Exploring Chaos in Neural Networks for Reinforcement Learning Efficiency

Yanbo Cheng
University of Toronto
yanboc.cheng@mail.utoronto.ca

Dennis Tat
University of Toronto
dennis.tat@mail.utoronto.ca

Abstract

Inspired by the chaotic firing of neurons in the brain, ChaosNet[1] is a novel artificial neural network architecture leveraging the chaotic dynamics of the 1D Generalized Luröth Series (GLS) map. Traditionally applied to classification tasks, ChaosNet is known for achieving competitive performance with minimal training data, owing to the topological transitivity property of its chaotic neurons. In this work, we extend ChaosNet to the reinforcement learning (RL) domain, proposing a novel algorithm that balances exploration and policy learning through its inherent chaotic dynamics. Remarkably, the adapted algorithm can achieve effective performance with as few as 10 training episodes, demonstrating significant sample efficiency compared to traditional RL methods. This study highlights the potential of chaos-driven architectures for efficient learning in dynamic decision-making tasks, paving the way for scalable RL frameworks with minimal data requirements.

1 Introduction

The rapid advancements in Artificial Intelligence (AI) through Machine Learning (ML) and Deep Learning (DL) have revolutionized fields like medical diagnosis, computer vision, and natural language processing. However, traditional artificial neural networks (ANNs), while inspired by biological neural networks, rely heavily on large datasets and optimization techniques to achieve high performance. This dependence on large training data limits their effectiveness in real-world scenarios where data may be scarce. To overcome this, researchers are exploring biologically-inspired algorithms that leverage chaotic dynamics, a hallmark of brain function. Chaotic systems, characterized by their adaptability and sensitivity to initial conditions, support efficient learning and rapid adaptation. One such model, ChaosNet, utilizes Generalized Luröth Series (GLS) neurons, which demonstrate the ability to perform classification tasks with lim-

ited data by leveraging chaotic properties such as topological transitivity.

In this work, we extend ChaosNet to reinforcement learning (RL), addressing the challenge of sample efficiency by exploiting the chaotic dynamics of GLS neurons. Our proposed RL algorithm seeks to reduce random exploration by leveraging the representational ability of the neurons, enabling effective predictions after as few as 10 training episodes. This makes it well-suited for data-limited domain. Experimental results on benchmarks such as CartPole, Black Jack, and LunarLander from OpenAI's Gymnasium library demonstrate the system's ability to learn rapidly and effectively. By integrating chaotic dynamics into RL, our approach offers a promising direction for developing adaptive and efficient AI systems for real-world applications.

2 Proposed Architecture

The architecture in this paper follows closely to the proposed architecture of the original ChaosNet, with the major modification being **TT-SS Prediction**

The proposed neuron is a Binary Tent map. This is a piecewise linear 1D chaotic map that is known as Generalized Luröth Series or GLS, often used for chaotic neuron models. It operates within the interval $[0, 1)$ and is defined by the following piecewise function:

$$T_{\text{Binary}} : [0,1) \to [0,1)$$

$$T_{\text{Binary}}(x) = \begin{cases} \frac{b}{x} & \text{for } 0 \leq x < b, \\ \frac{(x-b)}{(1-b)} & \text{for } b \leq x < 1 \end{cases}$$

1. Here, b is a parameter with $0 < b < 1$ that skews the map, and x is the initial value. The parameter b also acts as an internal discrimination threshold of the neuron which will be used for feature extraction.

2. A single layer of a finite number of GLS neurons satisfies a version of the *Universal Approximation Theorem [1]*

Single layer Topological Transitivity - Symbolic Sequence (TT-SS) based Reinforcement Learning Algorithm

The proposed architecture is as follows: It consists of a single input and single output layer. The input layer consists of n GLS neurons $C_1, C_2, ..., C_n$ that extracts patterns from each sample of the input data. The nodes $O_1, O_2, ..., O_s$ in the output layer stores the representation vectors corresponding to a certain bucket representing an action-reward pair. The input data is a vector of length n, representing the observations on every step of the RL environment.

The initial neural activity of each neuron is a hyper-parameter q, the input to the network is represented as $\{x_i\}_{i=1}^n$. The Chaotic firing of a GLS neuron C_i stops when its chaotic activity value $T_i(t)$ with initialization q reaches ϵ-neighhourhood of the input x_i (Stimulus).

Feature extraction: TT-SS based feature extraction is represented in the pseudo-code provided in figure 4 below. For each iteration of the GLS neuron C_k, if its value in $T_k^i(q)$ is not within the ϵ-neighborhood of the stimulus, it is added to a list; this list is denoted as the trajectory $A_k = [q, T_k(q), ..., T_k^i(q)]$. TT-SS feature is denoted by p_k.

$$p_k = \frac{h_k}{N_k}$$

Where h_k represents the duration of firing (number of iterations) where the chaotic trajectory is above the threshold b, and N_k is the total firing time for the k-th GLS neuron.

TT-SS Training: Given an input vector $X = [x_1, x_2, ..., x_n]$, use TT-SS feature extraction to retrieve each symbolic sequence feature $H = [h_1, ..., h_n]$. Depending on the action taken and reward, the feature is averaged with the bucket corresponding to the action-reward pair.

$$M^i = \frac{1}{m}[\sum_{i=1}^{m} h_1, \sum_{i=1}^{m} h_2, ..., \sum_{i=1}^{m} h_n]$$

M^i is a row vector and is termed as *average internal representation* corresponding to the action-reward bucket.

TT-SS Prediction: Given a normalized state of the current environment $Z = [z_1, z_2, ..., z_n]$, apply TT-SS based feature extraction to the data. Let the feature extracted data be denoted as $F = [f_1, f_2, ..., f_n]$. Next, F is used for cosine similarity computation with each of the average internal representation vectors $M_1, M_2, ..., M_s$ respectively:

$$\cos(\theta_1) = \frac{F \cdot M_1}{||F||_2||M_1||_2}$$
$$\cos(\theta_2) = \frac{F \cdot M_2}{||F||_2||M_2||_2}$$
$$\vdots$$
$$\cos(\theta_s) = \frac{F \cdot M_s}{||F||_2||M_s||_2}$$

Out of these s scalar values, pick the action label l as the one which maximizes the difference in cosine similarity between positive and negative rewards (recall that each bucket is an action-reward pair, an action k can have multiple reward

pairings ranging from positive to negative, each of these parings will incur a different average internal representation M):

$$\theta_l = \arg\max_{\theta_i} \big((\cos(\theta_{1p_1}) + \cdots + \cos(\theta_{1p_j}) - \cos(\theta_{1n_1}) - \cdots - \cos(\theta_{1n_k})),$$
$$\cdots,$$
$$(\cos(\theta_{mp_1}) + \cdots + \cos(\theta_{mp_a}) - \cos(\theta_{mn_1}) - \cdots - \cos(\theta_{mn_b}))\big)$$

Where each $\cos(\theta_{ip_k})$ represents the action i and positive reward and $\cos(\theta_{in_k})$ represents the action i with negative reward. If more than one label has the maximum cosine similarity, we take the smallest label.

3 Results and Discussion

The results of the proposed 1 layer RL algorithm is compared against the following methods: Q-Learning and DQN
The gym environements these models will train on are: CartPole, BlackJack, and LunarLander.

Environment	Initial Neural Activity (q)	Discrimination Threshold (b)	ϵ
CartPole	0.61	0.11	0.01
Blackjack	0.41	0.21	0.01
LunarLander	0.71	0.01	0.01

Table 1: Hyperparameter settings.

3.1 CartPole Results

Figure 1: CartPole environment visualization.

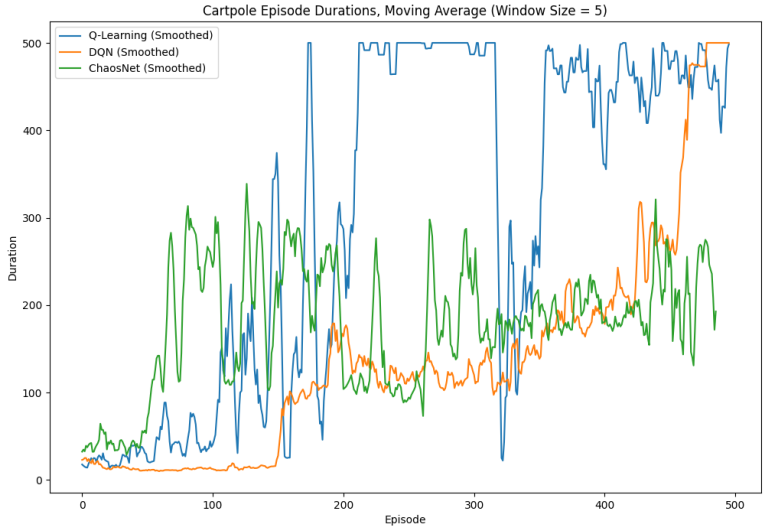

Figure 1 provides a visual representation comparing the performance of the three implementations: Q-learning, Deep Q-Networks (DQN), and ChaosNet. The results are smoothed using a moving average with a window size of 5 for better clarity.

ChaosNet does not perform as well as other models as the number of episodes increases. This is likely due to its small number of buckets (4, since the action-reward space is small), which limits its representational capacity. However, it shows surprisingly strong performance in the early stages. This suggests that ChaosNet is able to generalize and predict with much higher accuracy than the other two models within a small training sample. Its average episode duration(reward) is also notable, reaching approximately 170 timesteps per episode, demonstrating a good overall performance despite its simplicity.

The average computation time per episode further highlights the efficiency differences among the models. Q-learning achieves an average time of 0.058s per episode, making it the fastest. ChaosNet maintains an average episode time of 0.265s, indicating a balance between computational cost and performance. Finally, DQN, which takes the longest time at 0.414s per episode. This extended time reflects the higher computational demands of neural networks but also correlates with their ability to model more complex relationships.

3.2 Black Jack Results

Figure 2: Blackjack environment visualization.

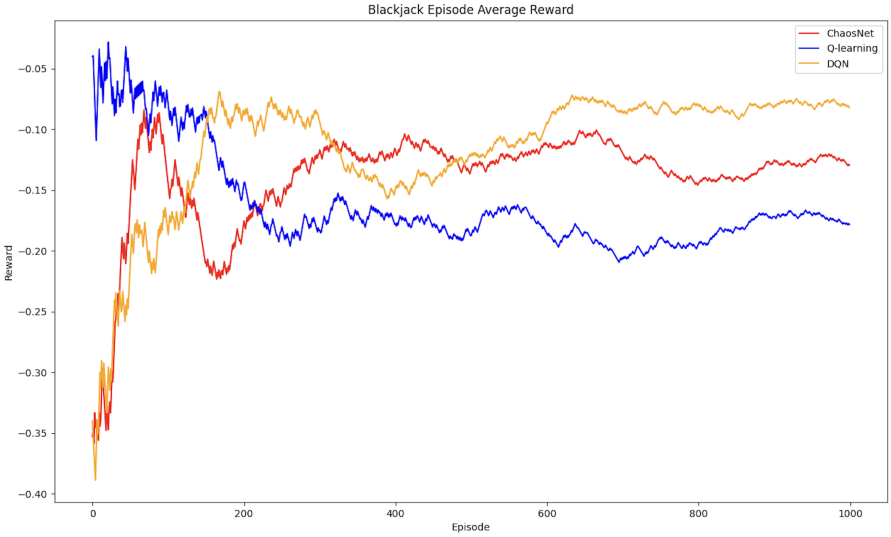

Figure 2 compares the performance between the three models in the Blackjack environment.

ChaosNet's performance surpasses that of Q-learning, but falls slightly short of DQN. Its average reward of -0.129 per episode is still notable, considering the inherent advantage of the dealer. This disadvantage, coupled with the stochastic nature of the game and its limited observation space, imposes an upper limit on performance of the models. As such, the performance of the three models do not fall far from each other.

Although training within the Blackjack environment is relatively fast, it is still valuable to compare the computational complexity of each model. ChaosNet achieves an average training time of 0.92ms per episode, outperforming DQN's 3.66ms per episode. However, Q-Learning remains the fastest, at 0.17ms seconds per episode. This puts ChaosNet as a good middle ground between the other two models, balancing both efficiency and accuracy.

3.3 LunarLander Results

Figure 3: LunarLander environment visualization.

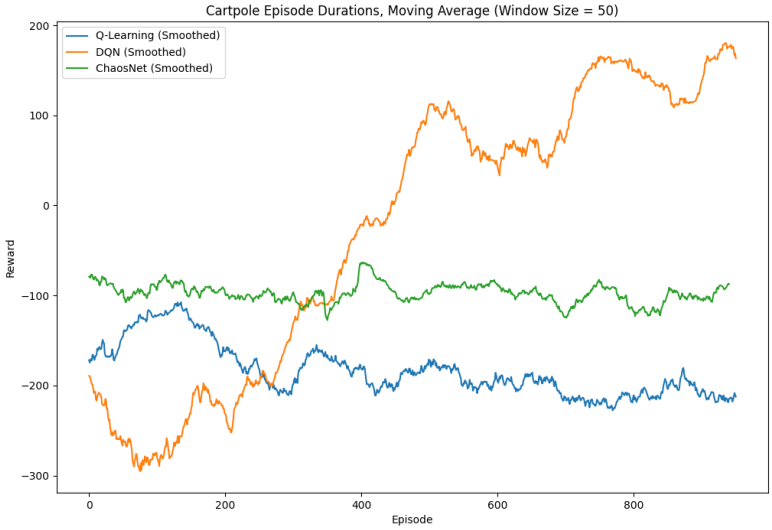

Figure 3 illustrates the performance of the three models in the LunarLander environment, smoothed using a moving average with a window size of 50 for improved clarity.

While ChaosNet demonstrates moderate performance increase against Q-Learning but, it is still comparable to a random agent, with an average rewards/episode of -97. DQN significantly outperformed ChaosNet, leveraging its architecture to capture the the complex continuous space.

Because of the continuous nature of the reward space, it posed a challenge to ChaosNet due to its discrete representation of buckets, making it difficult to capture a complete representation of the environment.

The times for each model are as follows: 0.035s per episode for Q-Learning, 0.41s per episode for ChaosNet, and 1.1s per episode for DQN. These results highlight the trade-off between computational efficiency and the ability to handle the environment's complexity.

3.4 Discussion

1. **Chaotic Behaviour:** Due to the inherent randomness in the initial training process, the model may sometimes learn a representation that poorly reflects the actual environment. This misrepresentation can cause the model to converge prematurely to a suboptimal solution, effectively get-

ting stuck in a local minimum. As a result, the model may struggle to generalize or adapt to the true underlying dynamics of the problem, leading to degraded performance. This issue can be addressed by increasing the number of random training episodes at the beginning.

2. **Hyperparameter Tuning:** Since it depends on two key hyperparameters, b and q, it would theoretically make sense to explore all possible values in the $[0, 1]^2$ space to find the optimal configuration. However, achieving high precision in such a search can be computationally expensive, making this approach impractical for extreme fine-grained tuning.

3. **Optimization:** This model cannot take advantage of the GPU's optimized matrix multiplications because its core operation involves finding the firing time for each neuron. Specifically, it requires determining when the skew tent function reaches an epsilon neighborhood of the stimulus. This process is linear in time with respect to finding the precise firing time, making it incompatible with the parallelized matrix operations typically used for efficient GPU computation.

4. **Multi-Layered ChaosNet:** While ChaosNet can be extended to include a second or even an arbitrary number of layers, the computation time would scale multiplicatively by $O(n)$ where n represents the time required to calculate the firing time for each additional layer. Moreover, each layer would require its own Discrimination Threshold b, significantly increasing the complexity of hyperparameter tuning as the number of layers grows. Given that a single layer already satisfies the Universal Approximation Theorem [1], allowing ChaosNet to approximate any continuous function with sufficient capacity, we restricted our implementation to a single-layer architecture. This approach ensures a balance between computational efficiency and representational power while avoiding the challenges of tuning multiple layers.

4 Conclusion

In conclusion, ChaosNet demonstrates notable potential in discrete action-reward environments, offering competitive performance in early stages of training and computational efficiency relative to traditional models like Q-learning and DQN. However, its current limitations in representing continuous spaces, as evidenced in the LunarLander environment, highlight the need for improvements in its architecture and parameter tuning. The discrete nature of its representation, defined by a small number of buckets, constrains its ability to fully capture complex environments. Despite these limitations, ChaosNet's strong early-stage performance suggests that it is particularly effective in scenarios with limited training data, making it a promising candidate for lightweight and fast decision-making systems.

Future research should focus on enhancing ChaosNet's representational capacity to handle continuous spaces more effectively. This could involve fine-tuning the model's hyperparameters, such as the discrimination threshold b and bucket quantization q, to better capture diverse environments. Additionally, exploring multi-layered ChaosNet architectures could significantly improve its ability to approximate more complex functions, potentially bridging the gap between its efficiency and the representational power of deep neural networks. These advancements could position ChaosNet as a versatile alternative for environments with varying complexity, balancing computational efficiency and performance across a wide range of applications.

The implementation for ChaosNet RL is available at `https://github.com/iamyanbo/Reinforcement-Learning-ChaosNet`.

5 References

References

[1] Harikrishnan Nellippallil Balakrishnan, Aditi Kathpalia, Snehanshu Saha, and Nithin Nagaraj. ChaosNet: A Chaos Based Artificial Neural Network Architecture for Classification. *arXiv preprint arXiv:1910.02423*, 2019.

[2] Raffaele Pumpo. (2023). CartPole-v1 Q-learning. *GitHub*. Retrieved from `https://github.com/RaffaelePumpo/CartPole-v1-Q-learning/blob/main/Cartpole.ipynb`

[3] Avgeridis, L. (2021). LunarLander-v2. *GitHub*. Retrieved from `https://github.com/lazavgeridis/LunarLander-v2`

[4] PyTorch. (2023). Reinforcement learning (Q-learning) tutorial. *PyTorch*. Retrieved from `https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html`

[5] Farama Foundation Gymnasium. (2023). Solving Blackjack with Q-Learning. *PyTorch*. Retrieved from `https://gymnasium.farama.org/tutorials/training_agents/blackjack_tutorial/`

[6] HK, Abhimanyu. (2024). Mastering Deep Q-Learning with Pytorch: A Comprehensive Guide. *PyTorch*. Retrieved from `https://medium.com/@hkabhi916/mastering-deep-q-learning-with-pytorch-a-comprehensive-guide-a7e690d644fc`

6 Appendix

Hyperparameters used to generate Q-Learning and DQN results are as follows:

1. For the CartPole case, no changes were made to the code from the GitHub repository, and the code was run entirely as-is.

2. For the Blackjack case, Q-Learning learning_rate was changed to 0.1, and n_episodes to 1000. DQN required changing environment from breakout to blackjack, and updating the dimension space to 3.

3. For the LunarLander case, the following command-line arguments were added: –n_episodes 1000 –lr 0.0001 –gamma 0.99

Environments used:

1. CartPole: CartPole is a classic control environment provided by Gymnasium, where the goal is to keep the pole balanced upright. Its action space comprises two values, {0, 1}. Observation space is a 4-tuple corresponding to: cart position, cart velocity, pole angle, and pole angular velocity. A reward of 1 is awarded for every non-terminating step, including the terminating step (For our algorithm, we will consider the terminating step as a reward of -1). The episode terminates if the pole angle exceeds $\pm 12^{\circ}$, cart position is greater than ± 2.4, or episode length is greater than 500.

2. Blackjack: Blackjack is a card game environment provided by Gymnasium, where the goal is to beat the dealer by getting closer to 21 without exceeding it. The action space is {0, 1}, corresponding to stick or hit. The observation space is a 3-tuple: the player's current sum, the dealer's face-up card value, and whether the player has a usable Ace. Rewards are +1 for a win, -1 for a loss, 0 for a draw. The episode ends if the player sticks or busts by exceeding 21.

3. LunarLander: LunarLander is a rocket trajectory optimization environment provided by Gymnasium. Its action space comprises four discrete actions: {0, 1, 2, 3}, corresponding to doing nothing, firing the left orientation engine, firing the main engine, or firing the right orientation engine. The observation space is an 8-dimensional vector representing the lander's position, velocity, angle, angular velocity, and leg contact with the ground. Rewards are given based on proximity to the landing pad, speed, tilt, and leg contact, with penalties for engine usage and -100/+100 for crashing or landing safely. The episode ends when the lander crashes, goes out of bounds, or becomes inactive.

Input: Stimulus (normalized) x to the GLS neuron (chaotic map $T(\cdot)$).
Output: TT-SS based feature p.

Initialize: GLS neuron is initialized with membrane potential q and internal discrimination threshold parameter b, ε-neighbourhood $I = (x - \varepsilon, x + \varepsilon)$ with $\varepsilon = 0.1$.

symbolicsequence $\leftarrow$ []
$w \leftarrow q$
$N \leftarrow 0$
$h \leftarrow 0$
while $w \neq T(w)$ **do**
    $w \leftarrow T(w)$
    $N \leftarrow N + 1$
    if $w < b$ **then**
        symbolicsequence $\leftarrow$ [**symbolicsequence**, L]
    else
        symbolicsequence $\leftarrow$ [**symbolicsequence**, R]
if $N == 0$ **then**
    $p \leftarrow 0$
for $i \leftarrow 1$ **to** N **do**
    if ***symbolicsequence***$[i] == R$ **then**
        $h \leftarrow h + 1$
$p \leftarrow \frac{h}{N}$ **return** p

Figure 4: TT-SS based feature extraction