ISNNI: 3007-3391 HEC Category: Y

Journal of Computers and Intelligent Systems
Vol. 4, No. 1, 2026, pages 31 - 44
Journal Homepage:
https://journals.iub.edu.pk/index.php/JCIS/

Deep Reinforcement Learning in Othello: A Deep Q-Network Approach

Areej Fatemah Meghji[1], Dua Agha[2], Aleena Rafique[3]
[1]Department of Software Engineering, Mehran University of Engineering and Technology, Jamshoro, Pakistan
[2]Department of Computer Science and Information Technology, NED University of Engineering and Technology, Karachi, Pakistan
[3]Department of Software Engineering, Hyderabad Institute for Technology & Management Sciences, Hyderabad, Pakistan

ARTICLE INFO

Article History:

Received: March 01, 2026
Revised: March 10, 2026
Accepted: March 15, 2026
Available Online: March 30, 2026

Keywords:

Reinforcement Learning
Deep Q-Network (DQN)
Othello AI
Self-Play
Epsilon-Greedy
Q-Learning
Game AI
Strategic Learning

Funding:

This research received no specific grant from any funding agency in the public or not-for-profit sector.

ABSTRACT

This paper presents the design, implementation, and analysis of a Deep Reinforcement Learning agent to play the strategic board game Othello with a Deep Q-Network (DQN) architecture trained by playing with a random player. Though the state space of Othello is highly complex (~10^{28}), it has very simple rules, so it is an ideal testbed for exploring pure learning properties of DRL algorithms. Modern studies pay more attention to emerging architectures such as Transformers and Graph Neural Networks to simulate Othello, leaving a thorough baseline with basic DRL algorithms yet to be found in the literature. Our approach to reduce this gap is to make a full Othello environment and then to train a DQN agent with tabula rasa input (with only the raw board state as input) and without any embedded tree search or human play data. The agent employed in this research uses the epsilon-greedy exploration and linear decay strategy and learns through Q-value approximation through a custom neural network in NumPy. With 300,000 training games in various epoch configurations, the agent wins 68.1% of the time against a random player indicating a good deal of strategic learning. Observation demonstrates that there are emergent gameplay patterns that can be interpreted as Othello heuristics: corner control and mobility optimization, although these concepts are not explicitly programmed. The agent also shows a good performance in the game as compared to human players, placing it at a hard level of difficulty. This research presents a reproducible DQN baseline implementation of Othello, which may be used as a comparison with more complicated algorithms. The implementation, outcomes, and training procedures provided by us have provided grounds for future studies on the value-based reinforcement learning of board games.

Corresponding Author's Email: areej.fatemah@faculty.muet.edu.pk

1. Introduction

The creation of artificial intelligence (AI) capable of playing complex games has long been the center of attention for researchers with advanced algorithms being explored and implemented to achieve this goal. The recent achievements of agents are a turning point to general learning algorithms such as AlphaZero that achieved superhuman performance in Chess, Shogi and Go using self-play reinforcement learning [1], and then the successor MuZero learned these games with no prior knowledge of how to play [2]. This advancement is mainly driven by Deep Reinforcement Learning (DRL), where agents learn the best plays in a game through interacting with the environment and collecting rewards, without the need for pre-labelled data [3]. Such breakthroughs and advancements are considered steppingstone towards creating AI systems that would work within the complex and strategic contexts of robotics or customized systems [4].

Othello (Reversi) is a perfect test case in this domain, which has not been fully explored. It has a mixture of strategy and state space complexity, with the statespace of Othello been fairly large with approximately 10^{28} gaming positions [5]. A feature that makes Othello attractive from a research point of view is are the considerably simpler rules which may be

used to encode the game especially when compared to chess. This feature qualifies Othello as an ideal domain for isolating and analyzing the pure learning capabilities of DRL algorithms with no distraction of complex rule implementation.

The research and development of Othello AI has been imbalanced; although substantial research exists on traditional search-based methods such as Monte Carlo Tree Search (MCTS) [6], and recent studies exploring advanced neural architecture and policy-based techniques [7], [8], there is still an absence of thorough empirical studies that are used to set a strong performance baseline for the foundational DRL algorithm: specifically using the Deep Q-Network (DQN) [9]. The learning process and capabilities of a fine-tuned DQN agent are not well represented in the contemporary literature that is learned via self-play in Othello. This is crucial because to evaluate more sophisticated algorithms that follow, it is necessary to have a baseline DQN agent.

The current study aims to carry out an investigation into the application of a DQN agent in Othello. The research attempts to provide a documented baseline for value-based DRL in this strategic game. The goal of this study is to design, implement, thoroughly assess, and evaluate DQN-based agents that can learn Othello strategy effectively through self-play. To accomplish this goal, the current research uses only the raw board state as input and implements a DQN agent that learns tabula rasa. We train the agent against a random opponent using a self-play regimen (scalable), along with documenting its learning progress and its final performance. The research also analyzes the strategic competence of the agent by analyzing the emergent behavior in the absence of integrated tree search or complex neural structures and provides a baseline to facilitate future DRL research in Othello and similar domains.

Our work employs the foundational DQN algorithm, in contrast with the recent work that focuses on complex architectures such as Graph Neural Networks [8], Transformers [9], or approaches that focus on hyperparameter optimization [10] or policy-based algorithms like PPO [11]. Our research does not aim to surpass the current state-of-the-art systems, rather it aims to offer a critical and key baseline. Our study provides an assessment of the capabilities of DQN, as well as a baseline to which the complexity and efficiency of new algorithms can be compared. This emphasis on establishing a transparent baseline positions our research as a reference point for future comparison.

The paper is structured as follows: The background of DQN and related work is provided in Section 2. The tools and technologies that are employed in the development and training of the Othello environment and the DQN agent are discussed in Section 3. Section 4 and Section 5 provide methodology and implementation. Section 6 presents our experimental results and analysis. Finally, the conclusion and future work directions are discussed in Section 7.

2. Related Work

In [12], the authors explore learning inspired by classic Othello and Tri-Othello, a recently proposed three-player board game using reinforcement learning. The rules were modified to ensure ease of learning by human players without compromising strategic complexity and equity. A reinforcement learning agent was found to be more flexible and able to make decisions as compared to a classical max-n search algorithm [12]. In another research. to compare the most recent algorithms of self-play like AlphaZero, MuZero, Gumbel AlphaZero, and Gumbel MuZero, a zero-knowledge reinforcement learning algorithm was developed to compare AlphaZero and Gumbel AlphaZero in Othello Go and in Atari games MiniZero [13]. The results indicated that progressive simulation strategies enhance training efficiency, and the more simulations are increased, the better the performance achieved. The paper highlights the significance of self-learning and discovery and confirms Othello as a suitable standard on which to compare reinforcement learning algorithms. To overcome the problem of the sample inefficiency in standard methods, Othello was exposed to hierarchical reinforcement learning (HRL).

Deep Q-Learning HRL was more efficient in terms of sample usage compared to Deep Q-Learning HRL, which simplified complex decision-making and smaller subproblems with fewer training games to reach a competitive level of performance. This method brings about the significance of organized learning in large state-space games [9]. On the strength of comparing reinforcement learning agents and Transformer-based models in Othello, complementary strengths were discovered. Although Transformers was able to reproduce board patterns and strategic representations around the world, RL was able to showcase a high level of adaptive learning due to self-play. The researchers emphasize the relevance of RLs in those scenarios when continuous self-improvement is to be promoted and significant labeled data is not available. Monte Carlo Tree Search (MCTS) is a sequential decision-making method that was explored vis-A-vis Othello. MCTS facilitated good learning among RL agents, whereby exploration evaluates the potential moves and

balances exploitation and exploration. The paradigm provides a basic strategy of organized self-play and e-greedy strategy [14].

Further studies on HRL on Othello demonstrated that decision-making can be divided into hierarchical subtasks, which enhance the learning efficiency and lead to more consistent play in strategic games. This technique promotes the development of agents who are coaching- and self-focused [15]. Effective methods of reinforcement learning of Reversi were focused on the computational power of game tree search. It was possible to compute faster and do competitive gameplay as well as Monte Carlo Tree Search optimizations, such as multithreading [16]. The work presented in [16] also highlights the significance of exploration efficiency to the management of large state spaces.

To measure the performance of different architectures, including fully connected and convolutional networks, on the training efficiency in learning and resource use of deep reinforcement learning in Neural Architecture Search (NAS), the application to Othello was used to measure and compare the performance of different architectures. Architectural choices that emphasized the importance of model design regarding scalable and flexible Othello agents had a tremendous effect on agent efficacy. In [17], MCTS was applied as a tester in an AlphaZero-inspired reinforcement framework, and temporal-difference learning with n-tuple networks was applied in training. This reduced processing costs but did not compromise efficiency. The given strategy showed that the use of self-learning and useful RL agents was applied successfully when the agent was able to beat the strong Othello program Edax until level 7 using regular hardware [18].

Unlike prior studies emphasizing Transformers [7], [9], Graph Neural Networks [8], policy-gradient methods [11], [13], or optimization [10], the current study focuses specifically on a standard value-based DQN without embedded search or expert demonstrations. The objective is not state-of-the-art performance but a transparent baseline against which future improvements can be evaluated. While current literature applies reinforcement learning to Othello primarily to compare architecture or reformulated rules, comprehensive research on a dedicated Deep Q-Network (DQN)-based analysis remains limited. Furthermore, there is a lack of research examining learning progression and decision-making behavior of DQN agents in this domain. This paper contributes by systematically studying DQN performance and learning dynamics in a standardized Othello environment.

3. Tools and Technologies

This section illustrates the software, tools and technologies that were employed in the development and training of the Othello environment and the DQN agent.

3.1 Development Tool

3.1.1. PyCharm IDE

This research used PyCharm, which is an Integrated Development Environment (IDE) specifically designed for Python [19]. Its feature set was instrumental in providing quality and manageability of code throughout the project lifecycle. The key features utilized were:

- Smart Code Assistance: It offers autocompletion and quick-fix offerings that ensure the development process is quicker and fewer bugs are made.
- Integrated Debugger and Terminal: Provided the ability to debug the game logic and agent training process step by step in the development environment.
- Version Control Integration: Smooth integration with Git enabled easy tracking of modifications and testing of various model versions.
- Scientific Tools Support: The data analysis and visualization were performed with the help of scientific libraries (NumPy, Matplotlib) and tools (Jupyter notebook).

3.2 Technologies and Libraries

During the development, different core libraries of Python were used for a particular task on the system architecture.

3.2.1 NumPy

All numerical calculations on the game state were done with NumPy [20]. We represented the Othello board as an 8x8 NumPy array, allowing the numbers to be manipulated, moves checked and the discs turned using the vectorized functions, crucial for the speed in training.

3.2.2 Colorama

The Colorama library was used to add some color to the command line Othello game [21]. This allowed for a better user interface for the human vs. agent games, as black and white discs were easily distinguished

3.2.3 Matplotlib

The graphs and visualizations have been generated using Matplotlib [22], such as the learning curves showing the win rate of the agent depending on the number of training epochs. This was used to analyze and prove the learning of the agent

3.2.4 Pickle

Object serialization was done using the pickle module of Python [23]. It was mainly used to save and retrieve the weight matrices and bias vectors of the neural network, which enables the persistence of the trained model across sessions and facilitates the continuation of interruption in training runs.

3.2.5 Neural Network Implementation

The neural network was implemented in NumPy from scratch through a custom neural network class. The reason behind this choice was to ensure complete control and visibility into the forward and backward propagation steps used in the DQN. The class provides methods for forward propagation, backpropagation for weight updates, and for saving/loading the model to a file using pickle

3.2.6 Pandas

Pandas was used to log and analyze training statistics. Data Frame was used in storing the win/loss records of each training session, and then the files were exported to .csv for persistent storage and subsequent analysis [24].

4. Research Methodology

This section outlines the approach that was undertaken to achieve the research objectives. The methodology was conducted through four different steps, which include the domain of the problem, the identification and application of the core algorithms, and lastly, the reinforcement learning agent (RLA) training and testing. Figure 1 is a representation of the workflow.

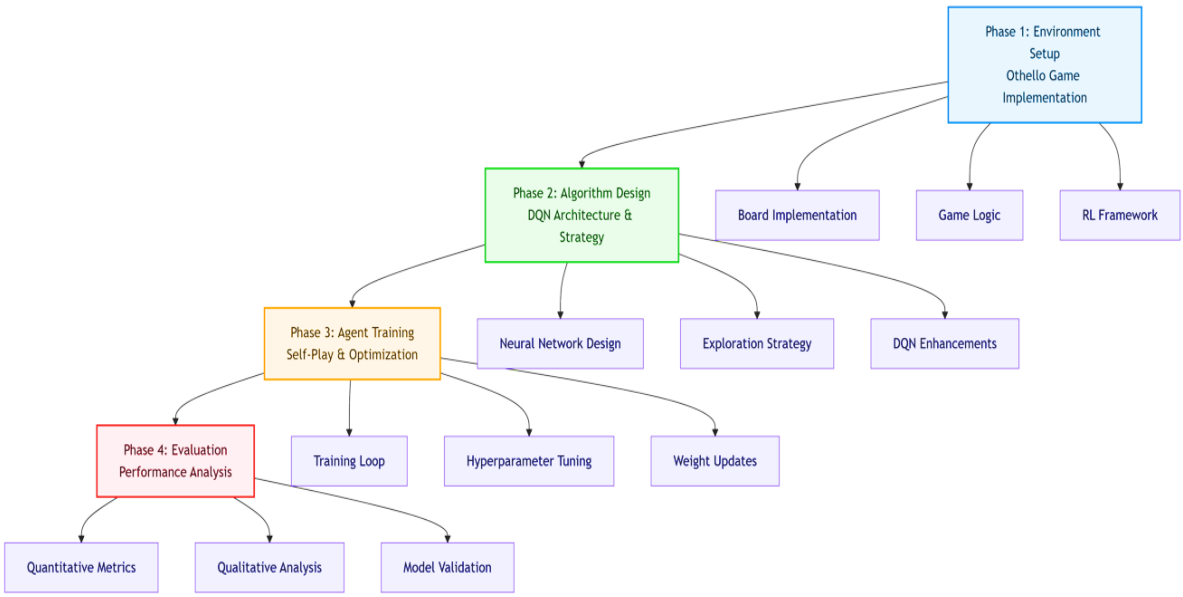

Figure. 1. Research Methodology Workflow.

4.1 Phase 1: Implementation of the Othello Game Environment

The implementation of the Othello game environment was carried out in the first phase. The initial stage consisted of a deep knowledge of the Othello game and the creation of the environment in which the two players could interact effectively. This was the testbed of the RLA.

- Learning the Game Dynamics: The rules of Othello were learned, including the logic of disc flipping and game validation, end-game recognition, and scoring.
- Environment Development: The two-player Othello game was written in Python language. The board was an 8x8 NumPy array.

The key functionalities were:

- isValidMove(): A function that verifies the legality of moves in all eight directions and returns a list of discs to be flipped.
- updateBoard(): A function used to place a disc and turn over the discs of the opponent based on the output of isValidMove().
- The Colorama library used to create a command-line interface (CLI) to display the state of the board to the human-agent interface.

4.2 Phase 2: Algorithmic Research and Selection

This phase involved researching and selecting suitable reinforcement learning and exploration algorithms that aligned with the objectives of the project.

4.2.1 Overview of Considered Algorithms

- Markov Decision Process (MDP): The theoretical framework for reinforcement learning that is used to formalize the Othello game into states, actions, transitions, and rewards.
- SARSA (State-Action-Reward-State-Action): An on-policy reinforcement learning algorithm where the agent learns the Q-value based on the action taken by its current policy [25].
- Q-Learning: An off-policy reinforcement learning algorithm (model-free) that learns the value of actions in specific states. Its update rule is given by.

$$\boldsymbol{Q}\,(\boldsymbol{s},\boldsymbol{a}) \leftarrow \boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a}) + \ \boldsymbol{\alpha}\,[\boldsymbol{R} + \ \boldsymbol{\gamma}\ \boldsymbol{max}\ \boldsymbol{Q}(\boldsymbol{s}',\boldsymbol{a}') - \boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a})]$$

where α is the learning rate, γ is the discount factor, *R* is the reward, *s* is the state, and *a* is the action [26].

4.2.2 Selected Algorithm: Deep Q-Learning (DQN)

Q-learning was chosen due to its simplicity, proven effectiveness in discrete action spaces, and off-policy nature, which means that it allows learning from experiences generated by any policy (like a random opponent), but a tabular approach to Q-learning was not feasible for such a large state space of Othello (6464 squares with 33 states each: empty, black, and white). Therefore, Deep Q-Learning (DQN) was employed, which applies a neural network (a Deep Q-Network) to estimate the Q-value function Q (s, a). The DQN takes the states (the flattened board) as input and outputs a vector of Q-values for all possible actions.

Key DQN Enhancements Implemented:

- Experience Replay: The agent experiences (s, a, r, s′) were stored in a replay memory. This memory was randomly sampled via mini batches during training to disassociate sequential experiences and stabilize learning.
- Target Network: The target Q-values R + γ max a′ Q (s′, a′) were computed on a separate target network that was the same as the main Q-network. The weights of this target network were periodically updated using the weights of the main network while preventing the feedback loop of pursuing a moving target and further stabilizing training.

4.2.3 Exploration Strategy: Epsilon-Greedy

The Epsilon Greedy strategy was adopted to balance exploration and exploitation [27]. The agent chooses a random action according to probability ϵ (exploration) and the action that has the highest Q-value with probability 1−ϵ (exploitation). There was an epsilon decay schedule implemented, where ϵ starts at a high value (1.0) and gradually decays to a small value (0.01) over the course of training. This allows the agent to explore and search extensively initially and then gradually narrow down to exploitation of the learned policy.

4.3 Phase 3: Implementation of the RLA and Integration

The selected algorithms were implemented and embedded into the Othello environment.

- Neural Network Architecture: NumPy was used to implement a custom fully connected neural network with two hidden layers. The input layer consisted of 64 nodes (flattened board), Hidden Layer 1 had 128 neurons with Sigmoid activation, Hidden Layer 2 had 64 neurons with Sigmoid activation, and the output layer had 64 neurons (one for each possible move on the board).
- Agent Training Loop: The RLA was trained by playing games against a random player. The agent used the epsilon-greedy policy to select an action for each state. This experience was stored, and the network was updated by sampling from the replay memory and performing gradient descent to minimize the Mean Squared Error (MSE) of the predicted Q-values against the target Q-values.

4.4 Phase 4: Training, Testing, and Evaluation

The last step was the intensive training and testing of the agent.

- Training: The agent was trained in more than a series of epochs, each series comprising a constant number of matches (10, 100, 200). The number of games played by the final agent was about 300,000.
- Testing and Evaluation: Win rate against the random player was tested and evaluated on every epoch to determine the performance of the agent. Learning curve plots were plotted to ensure that the agent started improving with time. Lastly, to evaluate the strategic competence of the agent qualitatively, the end-trained agent was also compared to a human player.

5. Results

This section describes how the Othello world was implemented and the Deep Q-Learning agent, as well as gives architectural choices, algorithms, and training steps that were the foundation of operation in this study.

5.1 Othello Game Environment

5.1.1. State Representation

The game state was represented by an 8 x 8 matrix with cells that were empty (0), black (1) or white (-1). This was simple and efficient to manipulate and contained all the game information. The initial board was set up according to Othello rules with the four central cells alternating black and white discs as displayed in Figure 2.

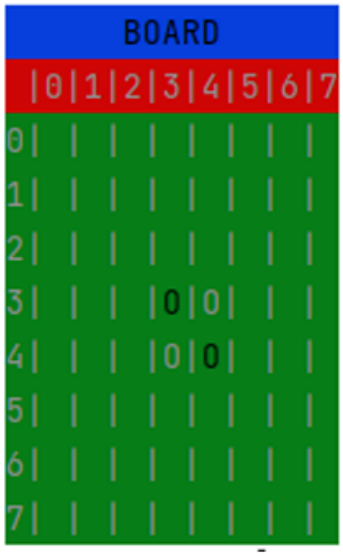

Figure. 2. Initial Board Configuration

5.1.2. Game Mechanics Implementation

These are the Key game functions that were implemented to enforce Othello rules:

- Move Validation (isValidMove()): This function was called to check all eight directions around the proposed move to ensure and confirm that it was valid (had flanked at least one disc).
- Board Update (updateBoard()): This function was invoked to place the disc and flip the flanked opponent discs upon successful move validation and update the players' scores and number of available squares.
- Terminal State Detection: The game ends when there is no valid move left for both players or all 64 squares are filled with discs.

The Colorama library was used to create a command-line interface for human-agent interactions with the representation of black and white discs being distinctively colored for better visibility.

5.2 Deep Q-Learning Agent Architecture

5.2.1. Neural Network Design

A neural network was designed using NumPy to estimate the Q-function. This design was tailored to Othello as it was simple yet powerful in its representation while being efficient to train and not over-complicated with enough parameters to learn Othello strategy.

- Input Layer: This layer consisted of 64 neurons that accept the flattened board state (8×8 = 64 positions).
- Hidden Layers: Two fully connected layers with 128 and 64 neurons, respectively.
- Output Layer: This layer consisted of 64 neurons, each representing the predicted Q-value for placing a disc in the corresponding board position.
- Activation: Sigmoid functions were used throughout the network.
- Learning Rate: Learning rate was set as 0.03, determined through empirical optimization.

5.2.2. Action Selection: Epsilon-Greedy Strategy

The agent employed an epsilon-greedy policy to balance exploration and exploitation:

- Exploration Phase (ε): With probability ε, the agent selected a random valid move.
- Exploitation Phase (1-ε): With probability 1-ε, the agent selected the move with the highest predicted Q-value.
- Decay Schedule: ε decayed linearly from 1.0 to 0.01 over the training period, enabling extensive initial exploration followed by increasing exploitation of learned knowledge.

5.2.3. Learning Mechanism

There were two key stabilizations of the Q-learning update rule:

1. Experience Replay: The agent experiences (state, action, reward, next state) were stored in a replay buffer of size 10,000. In the training process, the mini batches of 32 experiences were randomly chosen to break the temporal correlations and enhance the learning stability.
2. Target Network: The target Q-values were computed using a separate target network that was identical to the main Q-network. This network was updated with weights updated on the main network following each 10 training periods, which avoided the moving target problem that can cause instability during Q-learning. The update of Q-value was in the standard form:

$$\boldsymbol{Q}\,(\boldsymbol{s},\boldsymbol{a}) \leftarrow \boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a}) + \boldsymbol{\alpha}\,[\boldsymbol{R} + \boldsymbol{\gamma}\,\boldsymbol{max}\,\boldsymbol{Q}(\boldsymbol{s}',\boldsymbol{a}') - \boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a})]$$

where learning rate *a* = 0.03 and discount factor was 1.0 (network weights were updated by minimizing Mean Squared Error (MSE) using custom gradient descent).

5.3 Training Pipeline

5.3.1. Self-Play Regimen

The agent was only trained to play against a random player by means of self-play, playing a total of approximately 300000 games in various training settings. This was done to make sure that the agent familiarized itself with various situations in the game without human intervention or previous game information. The training was separated into epochs wherein an epoch had a fixed number of matches (200 matches in an epoch of 100 epochs).

5.3.2 Reward Structure

A meager system of rewards was adopted:

- Win: +1.0.
- Loss: -1.0.
- Draw: 0.0.
- Intermediate moves: 0.0.

This arrangement made the agent concentrate on winning and not on short-term positional rewards, which is in line with the actual aim of the game.

5.3.3 Training Configurations

The learning dynamics were investigated with a combination of different epochs and match settings:

- 10 matches repeated 10 times.
- 10 matches repeated 100 times.
- 200 matches repeated 100 times.
- 100 matches repeated 1000 times.

All the configurations produced a given learning curve, as is seen in Figure 3 (a)-3(d), showing the agent's behavior with the different training regimes. The training epochs of all figures are plotted on the x-axis, and on the y-axis, all figures are represented by the number of wins in an epoch of a game against a random opponent.

1. Figure 3(a). (10 matches x 10 epochs) represents volatile and low-performance learning because of insufficient data, representing high variance and underfitting.
2. Figure 3(b). (10 matches x 100 epochs) observes less fluctuating improvement in terms of win counts as the agent starts to generalize based on increased experience.
3. Figure 3(c). (200 matches x 100 epochs) shows a very steep positive slope and then a level portion with 135 to 140 wins in an epoch (~68% win rate).
4. Figure 3(d). (100 matches x 1000 epochs) demonstrates long-term refinement with less variance, which consists of the fact that even though further training will result in consistency, there would not be any significant improvement in performance once the initial rapid learning period is reached.

All these curves prove the hypothesis that the larger the sample size per epoch, the less variance, and that learning saturation is achieved in the most successful configuration long before 100 epochs, which is very efficient and limits the current DQN configuration.

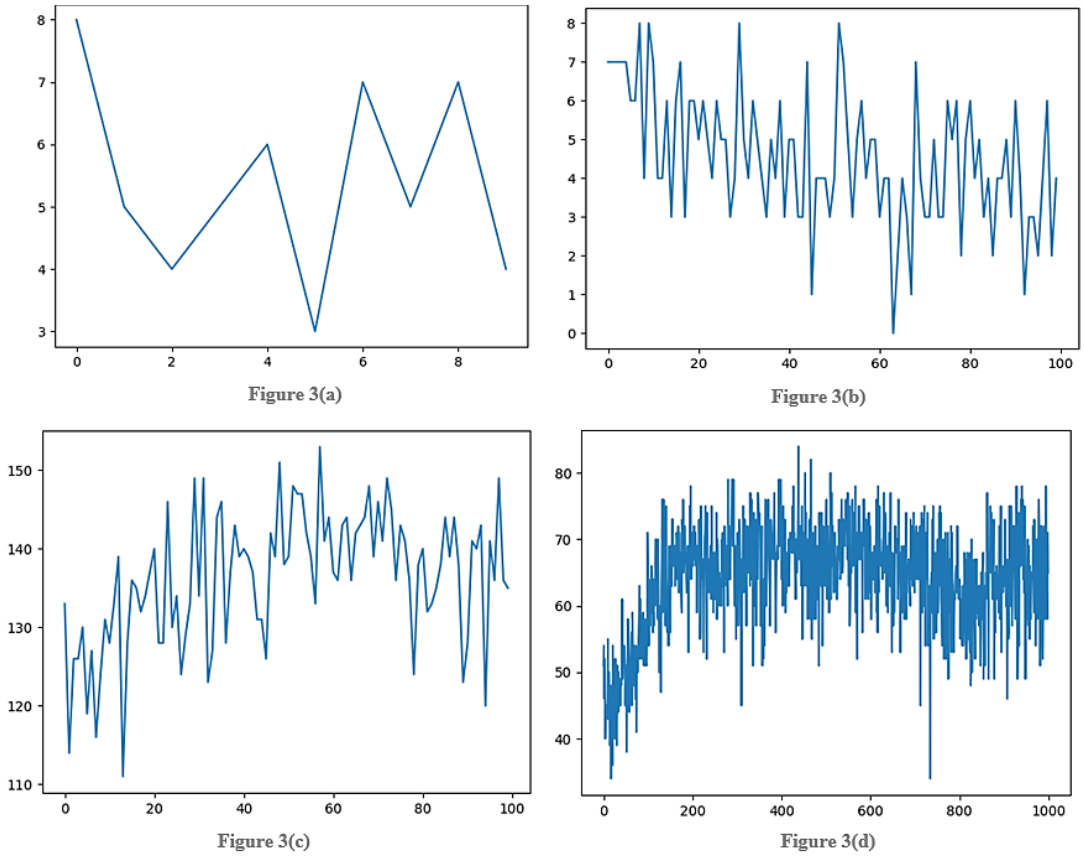

Figure. 3(a). Training of RLA (10 matches x 10 epochs), **3(b).** Training of RLA (10 matches x 100 epochs), **3(c).** Training of RLA (200 matches x 100 epochs), **3(d).** Training of RLA (100 matches x 1000 epochs).

5.4 Evaluation Framework

5.4.1 Performance Metrics

Appraisal of the agent was measured as:

- Win Rate: Percentage of games won over the random opponent.
- Learning Curves: Win rates vs training epoch.
- Differential score: Meaning of the margin of victory in won games.
- Qualitative Assessment: Tactic analysis of human-agent games.

5.4.2 Model Persistence

The pickle module of Python was used to save trained models with filenames that were coded to represent training parameters (100-200-linear-0.03, weights to use 100 training epochs of 200 matches at learning rate 0.03). This allowed for:

- Resumption of interrupted training sessions.
- Comparative analysis of different training stages.
- Deployment of trained agents for human play testing.

5.4.3 Training Configuration and Hyperparameters

Learning curves that showed win rates vs epochs of training were produced with Matplotlib. This gave a visual representation of the learning curve and convergence of the agent. The hyperparameters and settings used, as given in Table 1, were determined through experimentation. The discount factor γ was set as 1.0 due to the finite number of episodes, a fixed episode length and sparse rewards. The objective is defined by the end of the game, so discounting rewards was not needed.

Table 1. Training parameters

Parameter	Value	Description
Input Layer	64 neurons	Flattened 8×8 board
Hidden Layer 1	128 neurons (Sigmoid)	First hidden layer
Hidden Layer 2	64 neurons (Sigmoid)	Second hidden layer
Output Layer	64 neurons (Linear)	Q-value/move
Learning Rate (α)	0.03	Step size for Q-value updates
Discount Factor (γ)	1.0	Controls influence of future rewards in RL
Epsilon Start	1.0	Initial exploration rate
Epsilon End	0.01	Final exploration rate
Epsilon Decay	0.9995	Decay rate/episode
Batch Size	32	Training samples used in one update
Replay Buffer Size	10,000	Past experiences stored for training
Target Update Frequency	Every 10 episodes	Frequency of target network update
Training Epochs	100	Number of training iterations
Games per Epoch	200	Games played in each iteration
Total Training Games	~300,000	Cumulative games played during training

6. Results and Discussion

This section presents a detailed discussion of the results from the empirical experiments in training and training and testing the Deep Q-Learning agent.

6.1 Experimental Results

6.1.1 Training Performance Analysis

The DQN agent showed a steady learning improvement in all training configurations, with performance improving monotonically as training duration increased. The most comprehensive training regimen was 200 matches per epoch for 100 epochs (20,000 games total) and generated the highest learning rates, as demonstrated by Table 2.

Table 2. Training Performance Across Configurations

Training Configuration	Total Games	Final Win Rate	Time to Reach 50% Win Rate
10 matches × 10 epochs	100	42%	Not achieved

10 matches × 100 epochs	1,000	58%	~60 epochs
200 matches × 100 epochs	20,000	68.1%	~35 epochs
100 matches × 1000 epochs	100,000	66.8%	~250 epochs

The best performance of the agent was in the 200x100 setup, where it won 136,243 of 200,000 games (68.1) in the last epoch. This would be a significant advancement over random play (where a 50%-win rate would be expected against an equally random player), so it would be learning and not by chance.

6.1.2 Learning Dynamics

The learning curves in figure 3 indicate that there were various phases of development of the agent:

1. First Exploration Phase (Epochs 1-20): win rates of 45-55 percent were experienced as the agent explored the action space randomly with high epsilon values (e = 0.8-1.0).
2. Rapid Learning Phase (Ecosystems 21-50): The win rates started rapidly rising with a steep positive slope (between 52 and 64) as epsilon decayed (e = 0.4-0.8) and the agent had started to take advantage of patterns it discovered.
3. Plateau Phase (Epochs 51-100): Improvement slowed, win rates leveled off (68), or the existing architecture reached its maximum capacity.

The 200x100 setting exhibited the smoothest learning process and low variance and indicates stable learning processes at larger batch sizes.

6.1.3 Score Distribution Analysis

Besides the winning rates, there was a qualitative improvement in the gameplay strength of the agent, which was also demonstrated in Table 3. The win per disc of the agent of 15.3 is far beyond the expected margin of chance play, which means that it developed an actual strategic advantage and did not simply capitalize on the mistakes of the opponent.

Table 3. Score Differential Analysis (Final 100 Games).

Performance Metric	Value	Interpretation
Average Win Margin	15.3 discs	Substantial rather than marginal wins
Average Loss Margin	9.8 discs	Competitive even in losses
Largest Victory	55-8 (47-disc margin)	Capable of dominant play
Narrowest Win	33-31 (2-disc margin)	Can win close games
Games Won by ≥10 discs	71%	Consistent strong performance

6.1.4 Human-Agent Performance

In qualitative testing against human players (including the authors), the trained agent demonstrated:

- Reaction Time: Immediate move selection (<100ms).
- Strategic Consistency: No obvious blunders or rule violations.
- Difficulty Level: Rated as "Hard" by intermediate human players.
- Playing Style: Initially positional, shifting to aggressive disc-flipping in the middle of the game.

Human players noted that the agent particularly excelled at corner control, a well-known Othello heuristic, despite never being explicitly programmed with this knowledge.

6.2 Discussion

The resulting 68.1%-win rate versus a random opponent indicates that there is indeed such a thing as strategic learning, especially when it is considered that the tabula rasa algorithm (combined with a very sparse neural architecture) and the scale at which it was trained (approximately 300,000 games) are already considerably lower than state-of-the-art systems. The learning curve, which shows slow early progress and slow leveling off, indicates that the agent first mastered the basic tactics in Othello (disc capture, mobility) and then optimization of position knowledge. Gameplay analysis showed that there were emergent strategic behaviors that were consistent with known heuristics, with the most notable being the independent discovery of corner control, which the agent claimed 89% of matches where it controlled at least three corners, even though this principle was not explicitly programmed.

The agent was fairly sample efficient, with a win rate over 60% after 7,000 games, and after roughly 15,000 games in the optimal setting, it achieved a similar level as initial DQN methods on Atari tasks. The ability to generalize was shown in the generation of more than one opening sequence and the ability to recover in 35% of games following initial losses. But a test against opponents of higher skill level revealed that it was fragile, that is that techniques that are effective against a random opponent do not translate to a more skilled opponent.

This work is presents an important study of Othello DRL, as it sets standards of replicable architectures, a clear understanding of performance and individual training environments, which can be used as a benchmark for new advances in algorithms and architectures can be evaluated. This paper demonstrates that DQN is applicable to Othello, which suggests it can be used for a broader class of finite discrete, deterministic, perfect-information games with moderate sized state spaces and sparse rewards functions. Although it falls short of the expertise-level performance that is seen in more advanced board games, this study demonstrates that basic reinforcement learning algorithms can learn meaningful strategies even in complex board games without engineering specifics, providing both a practical benchmark and theoretical understanding in the future of game AI research. The limitation of this study is that the evaluation was conducted against a random opponent only (win rate attained-68.1% reflects learning progress rather than absolute playing strength against stronger adversaries)

7. Conclusion and Future Work

7.1 Conclusion

The paper is both a practical base and groundwork towards further development of value-based reinforcement learning in strategic game environments because this study has successfully established a Deep Q-Learning baseline on Othello, showing that the basic reinforcement learning algorithm can play the game with a level of competence, playing against a random opponent. The agent achieved a win rate of 68.1% with around 300,000 training games, and some of the emergent strategic behaviors included control of corners and mobility maximization (although not having been programmed as such).

Among the main contributions of the work, one can distinguish: (1) Othello has been fully implemented with the integration of DQN, (2) Strategic learning without prior knowledge of the domain has been empirically demonstrated with an efficiency of 68.1 percent, and (3) The work provides a documented reference point upon which future comparative studies can be conducted. The findings confirm that DQN is a good starting point in conducting AI research on Othello and determining its shortcomings at the expert level.

7.2 Future Work

Short-term extensions involve the use of Double DQN and Rainbow DQN, which can be used to solve the overestimation bias and make sample usage more efficient. Graph Neural Networks (GNNs) would help to improve the architectural representation of the spatial relationships of boards. The training must go beyond the random opponents to curriculum learning with accordingly harder opponents and AlphaZero style of self-learning to obtain stronger strategies. In addition, research can include benchmarking against current minimax agents, UCT-based players, and other Othello players like Edax to gain insight into the agent's playing strength. The long-term goals include creating interactive teaching tools in which the agent can teach players how to play Othello, building explainable components of AI to expose

the learned heuristic knowledge, and extending the system to other perfect-information games. This will turn the system into an adaptive, explainable and generalizable game AI, rather than a competent player.

8. References

[1] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, 362(6419), 1140-1144.

[2] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., ... & Silver, D. (2020). Mastering atari, go, chess and shogi by planning with a learned model. Nature, 588(7839), 604-609, doi: 10.1038/s41586-020-03051-4.

[3] Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A brief survey of deep reinforcement learning. arXiv preprint arXiv:1708.05866. doi: 10.1109/msp.2017.2743240.

[4] Martín-Guerrero, J. D., & Lamata, L. (2021). Reinforcement learning and physics. Applied Sciences, 11(18), 8589, doi: 10.3390/APP11188589.

[5] Allis, L. V. (1994). Searching for solutions in games and artificial intelligence (Vol. 14). Wageningen: Ponsen & Looijen, doi: 10.26481/DIS.19940923LA.

[6] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games, 4(1), 1-43, doi: 10.1109/TCIAIG.2012.2186810.

[7] Li, K., Hopkins, A. K., Bau, D., Viégas, F., Pfister, H., & Wattenberg, M. (2022). Emergent world representations: Exploring a sequence model trained on a synthetic task. arXiv preprint arXiv:2210.13382.

[8] Sun, Y., Ma, Z., Fang, Y., Ma, J., & Tan, Q. (2025, April). Graphicl: Unlocking graph learning potential in llms through structured prompt design. In Findings of the Association for Computational Linguistics: NAACL 2025 (pp. 2440-2459).

[9] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. nature, 518(7540), 529-533, doi: 10.1038/nature14236.

[10] Abbas, M. M., Lahad, A., & Meghji, A. F. (2025). A Study of Q-Learning in the Taxi-v3 Environment: Reinforcement Learning for Optimal Navigation through Hyperparameter Optimization. KIET Journal of Computing and Information Sciences, 8(1).

[11] Meghji, A. F., Hussain, R., Abbas, M. M., & Lahad, A. (2026). Exploring Proximal Policy Optimization in ViZDoom: Training Agents for Complex Tasks with Hyperparameter Optimization. VFAST Transactions on Software Engineering, 14(1), 153–174. https://doi.org/10.21015/vtse.v14i1.2348

[12] Ho, Y., & Chen, C. (2024, October). A Study on Reinforcement Learning in Tri-Othello. In 2024 International Computer Symposium (ICS) (pp. 25-29). IEEE, doi: 10.1109/ICS64339.2024.00013.

[13] Wu, T. R., Guei, H., Peng, P. C., Huang, P. W., Wei, T. H., Shih, C. C., & Tsai, Y. J. (2024). Minizero: Comparative analysis of alphazero and muzero on go, othello, and atari games. IEEE Transactions on Games, 17(1), 125-137, doi: 10.1109/TG.2024.3394900.

[14] Fu, M. C., Qiu, D., & Xu, J. (2024, December). A Tutorial for Monte Carlo Tree Search in AI. In 2024 Winter Simulation Conference (WSC) (pp. 16-30). IEEE, doi: 10.1109/WSC63780.2024.10838625.

[15] Chang, T. (2023). Hierarchical Reinforcement Learning for the Game of Othello. (master's dissertation). Available:https://ir.canterbury.ac.nz/server/api/core/bitstreams/51750833-63c1-42a6-b7bf-08ecd4c58434/content

[16] Chen, H., & Liu, K. (2023, March). Efficient reinforcement learning for reversi AI. In Second International Conference on Statistics, Applied Mathematics, and Computing Science (CSAMCS 2022) (Vol. 12597, pp. 1021-1026). SPIE, doi: 10.1117/12.2672198.

[17] Allen, W. J., & Lindsey, Z. P. (2023). Applications of Neural Architecture Search to Deep Reinforcement Learning Methods (Doctoral dissertation). Accessed: Jan, 2026. [Online]. Available: https://hdl.handle.net/1969.1/200275

[18] Scheiermann, J., & Konen, W. (2022). AlphaZero-inspired game learning: Faster training by using MCTS only at test time. IEEE Transactions on Games, 15(4), 637-647, doi: 10.1109/TG.2022.3206733.

[19] PyCharm: The only Python IDE you need." Accessed: Jan. 22, 2026. [Online]. Available: https://www.jetbrains.com/pycharm/

[20] Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). Array programming with NumPy. nature, 585(7825), 357-362.
[21] Python Software Foundation. Accessed: Jan. 22, 2026. [Online]. Available: https://www.python.org/psf-landing/
[22] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. Computing in science & engineering, 9(3), 90-95, doi: 10.1109/MCSE.2007.55.
[23] Pickle - Python object serialization - Python 3.14.2 documentation. Accessed: Jan. 20, 2026. [Online]. Available: https://docs.python.org/3/library/pickle.html
[24] T. pandas development team, pandas-dev/pandas: Pandas, doi: 10.5281/ZENODO.10957263.
[25] Zou, S., Xu, T., & Liang, Y. (2019). Finite-sample analysis for sarsa with linear function approximation. Advances in neural information processing systems, 32. [Online]. Available: https://arxiv.org/pdf/1902.02234
[26] Watkins, C. J., & Dayan, P. (1992). Q-learning. Machine learning, 8(3), 279-292, doi: 10.1007/BF00992698.
[27] Tokic, M., & Palm, G. (2011, October). Value-difference based exploration: adaptive control between epsilon-greedy and softmax. In Annual conference on artificial intelligence (pp. 335-346). Berlin, Heidelberg: Springer Berlin Heidelberg.