    Adjust layout and display the plots
end procedure
```

The other plotting functions, `plot_low_unitary_acceleration` and `plot_low_leakage_acceleration`, follow a similar pattern. They first convert the respective error logs into NumPy arrays, then compute a cumulative score based on whether the error metric is below a preset threshold, and finally plot the cumulative score against the episode index.

5.5.3 Policy Evaluation with Vectorized Environments

The `evaluate_policy_vec` function is designed to test the trained model on a vectorized environment by:

- Resetting all parallel environments.
- Running the model to predict actions and stepping through the environment.
- Upon completion of an episode in any environment, it extracts the gate composition and error metrics, printing them in a formatted table if certain criteria (e.g., low closeness error and high leakage) are met.

Below is a high-level pseudocode overview for `evaluate_policy_vec`:

Algorithm 14 Evaluate Policy on Vectorized Environment

procedure EVALUATEPOLICYVEC(env, model, num_sequences)
    Reset environment: $obs \leftarrow$ `env.reset()`
    $episodes\_collected \leftarrow 0$
    while $episodes\_collected < num\_sequences$ **do**
        $actions, \_ \leftarrow$ MODELPREDICT(obs)
        $obs, \_, dones, infos \leftarrow$ STEP(env, actions)
        for each environment index i **do**
            if $dones[i]$ is True **then**
                Increment $episodes\_collected$
                Retrieve $gate\_stack$ and error metrics from $infos[i]$
                if conditions on closeness error and leakage are met **then**
                    Print formatted metrics and the operator string
                end if
            end if
        end for
    end while
end procedure

6 Results

In this section, we present the outcomes of our investigation into approximating gates in the local equivalence class of the CNOT gate via a reinforcement learning framework. Our findings provide strong evidence in support of the conjecture put forth in [10], namely that as one aims to approximate a highly entangling gate such as the CNOT, the probability of leakage out of the computational subspace necessarily increases.

This observation highlights an inherent tradeoff: improving the entangling capability of the gate invariably leads to larger leakage, corroborating the theoretical predictions from [10] and further supported by [6]. Here, we detail the performance of our reinforcement learning algorithm, discuss the balance between precise entanglement and mitigation of leakage, and explore the implications of this tradeoff for quantum gate design in the Fibonacci anyon model.

6.1 Key Findings

Our experiments revealed several promising outcomes, particularly in reducing leakage (*leakage reduction*) and maintaining near-unitarity. However, they also underscored the difficulty—if not impossibility—of obtaining a fully entangling gate using braiding alone. The main findings are as follows:

- **Unitary Gate Synthesis:** The reinforcement learning agent demonstrated a robust ability to discover braid sequences whose associated 4×4 submatrix U is close to unitary. This is exemplified in Figure 1, where we list several near-unitary candidates and their corresponding braid sequences.[3]

[3] In our tables, the entries in the *Operator* column represent sequences of braid operations indexed from 0 to 9, where 0–4 denote the braid representations, and 5–9 denote their respective inverses.

No.	Unitarity	Operator
1	2.827e-15	6519049996916036410669661
2	5.109e-15	009094013301455540400115011 8
3	2.264e-15	4096459006656543061400100446 41
4	5.757e-15	954103169664413664866143163836401
5	5.995e-15	651041691641691143468111660569335136
6	4.506e-15	699609306931849046450116999600148966 1
7	3.770e-15	43100409146061944441410533009663106 09
8	2.217e-15	4040609691494068013340
9	2.246e-15	639434096063940149153864
10	2.257e-15	06696006911396410616610919 11
11	5.480e-15	996384344144965163963166699149636666 6
12	3.494e-15	64909044435616045960066440096431919 6
13	3.998e-15	505640669456913104614659949949 96
14	4.664e-15	56168466066464081161 46
15	2.921e-15	61496364666615991446001066111936061
16	3.795e-15	1151495944963931091106041
17	3.998e-15	494646130593313641041416
18	3.699e-15	336491949616916161665964456064109
19	4.181e-15	364196391044615684630004060606
20	7.330e-15	0845100406486834550696139660469649646443
21	3.519e-15	466335016159319199660136660090054149 06
22	3.775e-15	06560541441365136311446
23	5.132e-15	04946009604430694906606543804641

Figure 1: Evaluation of near-unitary approximations.

- **Leakage Minimization:** By heavily penalizing any amplitude leaving the intended four-dimensional subspace, we trained the agent to learn braids that effectively suppress leakage (see Figure 2). Most solutions stayed well below a pre-defined leakage threshold, indicating that, in practice, braiding-based operations can be engineered to avoid the auxiliary dimension for two-qubit gates.

- **CNOT Approximation:** Despite prior numerical hints [10, 6], our direct attempts at approximating the CNOT gate resulted in only suboptimal solutions (see Figure 3). While the agent consistently satisfied unitarity and leakage constraints, it struggled to approximate CNOT to a high fidelity.

- **Local Equivalence to CNOT:** When we shifted our objective to finding gates *locally equivalent* to CNOT—using Makhlin invariants [19]—performance improved significantly. As illustrated in Figure 4, our agent found multiple novel gates that satisfy these local equivalence criteria, thus demonstrating that approximation of CNOT *up to local operations* is far more tractable than directly approximating CNOT itself.

Interestingly, our numerical logs repeatedly converged to a performance plateau around 0.618, reminiscent of the inverse golden ratio $1/\phi$. This recurring plateau likely arises from the properties of the ρ_3 matrix in our representation of the braid group.

No.	Leakage	Operator
1	1.000	1916410494941186166940
2	1.000	154134969194409638061450191111141
3	1.000	90445913919000650001001103994686
4	1.000	6660014036600606685469360919916 46
5	1.000	116349614161495609396469601694941
6	1.000	33696646160646650010990146568560348109
7	1.000	160466964454505649396145941410664166 0516
8	1.000	30611491801061319145
9	1.000	350644436460540601669
10	1.000	00414695111441331064581 5390
11	1.000	5100106340394110491090619963 33
12	1.000	0900034103696691140965139
13	1.000	66164006400615460100160393 6
14	1.000	04195399019946645966546
15	1.000	569911406149449619591648396469010149613
16	1.000	116506600096966619608066116 9
17	1.000	10165694060301401099160131 6
18	1.000	63069604465963540696666419 0
19	1.000	10494911996316086869106136166506 39
20	1.000	00446139690641396409800060 03
21	1.000	115110496109096606190316
22	1.000	050380003096011156961
23	1.000	490066913916100090130440954696060954019 4

Figure 2: Evaluation of leakage metrics.

6.2 Comparative Analysis

We implemented and compared two primary reward-shaping schemes to elucidate the agent's behavior:

1. **Direct CNOT Optimization:** We penalized the Schatten norm difference between the agent's gate U and the ideal CNOT. Although the agent successfully minimized leakage and maintained near-unitarity, the final gate remained a poor approximation of CNOT.

2. **Local-Equivalence Optimization:** By employing a Makhlin-invariants-based measure [19], the agent successfully identified multiple gate sequences that lie in the local equivalence class of CNOT. In stark contrast to direct CNOT optimization, these solutions achieved significantly better fidelity while maintaining reasonable near-unitarity and suppressed leakage. Our results thus confirm that although direct braiding of an exact CNOT is challenging, allowing for local equivalences effectively broadens the space of achievable solutions and leads to high-quality entangling gates in this topological setting agreeing with [6].

6.3 Challenges and Limitations

While our reinforcement learning approach effectively identifies high-fidelity, low-leakage gates within the local equivalence class of CNOT, several persistent challenges remain:

No.	Closeness	Leakage	Unitarity	Operator
1	8.8963e-01	1.000	2.319e-15	944119048551811634350445504 51
2	8.7576e-01	1.000	1.673e-15	409548419993510481569191 58
3	8.9720e-01	1.000	3.200e-15	8850916995899808554535494
4	8.9886e-01	1.000	2.962e-15	463689419356959994155561149
5	8.4070e-01	1.000	2.390e-15	18499314988153488189 64
6	8.9746e-01	1.000	2.824e-15	5818611948800904501999
7	8.5080e-01	1.000	3.332e-15	955599495509588195590499415 8859
8	8.6246e-01	1.000	3.028e-15	890094155499409095154848
9	8.8296e-01	1.000	2.283e-15	58116189481494554548
10	8.6225e-01	1.000	2.023e-15	51855481540488918901
11	8.7715e-01	1.000	2.470e-15	913955515164951489558191509 9944
12	8.7888e-01	1.000	3.273e-15	0989399863099555958538131559915559 9880
13	8.9866e-01	1.000	2.455e-15	9045049548465545843158
14	8.6465e-01	1.000	3.694e-15	0995855940899895159161190 49091
15	8.7896e-01	1.000	3.535e-15	5499984913885689515090451131959599444455
16	8.0922e-01	1.000	2.781e-15	03598994945935805340
17	8.1908e-01	1.000	5.110e-15	01950499869088589940055
18	8.7251e-01	1.000	4.186e-15	34894930859105059951484405985548416
19	8.9746e-01	1.000	2.037e-15	994991851951508934010188
20	8.0922e-01	1.000	1.968e-15	1554995043800469061350
21	8.9040e-01	1.000	2.732e-15	45151145589598355119114898409
22	8.6246e-01	1.000	2.350e-15	40805934919514099859

Figure 3: Evaluation of CNOT closeness, leakage, and unitarity.

- **Combinatorial Explosion of Braids:** The space of possible braid sequences grows exponentially with sequence length, demanding extensive exploration and thus significantly complicating the search process. Methods to constrain or reduce this space are crucial for efficient learning.
- **Local Optima:** Consistent with most reinforcement learning paradigms, the agent occasionally converges prematurely to locally optimal braid sequences. Although these solutions can still be effective, escaping local optima often requires multiple restarts or more sophisticated exploration strategies.
- **Computational Overheads:** Despite parallelizing our experiments over multiple CPU cores, the exponential size of the action space keeps computational costs high. Although large-scale or extended searches may reveal even better gate sequences, further optimizations—such as GPU- or distributed-accelerated reinforcement learning—would likely be necessary to handle deeper explorations.

6.4 Conclusion and Future Directions

Our results indicate the potential for carefully tailored reinforcement learning algorithms to discover gates of high fidelity that are locally equivalent to entangling gates. This outcome highlights that, even if directly braiding a perfect CNOT gate appears unattainable under current configurations,

No.	Closeness	Leakage	Unitarity	Operator
1	1.202e-09	0.992	1.594e-02	373373739737937373373
2	1.953e-02	0.980	4.031e-02	37337373343000773323
3	8.928e-01	0.936	1.238e-01	373323393737332975373373373787477337373373
4	1.953e-02	0.980	4.031e-02	37337373343035223733
5	7.051e-02	0.982	3.506e-02	373373747337334323737
6	6.785e-02	0.984	3.207e-02	70747074804807470770
7	1.264e-02	0.990	1.962e-02	74715417175771747519
8	3.380e-06	0.944	1.084e-01	1245435375285864252955520411248
9	2.169e-05	0.976	4.760e-02	330486312180998922152223419785719 5
10	1.042e-07	0.943	1.110e-01	2522122520252241121222952
11	1.042e-07	0.943	1.110e-01	2122521290220262257522525
12	3.210e-08	0.952	9.282e-02	212212922221206520262657
13	3.380e-06	0.944	1.084e-01	252282524225252528922929522 52
14	6.144e-04	0.947	1.041e-01	2922023932292292292292022229292092 29
15	3.653e-05	0.881	2.242e-01	754596019601765960646554191474 7
16	1.344e-05	0.991	1.846e-02	22626246064426262606626209022 6
17	5.262e-05	0.994	1.193e-02	2460606626260262126206626
18	1.220e-05	0.992	1.654e-02	762620226242026442267
19	1.128e-04	0.993	1.476e-02	2262620246256252262446
20	2.434e-07	0.998	3.955e-03	22626212426066202292620262262626 22
21	1.237e-01	0.984	3.134e-02	7397799322
22	1.202e-09	0.992	1.594e-02	373373739737937373373
23	1.953e-02	0.980	4.031e-02	37337373343000773323
24	9.661e-03	0.975	4.943e-02	2262620207742620262
25	1.220e-05	0.992	1.654e-02	20646764602426066276469262020 2
26	5.077e-09	0.986	2.840e-02	22626202066267640262262901202662
27	9.999e-05	0.981	3.753e-02	22626262020662620962262
28	1.344e-05	0.991	1.846e-02	22626246064426262606626242026
29	1.344e-05	0.991	1.846e-02	2262624606442626260662624242
30	9.999e-05	0.981	3.753e-02	22626246920206626209622679 22

Figure 4: Evaluation of gates locally equivalent to CNOT.

allowing for local unitary transformations substantially broadens the space of viable solutions. Importantly, the agent's ability to maintain near-unitarity and control leakage highlights the potential of machine learning as a powerful tool for topological quantum gate compilation.

6.4.1 Future Directions

Several promising research paths arise from these findings:

- **Augmented State Space:** Investigate the effect of further increasing the number of anyons or adding ancillary qubits. Such expansions may relax constraints on the five-dimensional fusion space, potentially enabling more robust entangling operations while maintaining leakage suppression.
- **Canonical Braid Reduction:** Embedding group-theoretic or combinatorial simplifications

into the agent's action space and environment transitions can eliminate physically equivalent but distinct braids, streamlining the search and improving both run time and interpretability of solutions.

- **Alternative Reinforcement Learning Methods:** Explore multi-objective optimizers or evolutionary algorithms designed to escape local minima more reliably, thereby uncovering subtle braiding patterns that satisfy multiple performance criteria (e.g., entangling power, unitarity, and leakage).
- **Extensions to Other Anyon Models:** Although Fibonacci anyons remain a prime candidate for universal topological quantum computation, investigating other non-Abelian anyon models—for instance, Ising anyons—could reveal whether our current challenges and successes generalize to different topological systems.

6.4.2 Final Remarks

In summary, the ability to identify gates locally equivalent to CNOT—while preserving unitarity and keeping leakage low—underscores the effectiveness of reinforcement learning in topological quantum compilation, even in a constrained, six-anyon environment. Although purely braiding a direct CNOT in this five-dimensional fusion space may remain elusive, our work demonstrates that, by broadening the objective to local equivalences, one can achieve highly entangling operations. Continual refinements of these machine learning approaches and their extension to larger systems or alternative anyon models may open new pathways toward realizing universal topological quantum computation.

References

[1] S. Awodey. *Category Theory*. Oxford Logic Guides. Ebsco Publishing, 2006. ISBN: 9780191513824. URL: https://books.google.ie/books?id=IK_sIDI2TCwC.

[2] B. Bakalov and A.A. Kirillov. *Lectures on Tensor Categories and Modular Functors*. Translations of Mathematical Monographs. American Mathematical Society, 2001. ISBN: 9780821826867. URL: https://books.google.ie/books?id=-TfyBwAAQBAJ.

[3] Bruce Bartlett et al. *Modular categories as representations of the 3-dimensional bordism 2-category*. 2015. arXiv: 1509.06811 [math.AT]. URL: https://arxiv.org/abs/1509.06811.

[4] Ruben Van Belle. *Probability monads as codensity monads*. 2022. arXiv: 2111.01250 [math.CT]. URL: https://arxiv.org/abs/2111.01250.

[5] Eric Benhamou. *Variance Reduction in Actor Critic Methods (ACM)*. 2019. arXiv: 1907.09765 [cs.LG]. URL: https://arxiv.org/abs/1907.09765.

[6] Phillip C. Burke et al. "Topological quantum compilation of two-qubit gates". In: *Physical Review A* 110.5 (Nov. 2024). ISSN: 2469-9934. DOI: 10.1103/physreva.110.052616. URL: http://dx.doi.org/10.1103/PhysRevA.110.052616.

[7] Matteo Capucci et al. "Towards Foundations of Categorical Cybernetics". In: *Electronic Proceedings in Theoretical Computer Science* 372 (Nov. 2022), pp. 235–248. ISSN: 2075-2180. DOI: 10.4204/eptcs.372.17. URL: http://dx.doi.org/10.4204/EPTCS.372.17.

[8] Bryce Clarke et al. "Profunctor Optics, a Categorical Update". In: *Compositionality* 6 (Feb. 2024), p. 1. ISSN: 2631-4444. DOI: 10.32408/compositionality-6-1. URL: http://dx.doi.org/10.32408/compositionality-6-1.

[9] Stable Baselines3 Contributors. *Recurrent PPO — Stable Baselines3 - Contrib 2.6.0 documentation*. Accessed: 2025-04-10. 2025. URL: https://sb3-contrib.readthedocs.io/en/master/modules/ppo_recurrent.html.

[10] Shawn X. Cui et al. *The Search For Leakage-free Entangling Fibonacci Braiding Gates*. 2019. arXiv: 1904.01731 [quant-ph]. URL: https://arxiv.org/abs/1904.01731.

[11] P. Etingof et al. *Tensor Categories*. Mathematical Surveys and Monographs. American Mathematical Society, 2016. ISBN: 9781470434410. URL: https://books.google.ie/books?id=Z6XLDAAAQBAJ.

[12] Michael Freedman, Michael Larsen, and Zhenghan Wang. *A modular functor which is universal for quantum computation*. 2000. arXiv: quant-ph/0001108 [quant-ph]. URL: https://arxiv.org/abs/quant-ph/0001108.

[13] Neil Ghani et al. *Compositional game theory*. 2018. arXiv: 1603.04641 [cs.GT]. URL: https://arxiv.org/abs/1603.04641.

[14] Marek Grzes. "Reward Shaping in Episodic Reinforcement Learning". In: *Sixteenth International Conference on Autonomous Agents and Multiagent Sytems (AAMAS 2017)*. ACM, May 2017, pp. 565–573. URL: https://kar.kent.ac.uk/60614/.

[15] Jules Hedges and Riu Rodríguez Sakamoto. *Reinforcement Learning in Categorical Cybernetics*. 2024. arXiv: 2404.02688 [cs.LG]. URL: https://arxiv.org/abs/2404.02688.

[16] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[17] David Jaz Myers. "Double Categories of Open Dynamical Systems (Extended Abstract)". In: *Electronic Proceedings in Theoretical Computer Science* 333 (Feb. 2021), pp. 154–167. ISSN: 2075-2180. DOI: 10.4204/eptcs.333.11. URL: http://dx.doi.org/10.4204/EPTCS.333.11.

[18] Vijay Konda and John Tsitsiklis. "Actor-Critic Algorithms". In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 1999. URL: https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[19] Yuriy Makhlin. In: *Quantum Information Processing* 1.4 (2002), pp. 243–252. ISSN: 1570-0755. DOI: 10.1023/a:1022144002391. URL: http://dx.doi.org/10.1023/A:1022144002391.

[20] Eugenio Moggi. "Notions of computation and monads". In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 55–92. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(91)90052-4. URL: https://www.sciencedirect.com/science/article/pii/0890540191900524.

[21] Mario Román. *Monoidal Context Theory*. 2024. arXiv: 2404.06192 [math.CT]. URL: https://arxiv.org/abs/2404.06192.

[22] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: 1511.05952 [cs.LG]. URL: https://arxiv.org/abs/1511.05952.

[23] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018. arXiv: 1506.02438 [cs.LG]. URL: https://arxiv.org/abs/1506.02438.

[24] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347.

[25] R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018. ISBN: 9780262352703. URL: https://books.google.ie/books?id=uWV0DwAAQBAJ.

[26] Jun Zhang et al. "Geometric theory of nonlocal two-qubit operations". In: *Physical Review A* 67.4 (Apr. 2003). ISSN: 1094-1622. DOI: 10.1103/physreva.67.042313. URL: http://dx.doi.org/10.1103/PhysRevA.67.042313.