Au Man Yi Sigmund | Leung Ching Fung

1 Introduction

This project investigates the application of reinforcement learning algorithms to train agents for playing tic-tac-toe. We focus on two prominent algorithms implemented in TF-Agents, a TensorFlow-based library: Deep Q-Network (DQN) and Proximal Policy Optimization (PPO). Our study evaluates the agents' performance in both the Super tic-tac-toe environment and the standard 3x3 tic-tac-toe board, comparing their learning efficiency and effectiveness against baseline opponents.

2 Experimental Setup

This section outlines the experimental framework for training and evaluating our DQN and PPO agents in tic-tac-toe. We define the game environments, including board configurations and dynamics, design a reward shaping strategy to optimize learning, and establish benchmarks and metrics to assess performance.

2.1 Game Environments

The board that we are given is in the shape of a cross, comprising of 5 squares with each square of 4x4 size. We can view it as a square of 12x12 size, with 4 corners missing 4x4 squares. The winning condition is updated to 4 in a row, or column, or 5 across the diagonal. We denote the above setting as Super tic-tac-toe. We also consider the conventional 3x3 tic-tac-toe board with a winning condition of 3 in a row, or column, or across the diagonal.
We also noticed that there is a significant difference in placement of the nought or cross in the given environment. The placement has $\frac{1}{2}$ chance of placing in the chosen square, with $\frac{1}{16}$ opportunity placing in the 8 random squares adjacent to the chosen square. It will be forfeited if the random choice is occupied, or it is out of the bound. We add a further assumption here that the agent is not allowed to choose any squares that are occupied nor out of the bound, in hope of the $\frac{1}{2}$ randomness in blocking opponents' move. We call this random placement for later settings.
We then consider the below 3 environments to evaluate whether our agents are learning:

1. Standard Tic-tac-toe environment (3x3 board with no random placement)
2. Random Tic-tac-toe environment (3x3 board with random placement)
3. Super Tic-Tac-Toe (12x12 board with 4 corners missing 4x4 squares with random placement)

We coded our environment to be adaptive to different sizing of boards and winning conditions. We also designed that our agent will be player 1 (X) for simplicity, with a randomness in the starting player. Slight modifications are made to each branch of reinforcement learning algorithms to facilitate the training process of the agent.

We consider tasks in which our agent interacts with the tic-tac-toe environment, in a sequence of actions, observations and rewards. At each time-step the agent selects an action a_t from the set of legal game action, the grids that the agent can place in our case, $A = \{0, \dots, K\}, K = \begin{cases} 79, & super\ tictactoe \\ 8, & else \end{cases}$. The action is then passed to the environment and modifies its internal state and game winning status. In general, the environment is stochastic. The environment's internal state is not observed by the agent, instead it observes 2 boards (board 1 and board 2 representing grids that player 1 and 2 occupies), which is an array. In addition, it receives a reward r_t representing the reward or penalty received by taking the action. It is worth noting that since the reward is only received at final state (winning/tie/losing), so additional rewards are shaped inside the environment to facilitate the learning process.

2.2 Reward Shaping

To optimize the learning process of the agent in Tic-Tac-Toe, we implement reward shaping derived from strategic patterns used in Gomoku, where aligning five pieces in a row, column, or diagonal secures a victory. These positional heuristics introduce intermediate rewards to mitigate the sparsity of terminal rewards (win, loss, or tie), guiding the agent toward strategically favorable board states.

The reward structure is based on "live" and "dead" configurations, characterized by sequences of the player's pieces with open or blocked ends:

- **Live 4**: A sequence of four pieces with both ends open, guaranteeing a win on the next move when the placement is not probabilistic (highest pattern reward)
- **Dead 4**: Four pieces with one open end, providing a single winning opportunity.
- **Live 3**: Three pieces with both ends open, representing a significant offensive threat.
- **Dead 3**: Three pieces with one open end, offering a limited extension opportunity.
- **Live 2**: Two pieces with both ends open, facilitating further pattern development.
- **Dead 2**: Two pieces with one open end (lowest pattern reward).

In a Super Tic-Tac-Toe environment, where four in a row or column or five in a diagonal constitutes a win, row or column live 3 and diagonal live 4 are prioritized. Conversely, in standard 3x3 Tic-Tac-Toe, where three in a line secures victory, dead 2 patterns are emphasized.

Terminal rewards are assigned for win, loss, or tie outcomes, while non-strategic moves yield no reward. To discourage the agent from favoring pattern formation over immediate victories, a step penalty is imposed with each move, deducting a small reward to incentivize rapid wins. Additionally, the agent's reward is reduced for failing to prevent the opponent from forming strategic patterns, ensuring a balance between offensive and defensive strategies. This reward framework fosters efficient gameplay, promotes proactive offensive moves, and reinforces robust defensive responses.

To evaluate our reward shaping strategy, we trained two PPO models in the Super Tic-Tac-Toe environment with identical hyperparameters, differing only in reward shaping. The reward-shaped model rewarded strategic patterns, while the baseline used standard rewards. Figure 1 below shows the reward-shaped model (orange line) achieved a 5% higher win rate (both >90%) with greater stability and required only 22 steps per episode compared to 33 for the baseline (blue line). This demonstrates that reward shaping enhances the agent's ability to learn winning strategies efficiently, improving both performance and convergence speed in complex game environments.

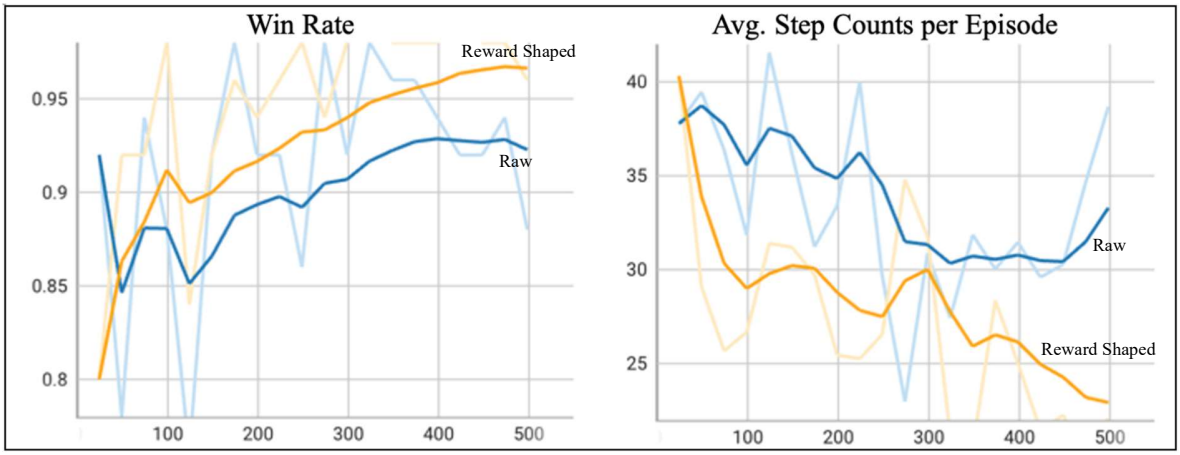

Figure 1: *Model performance with and without reward shaping*

3 Benchmark and Metrics

To evaluate the performance of our DQN and PPO agents, we use a random agent as the standard benchmark. Additionally, the PPO agent is evaluated against a self-play agent to further assess its performance. We emphasize four key metrics to monitor learning and performance, measured during training and evaluation phases. Additional metrics may be included for further analysis.

Training Metrics

- Training Loss: Measures the optimization error during training, indicating how well the model is learning.

Evaluation Metrics

- Win Rate: Percentage of episodes where the agent wins against the opponent agent.
- Average Reward per Episode: Average cumulative reward earned per episode against the opponent agent.
- Average Steps per Episode: Average number of steps taken to complete an episode against the opponent agent.

4 Deep Q-Network

Deep Q-Network (DQN) is an extension of Q-learning, which trains the agent to assign values to its possible actions based on its current state.

$$Q_\pi(s,a) \equiv \mathbb{E}[R_1 + \gamma R_2 + \cdots | S_0 = s, A_0 = a, \pi]$$

Where $\gamma \in [0,1]$ is the discount factor that trades off the importance of immediate and later reward. The optimal value is then $Q^*(s,a) = \max_\pi Q_\pi(s,a)$. The standard Q-learning update for the parameters after action A_t in state S_t and receive a reward R_{t+1} and the resulting state S_{t+1} is then

$$\theta_{t+1} = \theta_t + \alpha\left(Y_t^Q - Q(S_t, A_t; \theta_t)\right)\nabla_{\theta_t} Q(S_t, A_t; \theta_t)$$

With the target $Y_t^Q \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t)$ which resembles stochastic gradient descent on updating current Q value towards target Y_t^Q.

DQN does a similar job using a deep convolutional neural network, with layers of tiled convolutional filters to mimic the effects of receptive fields. We can also view that DQN algorithm is a function that maps the n-dimensional state space into the m actions, i.e. $\mathbb{R}^n \to \mathbb{R}^m$. The target used by DQN is

$$Y_t^{DQN} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-)$$

With $\theta_t^- = \theta_t$

Double Q-learning (DDQN) tries to overcome the issue of using the same values both to select and to evaluate an action as shown in standard Q-learning and DQN. The Double Q-learning error can be then written as below:

$$Y_t^{DoubleQ} \equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t{}')$$

The selection of the action, in the argmax, is still due to the online weights θ_t, as in Q-learning, we still estimate the value of the greedy policy according to the current values θ_t. However, a new set of weight θ_t' is introduced to evaluate the value for the policy. The second set of weights can be updated systematically by switching the roles of θ and θ'. This stabilizes the updates of the DDQN.

4.1 Algorithms

In our project we study the 4 Reinforcement Learning Algorithms from the DQN family:

1. DQN trained against random agent
2. DQN with Recurrent Neural Network, trained against random agent (DQRNN)
3. DDQN with Recurrent Neural Network, trained against random agent (DDQRNN)
4. DDQN with Recurrent Neural Network, trained in self-play method (DDQRNN self-play)

In our report, we adopted 2 kinds of Q networks to estimate the $Q^*(s, a)$.

4.2 Q-Network

Q-Network is a core component of a DQN agent, which its primary function is to approximate the optimal action-value function $Q^*(s, a)$. This function estimates the expected future discounted reward of taking an action a at state s and following optimal policy afterward.

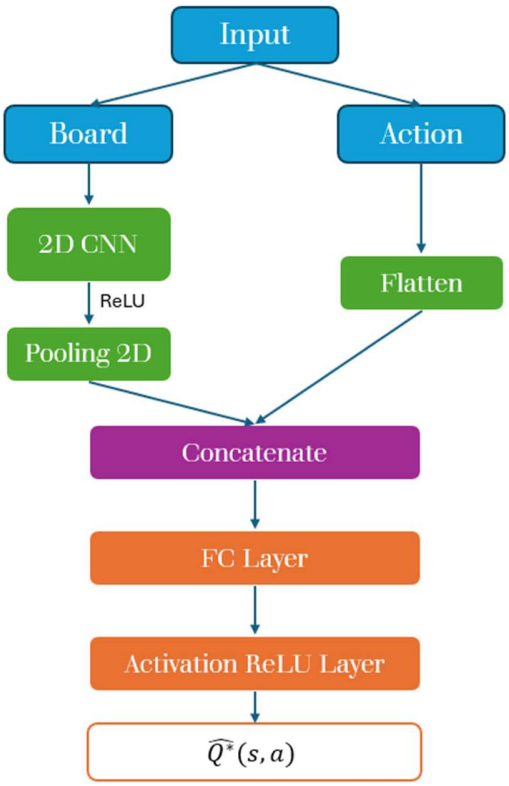

Figure 2: *Q Network Structure*

Our default setting for the fully connected layer (FC layer) is (128,64), and (16, (2,2),1) for Convolution Neural Network (CNN) 2D layer, in hope of capturing the possible states in the Super Tic-tac-toe environment.

4.3 Recurrent Neural Network

We also considered using the Recurrent Neural Network (RNN), which has the same preprocessing steps as our Q-network. The only difference is that there exists an additional layer of Long short-term memory (LSTM) layer, where the recurrent units in the RNN network holds a hidden state that maintains information about the previous inputs in the sequence, i.e. Recurrent units can "remember" information from prior steps, allowing the agent to learn the optimal Q^* faster, or get a better prediction on Q^* especially in the case of randomness.

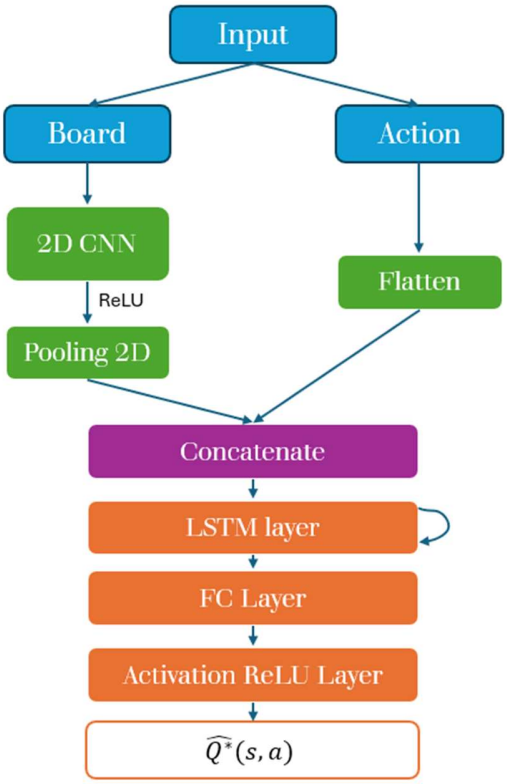

Figure 3: *Recurrent Neural Network Structure*

4.4 Self-play

Self-play is a commonly used technique for improving performance of reinforcement learning agents, not confined to only DQN family. The agents learn to improve performance by playing against the older version of itself. In some of our primitive trials, we found out that the self-learning trained agents have a weak performance as measured in performance (figure 3) V.S. random agent, i.e. overfitted. We acknowledge that the motivation of self-play is to train the agent with an opponent, which is on same level as the agent, and will gradually upgrade as agent upgrades. We therefore propose a hybrid approach to doing self-play as below with 2 hyperparameters p_{latest} and p_{random}, constrained on $\begin{cases} p_{latest} + p_{random} \leq 1 \\ p_{latest}, p_{random} \geq 0 \end{cases}$, and we choose our opponent based on the below schema:

$$opponent = \begin{cases} latest\ version, & p_{latest} \\ random\ agent, & p_{random} \\ older\ version, & 1 - p_{latest} - p_{random} \end{cases}$$

This method blends the perks of training against random agent, a level playing opponent and the possibility of overfitting into local optima.

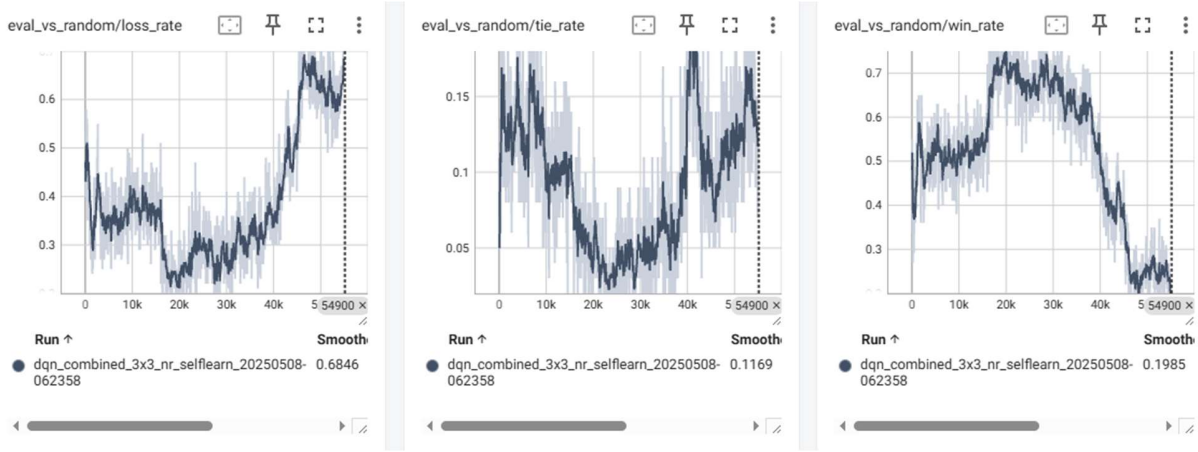

Figure 4: *DQN agent performance (V.S. Random agent) with standard self-play training*

4.5 Epsilon

For the epsilon selection, we considered a linear decay schedule which ends at 80% of the total training episodes. We set $\epsilon_{start} = 1$ and $\epsilon_{end} = 0.1$.

$$\epsilon = \begin{cases} \epsilon_{start} - \dfrac{\epsilon_{start} - \epsilon_{end}}{\dfrac{episode_{trained}}{0.8 \times episode_{total}}}, & \dfrac{episode_{trained}}{episode_{total}} \leq 0.8 \\ \epsilon_{end}, & else \end{cases}$$

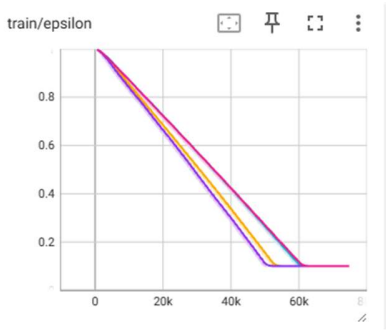

Figure 5: *Epsilon decay schedule visualized*

4.6 Results

For our standard tic-tac-toe environment (figure 5), we can see that our pink line (DDQRNN) outperforms in all metrics, but we noticed that the instability of the algorithm upon the last 2k iterations. We also noticed that the yellow line (DDQRNN self-play) exhibits a more stable learning curve, but we presume that more iterations are required to reach a perfect state. Another observation is that for all the algorithms, especially then DQN algorithm, there is an initial trough in reward driven by the active exploration. We generally see the algorithms learning to play fewer steps, or in other words, more direct at winning, thanks to our step penalty setting.

Whereas in the Random Tic-tac-toe environment (figure 6), we observed that due to the random nature of the environment, simple DQN algorithms does perform better than other algorithms in faster learning the pattern. We also observed that all agents are gradually learning how to win in the game, but not yet converged to the optimal solution due to limited training episodes. We also see that there are humps in the training loss function as compared to the smooth downward trend as shown in the standard tic-tac-toe environment, which is primarily driven by the stochastic nature of the environment after adding the random placement effect.

In our Super Tic-Tac-Toe environment (figure 7), we acknowledge that the random agent may not be a good competitor, yet a convergence in win rate to 100% for all models suggests that the agents are learning. We noticed that DQRNN and DDQRNN work best and learn to consistently win at around 3k episodes. The other agents are also converging to 100%-win rate against random. It is observed that the loss function is very volatile as the randomness in the environment is enormous for our exploration phase, thus the agent needs some time to overcome the issue. However, when the epsilon is lowered to ϵ_{end}, we can see a relatively stable result and loss function.

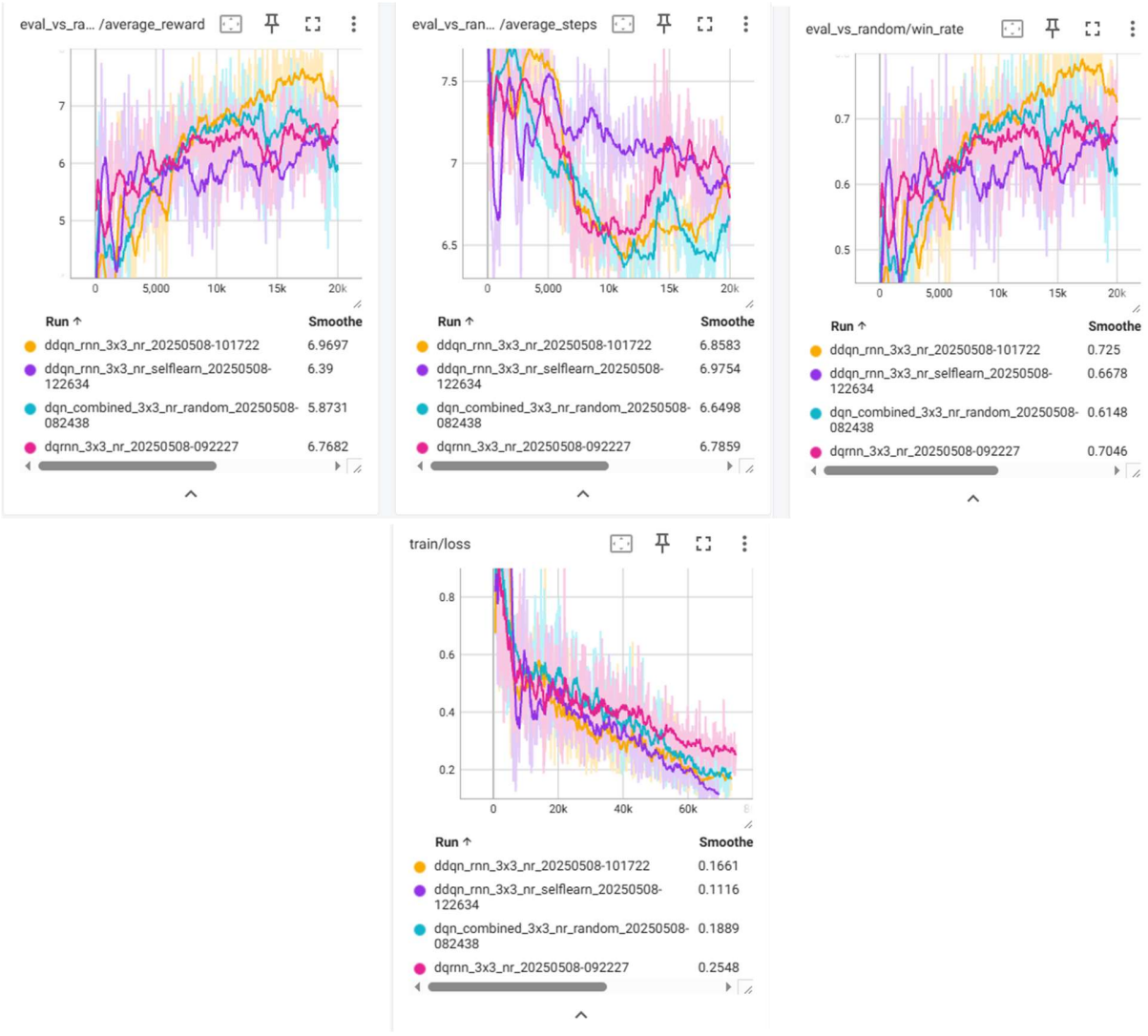

Figure 6: *DQN class Algorithms performance in Standard Tic-tac-toe Environment*

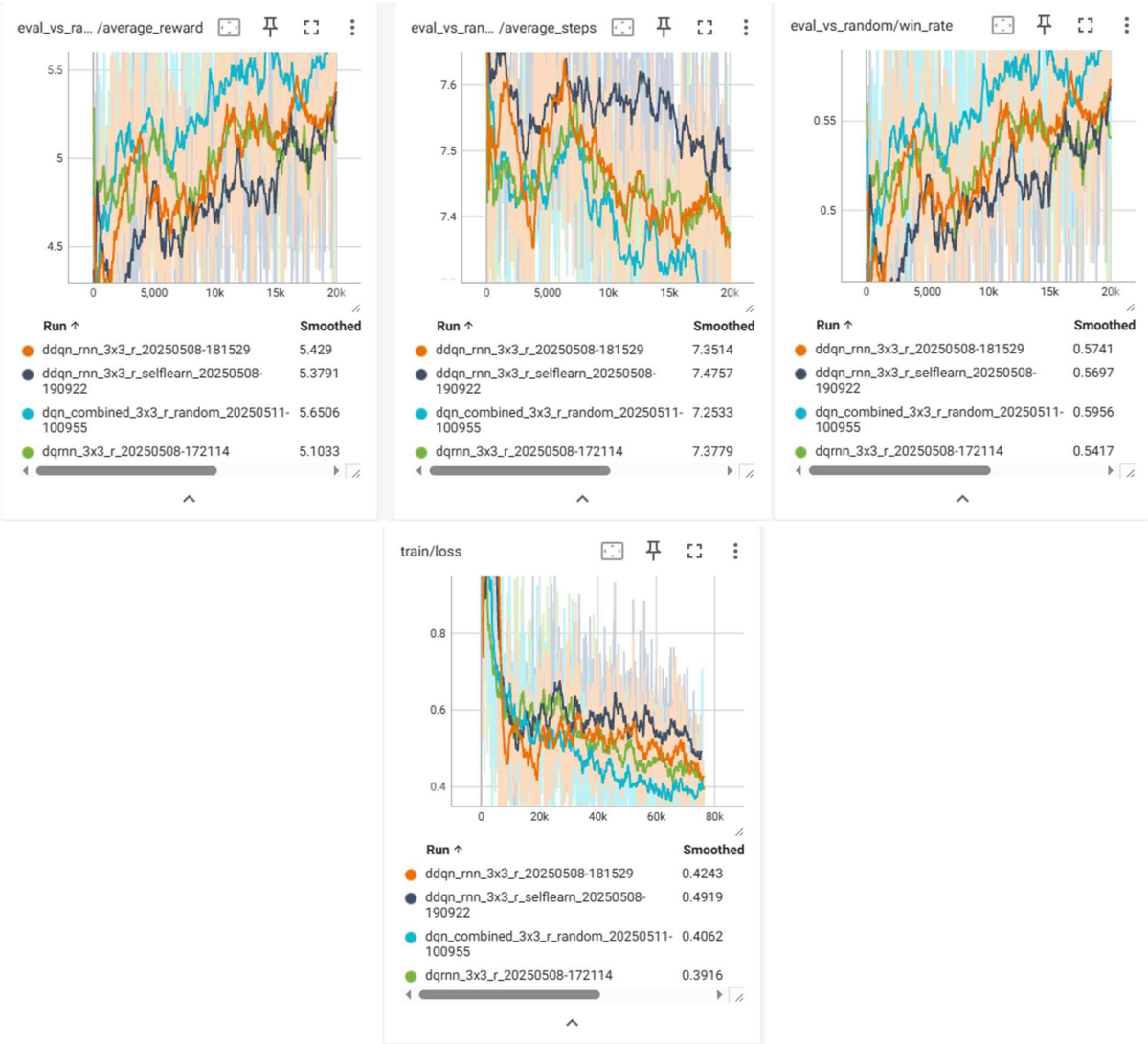

Figure 7: *DQN class Algorithms performance in Random Tic-tac-toe Environment*

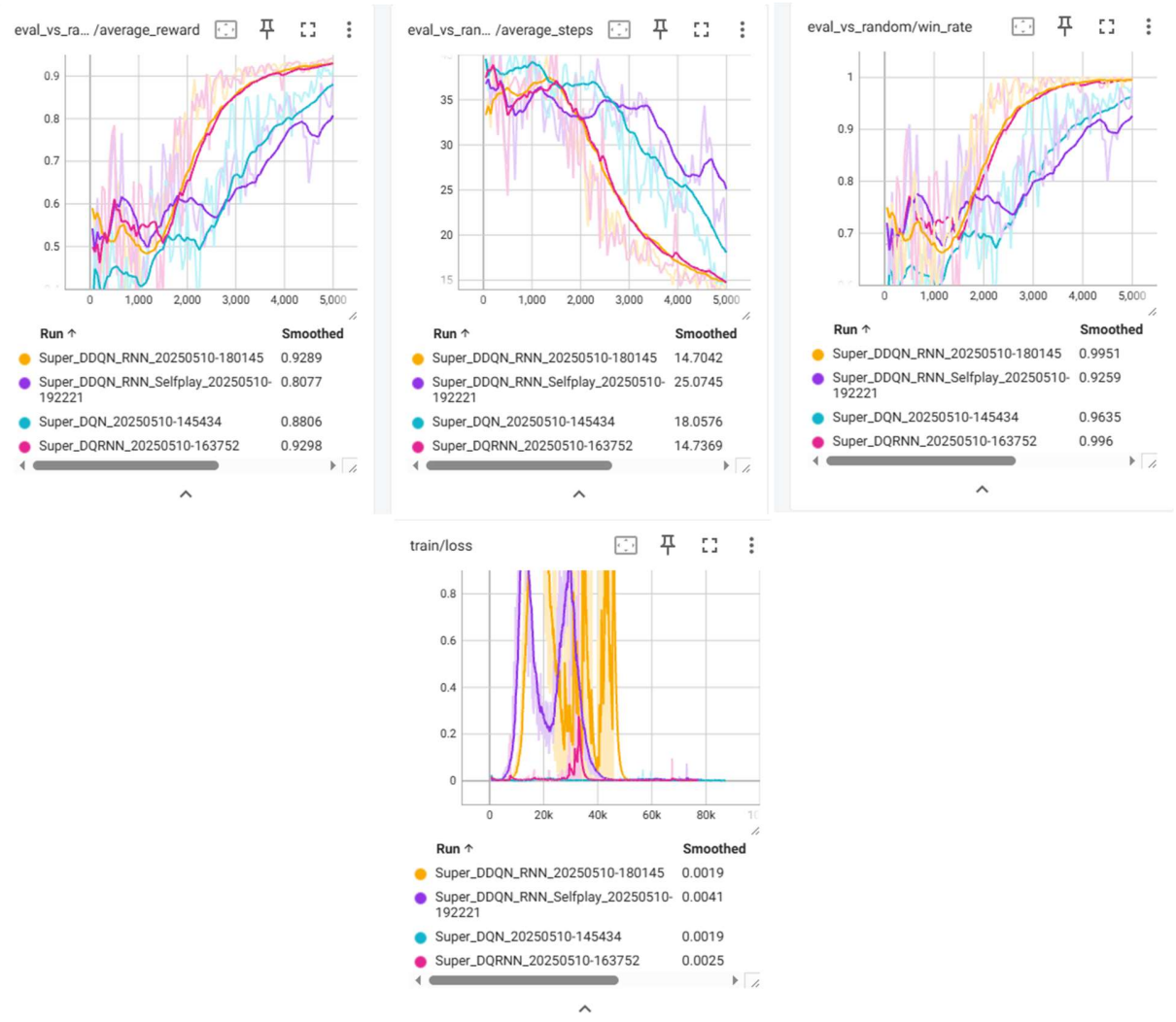

Figure 8: *DQN class Algorithms performance in Super Tic-tac-toe Environment*

5 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm designed to optimize a stochastic policy to maximize expected cumulative rewards while ensuring stable and efficient updates. PPO is an updated version of Trust Region Policy Optimization (TRPO), improving upon it by replacing TRPO's complex second-order optimization and trust region constraint with a simpler clipped surrogate objective, enhancing computational efficiency and practical applicability.

The primary goal is to maximize the clipped surrogate objective for the policy:

$$L^{CLIP}(\theta) = \mathrm{E}_t[\min(r_t(\theta)A_t,\ clip(r_t(\theta),\ 1-\varepsilon,\ 1+\varepsilon)A_t)],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|\mathrm{s_t})}$, is the probability ratio of the new policy to the old policy, A^t is the advantage estimate, and ε is a tuning parameter that limits policy updates to maintain stability. The advantage is computed using Generalized Advantage Estimation (GAE):

$$A_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \ \ldots\ + (\gamma\lambda)^{T-t+1}\delta_{T-1},$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$.

In scenarios where a neural network is employed to share parameters between the policy and value function, it is imperative to utilize a loss function that concurrently incorporates the policy surrogate and the value function error term. This objective function can be further refined by the inclusion of an entropy bonus, which serves to encourage sufficient exploration. By integrating these components, the new loss function will be:

$$L_t^{CLIP+VF+S}(\theta) = E_t[L_t^{CLIP}(\theta) - c_1L_t^{VF}(\theta) + c_2S[\pi_\theta](s_t)],$$

where c_1, c_2 are coefficients, and S denotes the entropy bonus, and L_t^{VF} is a squared-error loss $\left(V_\theta(s_t) - V_t^{targ}\right)^2$.

The PPO process then iterates as follows: (1) collect trajectories using the current policy, (2) compute returns and advantages, (3) minimize $L_t^{CLIP+VF+S}(\theta)$ to update both the policy parameters and value parameters, refining the policy to maximize rewards and the value function to improve predictions.

5.1 Actor Network and Value Network

In our PPO models, the actor network defines the policy $\pi_\theta(a_t|s_t)$, selecting actions by processing the input through a "Board" (2D CNN, ReLU, Flatten) and "Action Mask" (Dense, ReLU) stream, concatenating them, and passing through an FC layer to output a categorical distribution of action probabilities. The value network estimates the state value $V(s_t)$, aiding advantage computation, using a similar input flow—splitting into "Board" (2D CNN, ReLU, Flatten) and "Action Mask" (Dense, ReLU), concatenating, then applying an FC layer and a final Dense layer with one unit to predict the expected return.

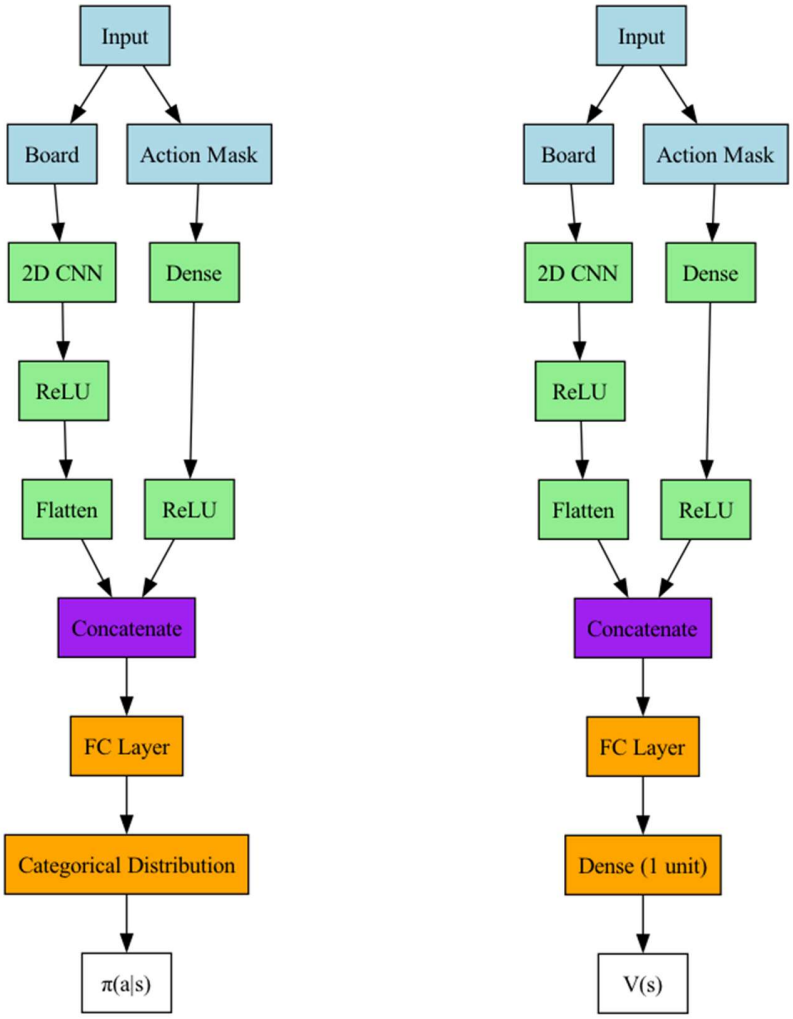

Figure 9: *Actor Network (left) and Value Network (right) of PPO algorithms*

5.2 Algorithms

Our PPO model employs a self-play training strategy to enhance its Tic-Tac-Toe performance, where the agent competes against a previous version of itself. Unlike random selection of prior policies, we strategically choose a policy believed to have mastered optimal play against a random opponent, ensuring the opponent provides a meaningful challenge. This approach fosters strategic evolution by exposing the agent to progressively stronger adversaries, promoting balanced skill development over mere exploitation of random policies.

Phase 1: Training Against Random Policy
In the first phase, the PPO agent is trained exclusively against a random opponent in the Tic-Tac-Toe environment. The objective is to develop an aggressive policy that efficiently forms winning lines, as random opponents lack strategic depth. Training continues until convergence, indicated by a consistently high win rate and stabilized loss values, ensuring the agent has learned an optimal aggressive strategy against random play.

Phase 2: Training Against Random and Phase 1 Policy
In the second phase, the agent trains against both the random policy and the converged aggressive policy from Phase 1, using a deterministic masked policy for self-play. This phase aims to refine the agent's strategy by balancing aggression with defensive tactics, as the Phase 1 policy challenges the agent to adapt to a strong, offensive opponent. The dual training ensures robustness across diverse scenarios.

Performance Comparison
To evaluate the self-play approach, we compare the Phase 2 agent's performance against a model trained solely against random opponents. Therefore, we will have 2 PPO algorithms:

i. PPO trained solely against random opponents, and
ii. PPO trained by self-play method

5.3 Results

The graphs below present the convergence outcomes of Phase 1 training against a random policy. Both win rate and average reward per episode exhibit a stable upward trend, indicating effective learning. The policy achieves win rates exceeding 90% in Standard Tic-Tac-Toe and 99% in Super Tic-Tac-Toe, with average step counts of approximately 6 and 15 episodes, respectively. This demonstrates the development of efficient, aggressive strategies that consistently outperform random opponents. In Random Tic-Tac-Toe, the win rate is notably lower at approximately 65%, likely due to the increased likelihood of forfeited moves in this environment. However, the stable average reward, consistent step counts, and converged training loss suggest robust policy optimization across all environments.

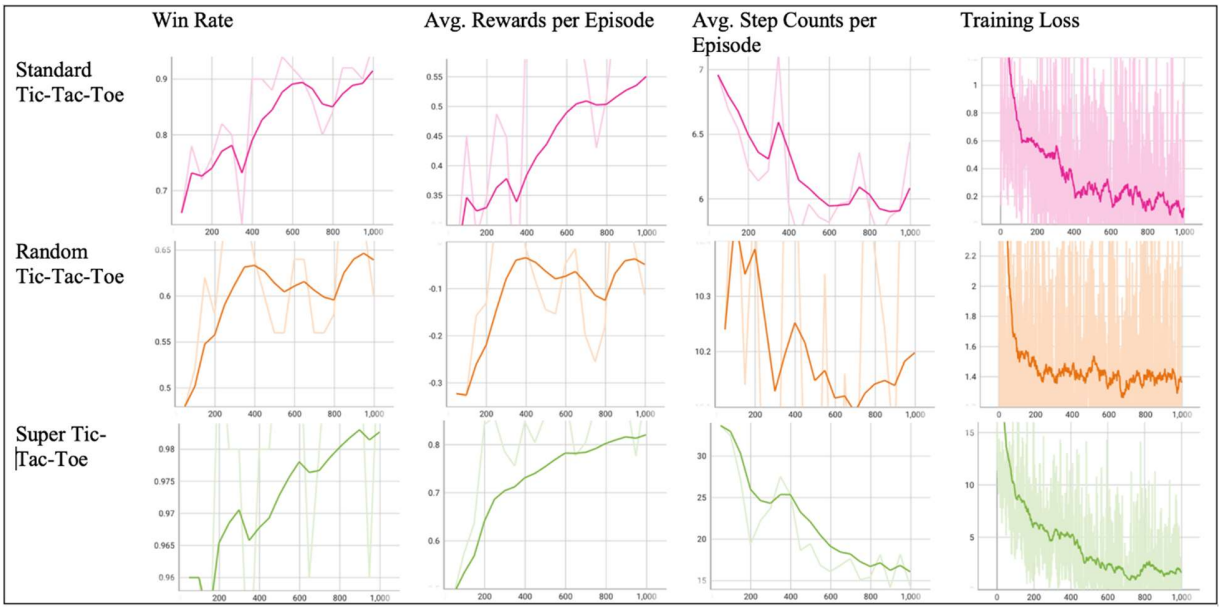

Figure 10: *PPO Agent Phase I performance*

The graphs below present the performance comparison between the self-play model (orange line) and the model trained solely against a random policy (blue line) in Phase 2, evaluated across Standard Tic-Tac-Toe, Random Tic-Tac-Toe, and Super Tic-Tac-Toe environments, respectively. The self-play model consistently surpasses the random-only model in most scenarios, particularly when competing against the Phase 1 aggressive policy, achieving significantly higher win rates and average rewards. Against a random policy in Phase 2, both models exhibit comparable win rates, with the self-play model slightly outperforming in Random Tic-Tac-Toe and Super Tic-Tac-Toe (by approximately 5% and 1%, respectively) but marginally underperforming in Standard Tic-Tac-Toe (by about 1%). However, when evaluated against the Phase 1 aggressive policy, the self-play model demonstrates substantial improvements, with win rates approximately 30%, 10%, and 20% higher in Standard Tic-Tac-Toe, Random Tic-Tac-Toe, and Super Tic-Tac-Toe, respectively, highlighting the efficacy of self-play in developing robust and adaptive strategies.

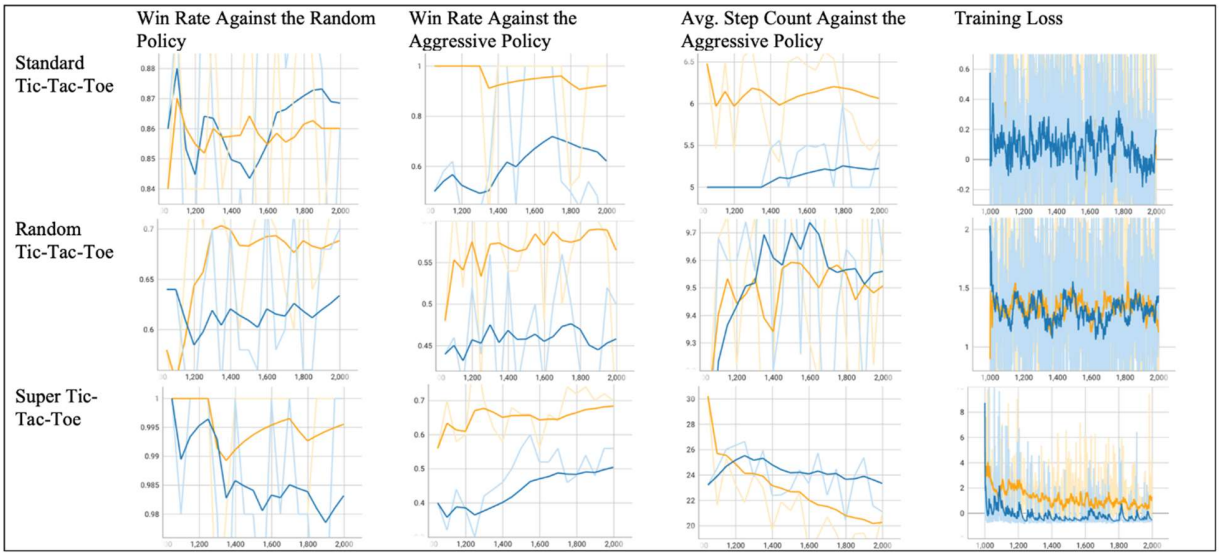

Figure 11: *PPO Agent Phase II performance*

6 Conclusion

In the DQN family, we realized that a significant improvement can be achieved with the introduction of Recurrent Neural Network to replace the standard Q-Network. Self-play method on the other hand, despite being widely regarded as an effective training methodology, did fall into a possibility of being overfitted to the opponent. Thus, we proposed a hybrid form of self-play which does a better job in learning. However, with a weak benchmark such as random agent, we do see a simple model (e.g. DQRNN) is sufficient for the job as it requires lower episodes of training for the agent to quickly grasp the winning formula. A tradeoff is observed in model complexity (finer performance) and learning speed (episodes to reach equilibrium).

Our PPO model demonstrates robust performance in self-play and rapid convergence when trained against both random and previous policy opponents. Unlike DQN, PPO reduces the risk of overfitting in self-play, likely due to its clipped objective function, which ensures stable policy updates, and its policy gradient approach, which adapts effectively to diverse opponents. The varied training experience, combined with our reward shaping strategy, accelerates convergence compared to DQN's value-based learning. These results suggest PPO's suitability for complex, stochastic environments like Random and Super Tic-Tac-Toe, where adaptability and stability are critical.

To enhance our models, we propose training agents against opponents from different algorithms (e.g., PPO vs. DQN) to improve robustness. Self-play can be refined by selectively sampling competitive previous policies based on performance metrics (e.g., win rate). Additionally, implementing curriculum learning to gradually increase opponent difficulty and tuning hyperparameters could further optimize convergence and performance across all tic-tac-toe environments.

7 Reference

1. Hasselt, H.V., Guez, A., Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. arXiv preprint arXiv:1509.06461.
2. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347.
3. Moreno-Vera, F. (2019). Performing Deep Recurrent Double Q-Learning for Atari Games. arXiv preprint arXiv:1908.06040.

8 Task Attribution

Proximal Policy Optimization: Leung Ching Fung (21133252)

DQN Policy family: Au Man Yi Sigmund (20504636)