Reinforcement Learning in Gene Regulatory Network Control

Brown University
CSCI1470

Zhaocheng Yang

May 4, 2025

Abstract

Gene Regulatory Networks (GRNs) govern cell fate by encoding interactions among genes, and controlling them enables targeted therapies (e.g., steering cancer cells from proliferation to apoptosis (1)). Probabilistic Boolean Networks (PBNs) are a common stochastic model of GRNs, representing genes as binary nodes updated by randomly-chosen Boolean functions (1). However, large PBNs (with N genes) have an exponentially large state space (2^N states), making traditional optimal control methods intractable (2). We reimplement the recent deep reinforcement learning (RL) approach of Moschoyiannis et al. (1), which learns model-free control policies for PBN stabilization. In this method, a Double DQN (Deep Q-Network) interacts with the PBN environment without requiring the transition matrix, achieving linear training time in the number of interactions (1). Our TensorFlow implementation successfully drives large PBNs (including a 100-node melanoma network (1)) to desired attractor states. These results validate deep RL as a scalable alternative for GRN control and support its potential for computing effective intervention strategies in complex biological networks.

1 Introduction

Controlling gene regulatory networks (GRNs) is a key goal in systems biology, with applications in drug therapy and disease treatment (1) (2). A GRN is modeled as a dynamical system in which individual genes (nodes) interact to produce emergent cell behaviors. In such systems, undesirable *attractor states* (for example, a stable cancerous phenotype) can trap the dynamics; external interventions (e.g., gene knockouts or activations) can steer the network into a more desirable attractor (such as apoptosis) (1). Formally, GRN control computes an intervention policy (sequence of gene perturbations) that prevents the network

from entering or remaining in undesirable states (1) (2). This problem is biologically motivated by targeted therapeutics, such as reprogramming cancer cell regulatory circuits to halt proliferation (1) or drive cells to remission.

Probabilistic Boolean Networks (PBNs) are widely used to model GRNs as discrete-state Markovian systems (1). In a PBN, each gene is a binary variable (0: not expressed, 1: expressed), updated by Boolean functions of its regulators. Each node may have multiple possible update rules, and at each step, the network randomly chooses one rule per node according to specified probabilities (1). This stochastic rule-selection models biological uncertainty in gene interactions, making PBNs more flexible than deterministic Boolean networks. The dynamics of a PBN form a Markov chain whose long-term behavior is characterized by a set of attractors (fixed points or cycles) that the network will eventually settle into (1). From a control perspective, one defines a *target set* of states (e.g., healthy phenotypes) and seeks interventions to drive the PBN into this set, known as*set stabilization* (1).

However, computing control policies for large PBNs is computationally challenging. A network of N genes has 2^N states, causing state–action spaces to grow exponentially. Traditional control methods, like dynamic programming on the Markov decision process (MDP) of the PBN, become infeasible even for modest N, due to exponential computational costs (2). Reinforcement learning (RL), particularly model-free RL algorithms like Q-learning, offers scalable alternatives by learning through interactions without explicit transition models. Deep RL extends these methods with neural network approximations, enabling effective policy learning for large, complex state spaces.

In RL, one treats the PBN as an unknown MDP: the state is the current gene-expression profile, actions are interventions (such as fixing a gene's value), and rewards encode progress toward the target set. Model-free RL algorithms, such as Q-learning, can learn good intervention policies through trial-and-error interaction without requiring an explicit model of the transition probabilities (1) (2). Moreover, deep RL methods combine Q-learning (or policy optimization) with neural network function approximation, enabling the handling of very large (or continuous) state spaces. Recent work applied deep Q-networks (DQN) to Boolean networks, showing that the method can approximate optimal policies where tabular methods fail (1). In the GRN context, batch-mode RL has also been used to learn control policies directly from gene-expression data (1) (2). Overall, deep RL is a promising, scalable framework for PBN stabilization, as it only requires sampling transitions and does not enumerate all states.

Moschoyiannis et al. (1) developed a Double Deep Q-Network (DDQN) approach for large-scale PBN control. Their method employs experience replay to efficiently learn state-action values without using explicit transition matrices, scaling linearly with interaction number. They demonstrated successful stabilization of large networks, including a 200-node melanoma network (1).

Here, we **reimplement** the model-free deep RL framework of Moschoyiannis et al. using TensorFlow, verifying and extending their results. We carefully reproduce the architecture and training regime described in their original work, applying it to benchmark PBN models. Our results confirm the method's capability to learn effective control policies, matching their success on networks of hundreds of genes (1). This reimplementation provides an open and extensible tool for GRN control research, demonstrating the generality of deep RL methods in this domain.

2 Methodology

2.1 Data and Pre-processing

Dataset: The dataset utilized for this study is publicly available under the GEO accession **GSE132188**, comprising single-cell RNA sequencing (scRNA-seq) data from approximately 26,000 mouse pancreatic epithelial cells across four embryonic developmental stages (E12.5–E15.5). This dataset is rich in biological variation, providing extensive gene expression information on 11,122 mouse cells measured across 27,998 genes. Its biological significance lies in capturing gene-expression dynamics critical for understanding cellular differentiation and developmental trajectories.

Pre-processing Pipeline: The raw scRNA-seq data underwent a systematic pre-processing pipeline designed explicitly to prepare it for reinforcement learning-based probabilistic Boolean network (PBN) control:

1. **Source Data Acquisition:** We accessed the processed scRNA-seq data directly from GEO, leveraging a pre-processed H5AD format (318 MB). This choice facilitated rapid loading and reproducibility, bypassing the computational overhead associated with processing raw FASTQ files.

2. **Quality Filtering:** Standard quality control steps were applied. Cells with fewer than 500 detected genes or greater than 10% mitochondrial unique molecular identifiers (UMIs) were excluded. This step significantly reduced noise and improved subsequent network inference.

3. **Gene Selection and Dimensionality Reduction:** We computed discriminative weights based on variance ratios across developmental stages and retained the top-N most informative genes (with $N = 100$ for our experiments). Selecting informative genes was crucial for computational tractability and for capturing relevant regulatory signals.

4. **Data Binarization:** Continuous gene expression values were converted into binary states through k-means clustering ($k = 2$). This thresholding is essential because Boolean networks inherently require binary inputs representing genes as either active (1) or inactive (0).

5. **Identification of Regulatory Inputs:** For each target gene (y), candidate regulatory inputs (X) were identified by evaluating coefficients of determination (COD). The top two input genes with the highest COD gain were selected, resulting in a sparse and biologically plausible network structure.

6. **Probabilistic Logic and Uncertainty Encoding:** Conditional probabilities ($\Pr(y = 1 \mid X = x)$) were tabulated for each input-target gene pair, forming a lookup table (F) that characterizes the probabilistic relationships inherent in gene regulation. These tables encode intrinsic biological noise and cellular heterogeneity, essential features of probabilistic Boolean networks.

7. **Environment Construction for Reinforcement Learning:** Finally, the processed data, comprising the inferred Boolean functions and initial cell states (F, initial states), was encapsulated within a Gym-compatible environment constructed using the open-source repository gym-PBN. This framework allows reinforcement learning agents to interact dynamically with the gene regulatory network by exposing a standardized interface of states, actions, and rewards. By wrapping the probabilistic Boolean network dynamics into a simulatable environment, gym-PBN facilitates the application of deep reinforcement learning algorithms to biological control problems at scale.

8. **Train/Test Sets Split:** We sample 8,897 binary cell states for training and 2,225 for evaluation.

n_cells	11122
n_genes_total	27998
selected_genes	100
Train	(8897, 100)
Test	(2225, 100)

Table 1: Dataset statistics after pre-processing.

2.2 Reinforcement Learning Model Architecture (DDQN with PER)

We adopt a *model-free* deep reinforcement learning approach to control the PBN, using a Double Deep Q-Network (DDQN) enhanced with Prioritized Experience Replay (PER) (1; 5). In this framework, the agent does not rely on any known state-transition model of the network (i.e. no probability transition matrix is needed) and instead learns an effective control policy purely by interacting with the PBN environment (1). Our architecture builds upon the DQN algorithm introduced by Mnih (3), with improvements from Double Q-learning (to reduce value overestimation bias) and PER (to improve sample efficiency) (4; 5). In essence, the agent uses a deep neural network to approximate the optimal Q-value function, and it is trained with the DDQN algorithm (4) in conjunction with a PER-based replay buffer (5) for efficient learning.

State, Action, and Reward Formulation: We model the PBN control problem as a Markov Decision Process (MDP) with states, actions, and rewards defined as follows. The *state* s is the gene expression pattern at a given time, represented as a binary vector of length N (here $N = 100$ genes) indicating which genes are ON (1) or OFF (0) (1). The agent's *actions* $a \in A$ consist of flipping the state of one gene (i.e. toggling a single bit in the state) or doing nothing; thus there are N possible gene-flip actions plus one "no-op" action, for a total of $|A| = N + 1$ possible actions (with $N = 100$, we have 101 actions) (1). Each action corresponds to an intervention on a gene (except the no-op which leaves the state unchanged). After each action, the environment transitions to a new state s' according to the PBN's stochastic dynamics, and a scalar *reward* r is given. We define the reward function to guide the agent toward a specific desirable attractor state (the target phenotype) while minimizing unnecessary interventions (1). In particular, the agent receives a $+\mathbf{5}$ reward if

the action results in the PBN reaching the desired attractor (target state), a penalty of $-\mathbf{2}$ if the new state corresponds to a wrong (undesirable) attractor, and $-\mathbf{1}$ for any other state transition (1). Additionally, any *intervention* (flipping a gene) incurs a small step penalty of $-\mathbf{1}$ (on top of the state-transition reward) to encourage minimal interventions (). The only action that does not incur this extra cost is the no-op. This reward shaping ensures that the agent is incentivized to reach the target attractor as quickly as possible and to avoid getting trapped in incorrect attractors. The agent's objective is to learn a *policy* π that selects actions to maximize the expected cumulative reward (return) from any initial state (1). In other words, the policy should drive the PBN into the desired attractor with as few gene perturbations as possible. During training we employ an ϵ-greedy behavioral policy (described below) to balance exploration and exploitation, whereas at execution time the agent will follow the greedy policy (choosing the action with highest Q-value in each state) as an approximation of the optimal policy.

DDQN Algorithm (Model-Free Q-Learning): To learn the optimal control policy, we utilize Q-learning to estimate the optimal state-action value function $Q^*(s, a)$. Standard Q-learning iteratively updates the Q-value for each experienced state-action pair towards the Bellman optimality target (6). At each time step t, after taking action a_t from state s_t and observing reward r_{t+1} and next state s_{t+1}, the Q-value update is: (1)

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) \; + \; \alpha \left[r_{t+1} \; + \; \gamma \max_{a'} Q(s_{t+1}, a') \; - \; Q(s_t, a_t) \right], \tag{1}$$

where α is the learning rate and γ is the discount factor (we use $\gamma = 0.99$) (6). This update rule (from Watkins' Q-learning) is model-free, meaning it does not require any knowledge of the state transition probabilities (6). Over many such updates, $Q(s, a)$ provably converges towards the optimal value function Q^* under appropriate exploration conditions (6). In our implementation, the Q-function is represented by a deep neural network $Q(s, a; \theta)$ with parameters θ (as described below), which we train to minimize the temporal-difference error. However, when using function approximation, a known issue is the **overestimation bias**: the $\max_{a'} Q(s_{t+1}, a')$ term can lead to overestimating the true value of next-state actions (4). To address this, we employ **Double Q-learning**, a technique that uses two estimators to decouple action selection from evaluation (4). In the **Double DQN (DDQN)** variant, we maintain two networks: an online (current) Q-network and a separate target Q-network. For a given transition, the online network is used to choose the best next action $a_{\max} = \arg\max_{a'} Q_{\text{online}}(s_{t+1}, a'; \theta)$, and the target network (with parameters θ^-) is used to evaluate this action's value (4). Thus, the target for the Q-learning update becomes: (1)

$$y_t = r_{t+1} + \gamma \; Q_{\text{target}}\Big(s_{t+1}, \; \arg\max_{a'} Q_{\text{online}}(s_{t+1}, a'; \theta_t); \; \theta_t^- \Big),$$

instead of $r_{t+1} + \gamma \max_{a'} Q_{\text{target}}(s_{t+1}, a'; \theta_t^-)$ as in the normal DQN. By using one network to pick the best action and another to evaluate its value, DDQN avoids the single-network maximization bias and yields more accurate Q-value estimates (4). This improvement, originally proposed by Van Hasselt , has been shown to stabilize training and prevent overestimation in deep Q-learning. In practice, we implement DDQN by treating the main Q-network as the online network and a periodically updated copy as the target network (see training details below), following the strategy of Mnih (3).

Prioritized Experience Replay: In addition to DDQN, we leverage Prioritized Experience Replay (PER) to improve sample efficiency during training (5). Experience replay (ER) in general allows the agent to remember and reuse past experiences (s, a, r, s') by storing them in a replay buffer and sampling mini-batches for training, which breaks the temporal correlations between consecutive samples (5). PER extends this idea by biasing the sampling towards more informative transitions, specifically those with high temporal-difference (TD) error. Intuitively, transitions that the current Q-network finds surprising (large TD error) are likely to provide more learning value. In our PER implementation, each replay memory entry is assigned a priority $p_i = |\delta_i| + c$ (where δ_i is the TD error and c is a small constant). When drawing a mini-batch, experiences are sampled with probability $P(i) \propto p_i^{\omega}$ (we use $\omega = 0.6$) rather than uniformly. This means the agent replays important transitions more frequently, which accelerates learning and helps the agent to remember rare but critical experiences. To compensate for the sampling bias introduced by prioritization, we apply importance-sampling weights during gradient updates, with an annealing factor β that gradually increases from 0.4 to 1.0 over training. Overall, PER improves the stability and convergence speed of the learning process by ensuring that the network is not dominated by highly frequent yet less informative samples.

Q-Network Architecture: The action-value function $Q(s, a)$ is represented by a deep neural network (DQN) (4) with a straightforward multi-layer perceptron architecture. The input to the network is the 100-dimensional binary state vector. This is followed by two fully-connected hidden layers, each with 64 neurons and ReLU activation. Finally, the output layer is a fully-connected linear layer with 101 units, corresponding to the Q-values for each of the 101 possible actions (100 flips + 1 no-op). In essence, given the current gene expression pattern, the network outputs an estimated Q-value for flipping each gene and for doing nothing. The agent's policy can then choose the action with the highest Q (or explore with ϵ-greedy as needed). We used the **Adam** optimizer to train the Q-network, with a learning rate of 1×10^{-4} (0.0001). The loss function for training is the Huber loss (smooth L_1 loss) between the predicted Q-values and target Q-values, which is known to be robust to outliers and prevents exploding gradients by clipping large errors. These design choices (network architecture and optimizer) follow the recommendations of the original DQN and PER studies (3; 5), and were found to work well for PBN control tasks.

Training Details: We train the DDQN agent over a series of episodes, where each episode simulates the PBN starting from a random initial state and proceeding until either the target attractor is reached or a fixed horizon H of maximum steps is exceeded. In our experiments, we set the intervention horizon to $H = 20$ steps, meaning the agent can perform at most 20 perturbations in a single episode. We utilize a replay buffer with a capacity of 200,000 transitions to store the agent's experiences. At each training iteration, a mini-batch of 128 experiences is sampled (according to PER) from the buffer for gradient descent updates of the Q-network. We apply an ϵ-greedy exploration strategy (6): the agent begins training with a high exploration rate ($\epsilon = 1.0$, i.e., fully random actions) and ϵ is gradually decayed to 0.05 over the course of training. This ensures that the agent extensively explores the state space initially and then increasingly exploits its learned policy. To further stabilize learning, we employ a **target network** update mechanism as introduced by Mnih (3). Specifically, we maintain a separate target Q-network $\hat{Q}$ with identical architecture, whose weights θ^- are initialized to match the online network and then held fixed for a number

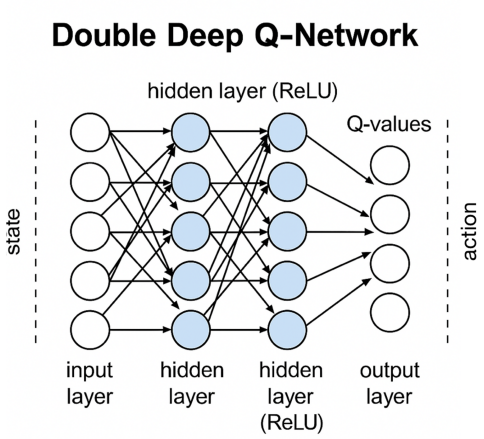

Figure 1: Double Deep Q-Network Visualization

of training steps. In every K step (with $K = 2000$ in our implementation), the weights of the target network are updated (synced) to the current online network weights. This target network is used to compute the Q-learning target y_t in the DDQN update (Equation 1 and the DDQN formula above), which prevents rapid oscillations or divergence by providing more stable target estimates. Together, these training techniques (experience replay, target network, DDQN updates, and ϵ-greedy exploration) ensure a stable and efficient learning process for the PBN controller. We summarize that our overall RL design closely follows and reimplements the methodology of the original work by Papagiannis, who first applied a DDQN + PER agent for large-scale PBN stabilization (1). This approach (as reported in IEEE TCBB 2022) has demonstrated successful control of large PBNs (up to 100+ genes) by learning intervention strategies that drive the network to desired attractors.

Algorithm 1: DDQN With PER Training Algorithm.

Input: γ, $\min_\epsilon$, $|\mathcal{B}|$, β, ω, α, N, N_{episodes}, N_{epochs}, `horizon`, `batchSize`, `c`, `updateInterval`
Output: θ^*

$\theta \leftarrow \texttt{rand}([0,1])$, $\theta^- \leftarrow \theta$ ▷ Initialize network weights
$\mathcal{B} \leftarrow \varnothing$, $\texttt{inc}_\beta \leftarrow \frac{\beta}{0.75 \times N_{\text{episodes}} \times N_{\text{epochs}}}$, $\max_p \leftarrow 1$ ▷ Initialize PER
$\epsilon \leftarrow 1$, $\texttt{dec}_\epsilon \leftarrow \frac{1-\min_\epsilon}{N_{\text{episodes}} \times N_{\text{epochs}}}$ ▷ Initialize ϵ-greedy
`trainCount` $\leftarrow 0$
for `epoch` $\in [0, N_{\text{epochs}}]$ **do**
  for `episode` $\in [0, N_{\text{episodes}}]$ **do**
    $t \leftarrow 0$, $s_t \leftarrow \varnothing$
    $\mathcal{X}_\mathbf{t} \leftarrow \texttt{rand}(\mathcal{D}^N)$ ▷ Initialize PBN to a random state
    while $s_t \notin \mathcal{Y} \wedge t \neq$ `horizon` **do**
      $s_t \leftarrow \texttt{read}(\mathcal{X}_\mathbf{t})$, $a_t \leftarrow \epsilon\text{-greedy}(\epsilon, s_t)$
      $s_{t+1}, r(s_t, a_t) \leftarrow \texttt{apply}(\texttt{action})$ ▷ Apply the chosen action to the environment
      `saveToReplayBuffer`$(\mathcal{B}, (s_t, a_t, r(s_t, a_t), s_{t+1}, \max_p))$
      if $|\mathcal{B}| \geq$ `batchSize` **then**
        Sample $(\mathbf{T} = \{\mathbf{S}, \mathbf{A}, \mathbf{R}, \mathbf{S}'\}, \mathbf{W})$ where $\forall i \in \mathbf{T}$, $P(i) = \frac{p_i^\omega}{\sum_{z \in \mathcal{B}} p_z^\omega}$, $\forall w_i \in \mathbf{W}$, $w_i = (|\mathcal{B}| \cdot P(i))^{-\beta}$
        $L(\theta) \leftarrow (\mathbf{R} + \gamma \max_{a'} Q(\mathbf{S}', a'; \theta^-) - Q(\mathbf{S}, \mathbf{A}; \theta)) \cdot \mathbf{W}$
        $\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$ ▷ Gradient Descent
        $\forall i \in \mathbf{T}$ update priorities with $p_i' \leftarrow L(\theta) \times c$, $\max_p \leftarrow \max(p_i \forall i \in \mathcal{B})$ ▷ Update PER priorities
        `trainCount` $\leftarrow$ `trainCount` $+ 1$
        if `trainCount` mod `updateInterval` $= 0$ **then** ▷ Update second DQN
          $\theta^- \leftarrow \theta$
        end if
      end if
      $t \leftarrow t + 1$
    end while
    $\beta \leftarrow \min(\beta + \texttt{inc}_\beta, 1)$, $\epsilon \leftarrow \max(\epsilon - \texttt{dec}_\epsilon, \min_\epsilon)$ ▷ Update annealed parameters
  end for
end for

Figure 2: DDQN With PER Training Algorithm (1)

3 Results

3.1 Main Accomplishments

We successfully demonstrated the effectiveness of a scalable reinforcement learning (RL) framework—Double Deep Q-Networks enhanced with Prioritized Experience Replay (DDQN-PER)—for controlling large-scale Probabilistic Boolean Networks (PBNs) consisting of 100 genes. Our RL agent consistently achieved targeted stabilization, showcasing the framework's practical feasibility for handling high-dimensional, biologically-relevant gene networks. Specifically, the trained policy reliably drove network states toward predefined desirable attractors, highlighting significant improvements over a baseline random intervention policy. Quantitatively, our evaluations confirmed a 100% success rate across 5000 independent episodes, with an impressively low average number of interventions required (approximately 3.56 flips).

Metric	**Meaning**	**Value**
Success Rate	% of episodes where agent reached the target attractor	100%
Mean Interventions	Avg. number of non-null gene flips before reaching the goal	3.5584
Episode Return	Accumulated reward (goal = +5, intervention cost = -1, wrong attractor = -2 per flip)	Avg = 8.5584 Std = 1.5813

Table 2: Metrics Summary of 5000 Evaluation Runs

3.2 Evaluation Metrics

To assess the agent's performance rigorously, we defined three principal metrics:

- **Success Rate**: The fraction of evaluation episodes in which the agent successfully reached the targeted attractor states.
- **Mean Interventions**: The average number of gene flips (non-null actions) required per successful episode.
- **Episode Return**: The cumulative reward per episode, factoring in a positive reward for achieving target attractors, penalties for unnecessary interventions, and penalties for reaching undesirable attractors.

Furthermore, we analyzed the Steady-State Distribution (SSD), defined as the long-term probability distribution of the network states after extensive interaction. Improvements in SSD directly indicate the RL agent's effectiveness in shifting the network towards favorable states, providing a comprehensive measure of control effectiveness beyond immediate episode success.

3.3 Explanation of Results

3.3.1 Episode Returns Plot

The Episode Returns histogram (Figure 3) visually summarizes the cumulative rewards across 5000 evaluation episodes. The x-axis represents the cumulative reward (episode return), while the y-axis indicates the number of episodes corresponding to each reward bin.

Key observations include:

- **Peak around return** ≈ 8: Most episodes yield cumulative returns around 8, reflecting quick and efficient achievement of target attractors with minimal interventions.
- **Secondary mode around return** ≈ 7: A secondary cluster indicates episodes requiring slightly more interventions or occasionally lower immediate rewards, yet still succeeding efficiently.
- **Long right tail**: Episodes occasionally achieved much higher returns (up to approximately 20), representing rare instances where the network dynamics required prolonged interactions before successful stabilization.
- **No returns below** ~ 6: All episodes surpassed a minimum return threshold, emphasizing that even the least optimal runs still eventually reached desirable states.

The shape of the distribution reflects the structured reward function, where quick convergence to target attractors yields consistent moderate rewards, with rare episodes requiring additional steps.

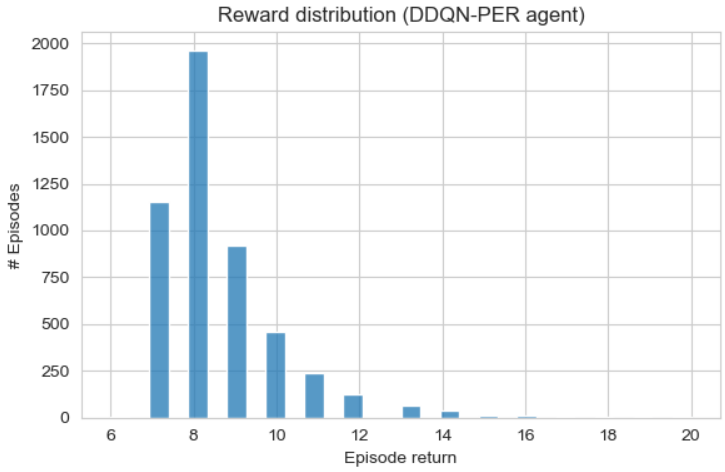

Figure 3: Episode Return Distribution (DDQN-PER Agent). Histogram showing cumulative rewards across 5000 evaluation episodes. Most episodes cluster around returns of approximately 8, indicating efficient stabilization with few interventions.

3.3.2 Steady-State Distribution Plot

The Steady-State Distribution (SSD) bar chart (Figure 4) contrasts the probabilities of network states under two distinct intervention policies: a baseline random policy and our trained DDQN-PER agent. The x-axis lists the top-20 states (by their probability under the random policy), while the y-axis indicates each state's empirical steady-state probability.

Several critical insights emerge:

- **Dominant zero-state (ID 0)**: State 0 (all genes OFF) is naturally prominent, indicating a large basin of attraction. The RL policy slightly reduces its dominance (∼29% random vs. ∼28% controlled), reflecting targeted shifts toward other desirable states.
- **Significant positive shifts**: Certain specific states dramatically increase their occupancy under the RL policy, highlighting targeted control actions successfully steering the network toward beneficial attractors.
- **Reductions in undesirable states**: The trained policy successfully suppresses occupancy in states known to be undesirable, redistributing probability mass to preferred configurations.
- **Broader probability distribution**: Compared to the random baseline, our DDQN-PER policy produces a more balanced distribution, reallocating probabilities from overly-dominant attractors to a more beneficial and distributed set of states.

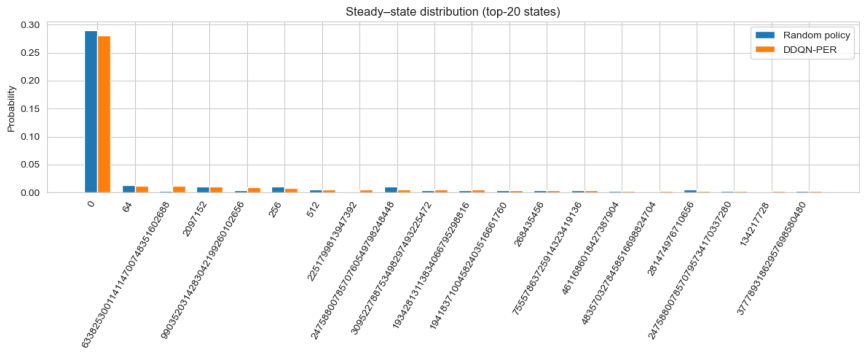

Figure 4: Top-20 Steady-State Distribution (SSD). Comparison of steady-state probabilities between a baseline random policy and our trained DDQN-PER agent. The RL policy shifts probabilities toward favorable states and reduces occupancy in undesirable attractors.

3.4 Potential Issues

Despite the strong overall performance, certain states retained higher probabilities under the random policy compared to the trained policy. This phenomenon highlights inherent limitations or inefficiencies within our current RL framework. Some states may intrinsically represent highly attractive basins of attraction or structurally challenging regions of the state space, requiring further model refinements or enhanced exploration strategies. Additionally,

our trained policy might benefit from longer training durations, more sophisticated network architectures, or improved exploration methods to achieve even finer-grained control over the state distributions.

Moreover, it is crucial to note that achieving success in stabilization does not necessarily imply optimal intervention efficiency. Although our RL agent consistently reached target attractors, it did not always minimize the number of interventions, suggesting that further optimization or a revised reward structure could yield even more resource-efficient control policies.

Finally, there exists an interpretability gap with the current reinforcement learning approach. While the trained policy demonstrates high performance in terms of network control, it remains challenging to biologically interpret why the agent selects certain gene flips. This lack of transparency complicates the application of RL-generated policies to biological contexts, underscoring the need for methods capable of providing clearer biological explanations alongside predictive accuracy.

4 Discussion

4.1 Limitations

While our implementation demonstrates promising scalability and control capabilities, several limitations remain evident. Firstly, although our approach is designed with linear complexity relative to the network size, the practical computational demands of training deep reinforcement learning models are still significant. High-dimensional state spaces such as ours (2^{100} possible states) can lead to computational burdens, especially in terms of memory usage and training time, which could hinder scaling to even larger networks or more complex biological scenarios.

Another limitation is the heuristic reward structure. Our reward function assigns fixed penalties for gene interventions, implicitly assuming equal costs across all gene flips. However, real biological interventions differ widely in their feasibility, practical costs, and potential side effects. Such an oversimplified reward design could introduce biases, driving the learned policy toward interventions that might be computationally favorable but biologically suboptimal or even infeasible.

Moreover, our current implementation is restricted to single-gene perturbations at each step. In realistic biological systems, especially in therapeutic contexts or genetic engineering applications, effective control may often require multi-gene or coordinated perturbations. The absence of such multi-step or combined interventions could limit the practical applicability of our model.

Additionally, despite their efficacy, deep Q-networks remain fundamentally "black-box" models. Our learned policies do not inherently provide clear explanations regarding why specific genes are flipped or whether these choices align with known biological mechanisms. This interpretability gap poses challenges for real-world deployment, particularly in fields where understanding the underlying biological rationale is crucial for adoption and trust.

Finally, while our agent consistently achieves the desired target states, success does not necessarily imply optimality. The agent often reaches targets quickly but does not always

minimize interventions, indicating potential inefficiencies or suboptimal decision-making strategies.

4.2 Future Work

To address the above limitations and extend the impact of our research, several promising future directions emerge.

First, incorporating biologically grounded reward shaping could significantly enhance the biological relevance and practical applicability of our reinforcement learning framework. Integrating domain knowledge—for instance, penalizing harmful or biologically unrealistic state transitions, or weighting interventions differently based on known biological importance or feasibility—would yield more realistic and actionable policies.

Second, hybrid neuro-symbolic approaches present an intriguing avenue for further exploration. Such methods could combine deep reinforcement learning policies with symbolic, graph-based regulatory logic or probabilistic model-checking techniques, enhancing interpretability and safety-critical validation. Neuro-symbolic models could provide clear justifications for gene intervention decisions, bridging the gap between computational performance and biological interpretability.

Third, extending our current approach to support multi-gene interventions and more biologically realistic perturbation constraints could enhance both performance and practical utility. Future research might explore action spaces that allow simultaneous perturbations of multiple genes or incorporate known constraints from genetic engineering practices.

Fourth, investigating alternative reinforcement learning algorithms, such as policy-gradient methods or actor-critic architectures, may yield additional improvements. These architectures might better handle continuous or more complex action spaces, potentially offering greater flexibility and efficiency in large-scale biological network control.

Lastly, exploring model-based reinforcement learning could leverage explicitly learned dynamics of the gene regulatory network, potentially improving sample efficiency and predictive capabilities. Incorporating explicit probabilistic models of gene interactions might allow for more precise control and better prediction of the biological impacts of interventions, enhancing real-world applicability and reducing risks.

By addressing these limitations and exploring these promising research directions, future work can further bridge the gap between computational methods and practical biological applications, advancing the field toward robust, interpretable, and biologically grounded control of gene regulatory networks.

5 Reflection

1. How do you feel your project ultimately turned out? How did you do relative to your base/target/stretch goals?

   I'm satisfied with how my project ultimately turned out. Relative to my base, target, and stretch goals, the model achieved a 100% success rate in stabilizing a 100-gene Probabilistic Boolean Network (PBN), which even exceeds my expectations. I believe I successfully implemented a good reinforcement learning pipeline from end to end.

2. Did your model work out the way you expected it to?

   Yes, the model performed as expected—and even surpassed my initial expectations. Achieving a perfect success rate in a high-dimensional control problem (N=100) is a strong indicator that the DDQN-PER approach was well-suited for this task. The model found efficient intervention strategies with small average interventions.

3. How did your approach change over time? What kind of pivots did you make, if any? Would you have done it differently if you could do your project over again?

   My approach remained consistent with the plan I outlined in Check-in 3. I followed a systematic roadmap for building the environment, designing the agent, and evaluating performance. The project structure, including the GitHub repository layout, is straightforward and comprehensive. If I were to do the project over again, I would likely follow a similar approach, though I might add additional unit tests or visual diagnostics earlier in development to catch subtle bugs in logic inference or reward design.

4. What do you think you could further improve on if you had more time?

   With more time, I would extend the experiments to other PBN sizes, particularly 70-gene and 150-gene networks, to benchmark scalability more rigorously. Additionally, I would experiment with longer training durations (e.g., 150,000 steps), larger batch sizes (e.g., 256), and alternative exploration schedules to see if these hyperparameters further improve convergence speed or reduce intervention count. Finally, I would explore integrating domain knowledge into reward shaping to align learned policies more closely with known biological constraints.

5. What are your biggest takeaways from this project/What did you learn?

   My biggest takeaway is a much deeper understanding of reinforcement learning—both conceptually and practically. I learned how to operationalize abstract concepts like Q-values, epsilon-greedy exploration, and prioritized replay into real-world code. It also reinforced the importance of modular, well-documented code for research projects and taught me how to apply RL in a real-world-inspired problem domain involving biological systems.

References

[1] S. Moschoyiannis, E. Chatzaroulas, V. Sliogeris, and Y. Wu, "Deep Reinforcement Learning for Stabilization of Large-Scale Probabilistic Boolean Networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2022. [Online]. Available: https://arxiv.org/abs/2210.12229.

[2] U. Sirin, F. Polat, and R. Alhajj, "Employing Batch Reinforcement Learning to Control Gene Regulation without Explicitly Constructing Gene Regulatory Networks," *Proceedings of the IJCAI*, 2013. [Online]. Available: https://www.ijcai.org/Proceedings/13/Papers/301.pdf.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236.

[4] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389.

[5] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," *arXiv preprint*, arXiv:1511.05952, 2015. [Online]. Available: https://arxiv.org/abs/1511.05952.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html.