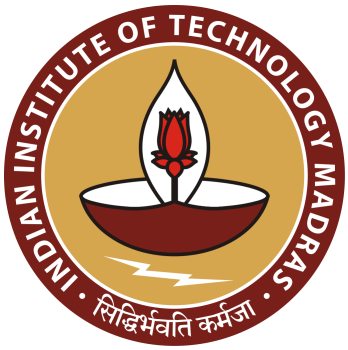
IDENTIFICATION OF STRATEGIES IN MULTIPLAYER GAMES USING REINFORCEMENT LEARNING



Department of Mathematics, IIT Madras

Name of the Project Guides

Dr. Sivaram Ambikasaran

Dr. Sri Vallabha Deevi (Tiger Analytics)

Submitted by

Tumpa Jalua (MA22M024)

May 10, 2024

INDIAN INSTITUTE OF TECHNOLOGY MADRAS
DEPARTMENT OF MATHEMATICS

NAME: Tumpa Jalua

Roll No: MA22M024

PROGRAMME: M.Tech.

DATE OF JOINING: 25.07.2022

Guide: Dr. Sivaram Ambikasaran

Co-guide: Dr. Sri Vallabha Deevi

Certificate

This is to certify that the report **"IDENTIFICATION OF STRATEGIES IN MULTIPLAYER GAMES USING REINFORCEMENT LEARNING"**, submitted by **Tumpa Jalua (MA22M024)** in partial fulfillment of the requirements for the degree of Master of Technology in Industrial Mathematics and Scientific Computing at the Indian Institute of Technology Madras, Chennai, is the record work done by her during the academic year 2023-2024 in the Department of Mathematics, IIT Madras, India, under my supervision.

Dr. Sivaram Ambikasaran
Department of Mathematics,IIT Madras
May 10, 2024

Dr. Sri Vallabha Deevi
Tiger Analytics,Chennai
May 10, 2024

Acknowledgements

First of all, I would like to thank the Almighty for granting me perseverance. I would like to express my gratitude to Dr. Sivaram Ambikasaran, Department of Mathematics, IIT Madras, and Dr. Sri Vallabha Deevi, Tiger Analytics. Their unwavering guidance, exceptional support, and constant encouragement have been the cornerstones of my journey through this thesis. I was truly fortunate to have the opportunity to work under them as a student. It was an honor and a privilege to work with them. Their assistance with technical writing and presentation style was extremely valuable, without which it would not have been possible for me to complete this project work. Also, I would like to thank the Department of Mathematics at IIT Madras and Tiger Analytics for providing me with the opportunity to work in this exciting field. Lastly, and most importantly, I would like to thank my parents for their years of unyielding love and encouragement. They have always wanted the best for me, and I admire their determination and sacrifice in supporting me through this journey.

Tumpa Jalua (MA22M024)
Department of Mathematics, IIT Madras
May 10, 2024

Abstract

In this project, we explore the application of Reinforcement Learning (RL) in playing two-player games such as Tic-Tac-Toe and Othello. We train RL agents to play the game of Tic-Tac-Toe using Q-learning approach. The agents, representing two players, play many games against each other to learn and refine their own policy. At the end of the training, there are two policies learned, first-player policy and second-player policy. To assess the learning, three strategies are compared on test games: 1) both players use Random Policy, 2) The first player uses learnt RL policy and the second player plays Randomly, and 3) The first player plays Randomly while the second player uses learnt Policy. We evaluate their performance by calculating the outcomes on a series of games for each strategy. Agent learning is demonstrated by comparing the outcomes of random games with games that use learnt policy. During random games of Tic-Tac-Toe, we observe that the first player has a marginal advantage over the second player. RL trained first player outperforms this random first player, demonstrating learning. Subsequently, in the Tic-Tac-Toe game, we transition to employing the Deep Q-Network (DQN) algorithm, a function approximation method based on neural networks, instead of Q-learning with a lookup table. DQN allows learning over a large number of states, that cannot be tabulated effectively in the Q learning approach. We train an RL agent using the TensorFlow Keras with the DQN algorithm. Comparing the proportion of wins in a series of games demonstrates RL learning, as we did in the Tic-Tac-Toe game using tabulated Q learning. Moving forward, we extend our approach to a bigger game, Othello. Our aim is to understand RL learning with simpler games that require less computational power to train using the TensorFlow Keras, employing the DQN (Deep Q-Network) algorithm. Similar to Tic-Tac-Toe, we compare three different strategies to evaluate learning. It is observed that learnt policy provides an advantage over random policy for either player in this game. With this learnings, we are confident of the ability of RL to handle broader and more general problems like online advertising, e-commerce and applications in digital platforms.

Keywords: Artificial Intelligence, Reinforcement Learning (RL), Q-learning, Deep Q-Network (DQN), Markov Decision Processes, Neural Networks, Tic-Tac-Toe, Othello.

List of Notations

- **A(s)** - set of actions possible in state s
- **R** - set of possible rewards
- **t** - discrete time step
- **T** - final time step of an episode
- $\mathbf{S_t}$ - state at t
- $\mathbf{A_t}$ - action at t
- $\mathbf{R_t}$ - reward at t, dependent, like S_t, on A_{t-1} and S_{t-1}
- $\mathbf{G_t}$ - return (cumulative discounted reward) following t
- $\mathbf{G^{(n)}_t}$ - n-step return
- $\mathbf{G^{\lambda}_t}$ - λ-return, where λ is called the discount factor
- π - policy, decision-making rule
- $\pi(s)$ - action taken in state s under a deterministic policy π
- $\pi(a|s)$ - probability of taking action a in state s under a stochastic policy π
- $\mathbf{v_\pi(s)}$ - value of state s under policy π (expected return)
- $\mathbf{v^*(s)}$ - value of state s under the optimal policy
- $\mathbf{q_\pi(s, a)}$ - value of taking action a in state s under policy π
- $\mathbf{q^*(s, a)}$ - value of taking action a in state s under the optimal policy
- $\mathbf{V_t(s)}$ - estimate (a random variable) of $v_\pi(s)$ or $v^*(s)$
- $\mathbf{Q_t(s, a)}$ - estimate (a random variable) of $q_\pi(s,a)$ or $q^*(s,a)$

Contents

1 Introduction **3**

1.1 Digital Interactions . . . 3

1.2 Reinforcement Learning for two player games . . . 3

1.3 Q-learning and Deep Q Network . . . 4

1.4 Outline of thesis . . . 5

2 Machine Learning and Reinforcement Learning **6**

2.1 Machine Learning . . . 6

2.1.1 Supervised Learning . . . 6

2.1.2 Unsupervised Learning . . . 7

2.1.3 Reinforcement Learning . . . 8

2.2 Application of Reinforcement Learning (RL) . . . 9

2.3 Markov Decision Process . . . 10

2.3.1 Reinforcement Learning Components: . . . 10

3 Reinforcement Learning algorithms **12**

3.1 Q-learning algorithm . . . 12

3.2 Exploration and Exploitation . . . 13

3.3 Neural Network for RL . . . 14

3.3.1 Activation Function . . . 15

3.4 Bellman Equation . . . 15

3.5 Deep Q-Network (DQN) . . . 16

4 Tic-Tac-Toe game using Q-learning **17**

4.1 Setting up the Game Environment . . . 17

4.2 Training the Agent . . . 19

4.3 Testing the trained agent . . . 23

4.4 Results . . . 24

5 Tic-Tac-Toe game using Neural Network **26**

5.1 Setting up the Environment . . . 26

5.2 Training the Agent 26
5.3 Testing trained agent 30
5.4 Results 31

6 Othello game using Deep Q-Network **33**
6.1 Setting up the Environment 34
6.2 Othello game Strategies using RL 36
6.3 Training the Agent 37
6.4 Testing the trained agent 42
6.5 Results 44

7 Conclusions **46**
7.1 Future Work 46

Chapter 1

Introduction

1.1 Digital Interactions

In today's digital world, customer interactions with companies are primarily digital. We search for items, compare prices, and purchase via mobile applications. Customer interactions with e-commerce companies can be modeled as two-player games. The customer is one player trying to find products/services at optimal prices and maximize discounts. The company is the other player trying to maximize revenue/profit from the customer, by influencing them to purchase. By understanding the user better, companies can learn to make better predictions and optimize strategies to increase revenue by providing the right offers. Two-player games are tricky because we don't always know how the other player will respond to a particular move. The game does not always proceed in the same sequence of steps that make a player win. Which is why two-player games are complicated. However, Two-player games are useful for both digital advertising and e-commerce platforms. RL is a suitable method for teaching agents to play two-player games.

1.2 Reinforcement Learning for two player games

Reinforcement learning (RL) [1] is a powerful paradigm in artificial intelligence where agents learn to play multi-step games. After a sequence of steps, the agent receives a reward that summarizes whether the sequence of steps was good or bad in terms of achieving the target goal. Unlike supervised learning, RL doesn't rely on predefined target actions but learns through trial and error, exploring the environment, taking actions, and adjusting based on feedback. This feedback, be it rewards or punishments, shapes the agent's future decisions. RL algorithms strive for a balance between exploitation and exploration, often employing strategies like epsilon-greedy [2], which guides the agent in choosing actions to either maximize returns based on past experiences or explore uncharted territories potentially offering higher returns. While too much exploitation can lead to suboptimal solutions, excessive exploration may slow convergence.

Reinforcement learning (RL) operates within the framework of the Markov Decision Process (MDP), making decisions in discrete time steps. At each step, the agent selects an action based on

the current environment state, which is influenced by previous actions. Policies help guide the agent's action selection; correct actions lead to positive rewards, while incorrect actions result in negative rewards or penalties. Various algorithms, such as Q-learning, Policy gradient method, Monte Carlo method, SARSA, etc., are utilized in RL. RL is a flexible approach that can be combined with other machine learning algorithms, such as deep learning, to enhance model performance. It is utilized to address a wide range of problems, including those about decision-making, optimization, and control systems.

1.3 Q-learning and Deep Q Network

In 1989, Watkins first introduced Q-learning, a model-free reinforcement learning algorithm [3]. Widyantoro et al. (2009) applied a Q-learning algorithm to play Tic-Tac-Toe. Q-learning algorithm can effectively teach an agent to play a simple classical game like Tic-Tac-Toe and learn an optimal strategy for the game. The method is described here [4]. Deep Q-learning (DQN) represents the neural network implementation of Q-learning [5]. To address the challenge posed by large state-action spaces, an effective approach involves utilizing a function approximation method, such as a neural network technique. The underlying concept of employing a function approximation method is to refrain from individually storing the Q-value for each state-action pair; instead, Q-values are stored as a function of the state and action. Neural networks, known for their ability to capture and represent complex input/output relationships, serve as a suitable tool for this purpose.

Deep reinforcement learning (DRL) [6] combines RL with deep learning techniques, leveraging neural networks to learn optimal strategies. The success of DRL was exemplified by the combination of CNNs with RL in playing Atari games [7]. Recent advancements in DRL focus on enhancing sample efficiency and learning speed through methods like semi-supervised learning and self-supervised learning [8].

In this project, we explore the application of Reinforcement Learning techniques in the context of two-player games, with a specific focus on Tic-Tac-Toe [9] and Othello [10]. These games serve as illustrative examples to explore the capabilities of RL algorithms in learning optimal strategies through interaction with the game environment. We investigate the efficacy of RL algorithms, such as Q-learning [11] and Deep Q-Networks (DQN) [12], in training agents to play these games and analyze their performance under different scenarios.

1.4 Outline of thesis

In Chapter 2, we will introduce the fundamental concepts of machine learning and reinforcement learning, accompanied by real-life applications of reinforcement learning and the Markov decision process. Moving on to Chapter 3, we will explore various reinforcement learning algorithms, including Q-learning, and the DQN algorithm. Chapter 4 will center on the application of Q-learning to the Tic-Tac-Toe game. In Chapter 5, we will investigate the utilization of a neural network, specifically the DQN algorithm, in enhancing the Tic-Tac-Toe game. Furthermore, Chapter 6 will scrutinize the implementation of the DQN algorithm in the game of Othello. Finally, Chapter 7 will present the conclusion of our thesis.

Chapter 2

Machine Learning and Reinforcement Learning

Machine learning history started in 1943 with the first mathematical model of neural networks presented in the scientific paper "A logical calculus of the ideas immanent in nervous activity" by Walter Pitts and Warren McCulloch [13]. Then, in 1949, the book " The Organization of Behavior" by Donald Hebb was published. In 1950 Alan Turing proposed the Turing Test [14] to determine if a computer has real intelligence. To pass the test, a computer must be able to fool a human into believing it is also human.

2.1 Machine Learning

Machine learning (ML) is a field of study in artificial intelligence concerned with the development and study of statistical algorithms that can learn from data and generalize to unseen data, and thus perform tasks without being explicitly programmed. ML algorithms are designed to analyze and interpret large amounts of data, identify patterns, and extract meaningful insights or make predictions. The learning process involves training the algorithms on historical or labeled data to build models that can generalize and make predictions on new, unseen data. The key goal of ML is to develop algorithms that can automatically learn and adapt from data, improving their performance over time without explicitly being programmed for every task. Machine learning is used in various applications such as image recognition, natural language processing, recommender systems, fraud detection, autonomous vehicles, etc. ML can be broadly classified into three areas- Supervised Learning, Unsupervised Learning, and Reinforcement Learning

2.1.1 Supervised Learning

In supervised learning [15], a model is trained to make predictions by learning from labeled data. The model is provided with input data as well as their respective correct outputs, through which it acquires the ability to predict outcomes for new data, leveraging its training. This enables the algorithm to correctly determine output values for unseen instances by generalizing from the training data to unseen situations in a reasonable manner. This allows the model to perform tasks such as

classification and regression. The regression algorithm is used to model a linear relationship between input features and output features, primarily for predicting continuous variables. There are several regression algorithms, including Linear Regression, Polynomial Regression, etc. In classification algorithms, the output variable is typically categorical, indicating Pass-Fail, Yes-No, etc. There are several supervised classification algorithms, such as Logistic Regression, Decision Trees, Random Forests, etc. For example, let's say we want to teach a computer to recognize pictures of dogs. We can show it pictures of different breeds of dogs along with name of each breed. Then the computer will try to figure out which characteristics typically go with each breed of dog. The process is shown below in Fig 2.1.

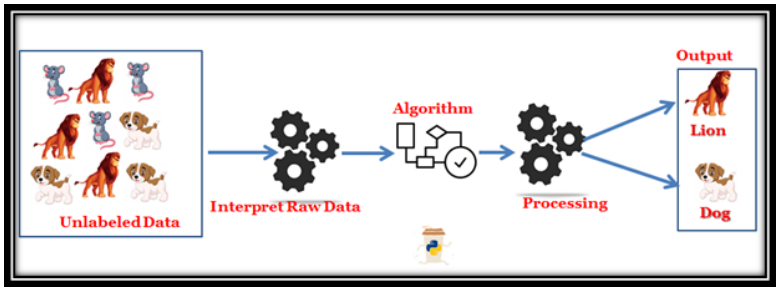


Figure 2.1: Supervised Learning [16].

2.1.2 Unsupervised Learning

Unsupervised learning is a method in machine learning where algorithms learn from data without labels. The aim is for the machine to understand its surroundings and create new ideas from it, similar to how humans learn by observing and exploring. Consequently, there is no definitive way to assess the correctness of the outcome, distinguishing it from supervised learning. These algorithms typically seek to discover hidden structures or patterns within the data by grouping or clustering similar data points together based on their features.

For example, bank fraud detection, where patterns of fraudulent behavior may emerge from the data without explicit labels, and recommendation systems, which can identify groups of users with similar preferences to make personalized suggestions. The process is shown in Fig 2.2.

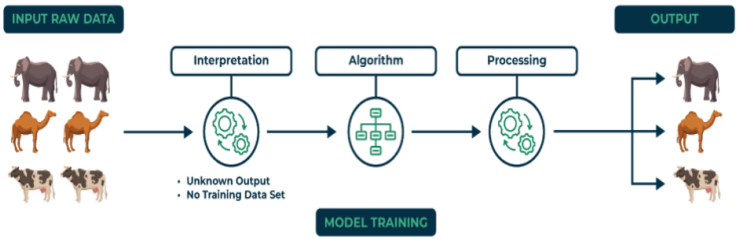


Figure 2.2: Unsupervised-learning [17].

2.1.3 Reinforcement Learning

Reinforcement learning, a field bridging machine learning and optimal control, deals with how smart agents should act in changing environments to maximize the cumulative reward. Reinforcement learning differs from supervised learning in that it doesn't need labeled data for training or explicit correction of wrong actions. Instead, it's about striking a balance between exploring new options and exploiting them to maximize the long-term reward, even when feedback may be incomplete or delayed [18].

For example, consider training an autonomous agent to navigate through a maze. The agent explores different paths, receiving positive rewards for finding the exit and negative rewards for hitting dead-ends. Over time, the agent learns which actions lead to higher rewards and adjusts its strategy accordingly to efficiently navigate the maze and maximize its cumulative reward. Reinforcement learning algorithms often model the environment as a Markov decision process (MDP), enabling them to use dynamic programming techniques. An outline of the approach is shown in Fig.2.3.

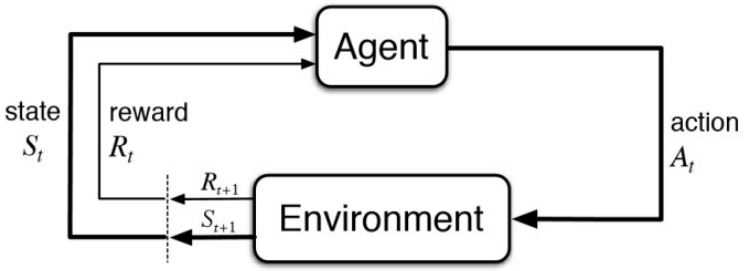


Figure 2.3: RL components [19] .

2.2 Application of Reinforcement Learning (RL)

RL has found applications in various domains. Some examples where RL is used are

- **Autonomous Vehicles:** RL plays a vital role in this domain because the more realistic our programs become, the better the machines perform. In autonomous learning for cars, there exists a reward system; if a car executes well, it receives a point, whereas if it fails a specific task, such as deviating from its lane, a point is deducted [20].

- **Robotics:** In today's world, RL is applied to various robotic platforms. RL allows a robot to autonomously discover optimal behavior by interacting with its environment through trial and error. Instead of explicitly outlining the solution to a problem, in RL, the designer of a control task provides feedback using a scalar objective function that measures the one-step performance of the robot [21].

- **Recommendation Systems:** In this area, RL is more useful because it leverages user preferences and item popularity from e-commerce platforms. These systems learn from user interactions and feedback to enhance the relevance and accuracy of recommendations over time [22].

- **Healthcare:** RL has more useful applications in healthcare domains ranging from dynamic treatment regimes in chronic diseases and critical care to automated medical diagnosis from both unstructured and structured clinical data, as well as many other control or scheduling domains that have infiltrated many aspects of a healthcare system. RL algorithms can optimize treatment plans for individual patients based on their medical history and genetic profile [23].

- **Video Games:** The video game, which builds a simulated world that accepts user inputs and outputs the corresponding result to the users, naturally provides an environment for the interaction of agents in the RL. The RL shows its power in playing video games that have a goal to achieve, like platform jumping, chess games, and other fields. These agents learn to play games by interacting with the game environment and optimizing their strategies to achieve high scores or complete objectives.

2.3 Markov Decision Process

A Markov decision process (MDP) [24] is a mathematical framework used to model reinforcement learning problems. Markov decision processes formally describe an environment that is fully observable for reinforcement learning. The agent is the learner and decision-maker. The environment is the system that the agent is interacting with. This interaction is continuous: the environment presents a state to the agent, the agent takes an action, and the environment responds to the action, generating a new state. This process is repeated at each time step, and the agent receives a reward, usually at the end of the episode or periodically in case of continuous games. The agent may not necessarily see a reward after each step. If there is a reward after each step, such a game can be treated as a supervised learning problem as well. The reward is a numerical representation of the value of the action that the agent took in a previous situation. The agent's goal is to maximize the overall reward over time.

MDPs provide a flexible and powerful framework for modeling sequential decision-making problems and training RL agents to make optimal decisions in uncertain environments. MDP contains a tuple of 4 elements (S, A, P_a, R_a):

- S is a finite set of states.
- A is a set of actions possible in state s.
- P_a is a state transition probability matrix at action a.
- R_a is a reward function due to action a.

2.3.1 Reinforcement Learning Components:

- **Agent -** The learner and decision maker that interacts with an environment. The agent's goal is to learn how to make decisions that maximize cumulative rewards over time.
- **Environment -** A system that a learning agent is interacting with. It provides feedback to the agent based on its actions and determines the outcomes of those actions. The environment may be deterministic or stochastic, and it can be discrete or continuous.
- **State -** The state is the information that is used to determine what happens next, depending on the agent's selected action and the environment. In a Markov Decision Process, the state encapsulates the history of the all past actions that led the agent to the current scenario based

on the current state, without needing to consider the path taken to reach that state. The state enables the agent to decide on the action based on its current situation and goals.

- **Action:** An action refers to a decision or choice made by an agent within an environment. The agent learns which actions to perform by observing the environment and the results of its actions. The ultimate goal of an RL agent is to learn the optimal actions to maximize its cumulative reward over time.

- **Reward system -** A crucial component of reinforcement learning that provides feedback to the agent. For each correct action, the agent receives a positive reward, and for incorrect actions, it receives a negative reward.

The RL components are shown in Fig. 2.4

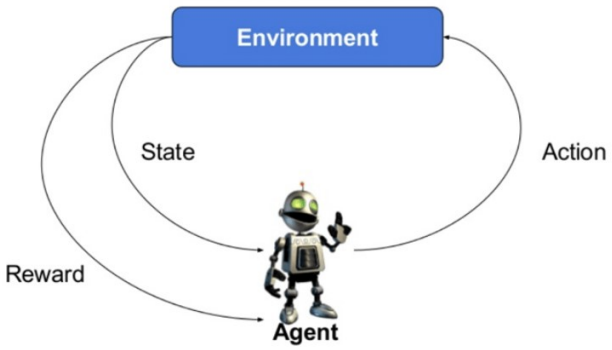


Figure 2.4: RL components [25] .

Chapter 3

Reinforcement Learning algorithms

3.1 Q-learning algorithm

The Q-learning algorithm [26] is a model-free reinforcement learning algorithm used to estimate the value of taking a specific action in a given state. Its objective is to learn a policy that maximizes the expected cumulative reward over time. The state identifies the current position in the agent's environment, determining the action to perform. The algorithm employs a Q-table of state-action pairs, also known as Q values. The state space represents possible agent states, while the action space represents potential actions. The Q-table is a matrix storing the expected reward for each action in each state, initialized to zero. In each iteration, the agent selects an action in the current state based on the highest Q value, executes it, receives immediate reward, and observes the next state. The reward is determined by the environment. The algorithm iteratively updates Q-values using the Bellman equation, representing the expected cumulative reward for a specific action in a given state. Updating continues until the convergence of the Q function to the optimal Q function. A simple example is shown in Fig.3.1 and a sample Q table is shown in Table 3.1.

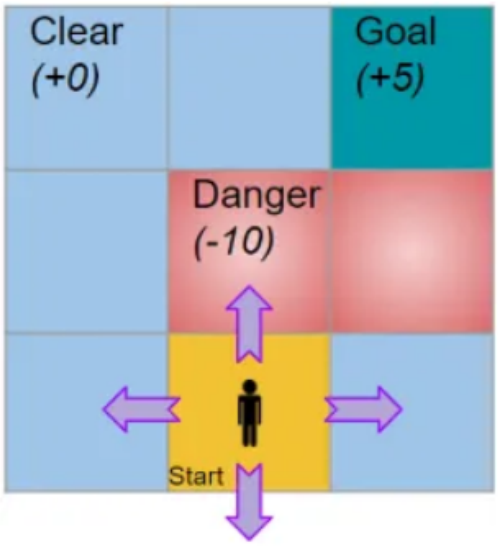


Figure 3.1: Example of Q-learning [27].

State	Actions			
	Action 1	Action 2	Action 3	Action 4
State 1	0.1	0.85	0.05	0.7
State 2	0.45	0.35	0.2	0.3
State 3	0.01	0.35	0.64	0.25

Table 3.1: State-Action Value Table

In games where the number of actions changes at each time step, the State-Value approach in another method to represent the learning. In this method, each State has a value associated with it. A few next states are linked to this state by means of possible actions. The value of each state is iteratively updated by the Bellman equation. A sample State-Value table is shown in Table 3.2.

State	Value
State i	0.15
State j	0.2
...	...

Table 3.2: State-Value Table

3.2 Exploration and Exploitation

- **Exploration:** Agent chooses a random action to gather more information about the environment and update its understanding.
- **Exploitation:** Agents make the best decision based on current information.
- **Epsilon Greedy Strategy:**To balance exploration and exploitation, one can choose between the two based on a threshold. The epsilon-greedy strategy is a policy that handles the exploration/exploitation trade-off. The agent will choose exploration or exploitation at each time step based on a set value of epsilon. In practice, we generate a random number between 0 and 1. If this number is greater than or equal to ε, then the agent will choose its next action via exploitation, i.e., it will choose the action with the highest Q-value for its current state from the Q-table. Otherwise, we do exploration, i.e., randomly choosing its action and exploring what happens in the environment.

3.3 Neural Network for RL

A neural network consists of several processing nodes fashioned into multiple layers that are typically densely connected. These layers comprise the following:

- **Input Layer:** The number of nodes in the input layer is generally fixed and corresponds to the size of input data, for instance, the number of states in an environment.
- **Hidden Layers:** There are typically one or more hidden layers in neural network architecture. The number of layers and nodes in each layer are hyperparameters of the architecture.
- **Output Layer:** The output layer also has a fixed number of nodes corresponding to the required output, for instance, the number of actions in an environment. The neural network architecture is defined in Fig 3.2.

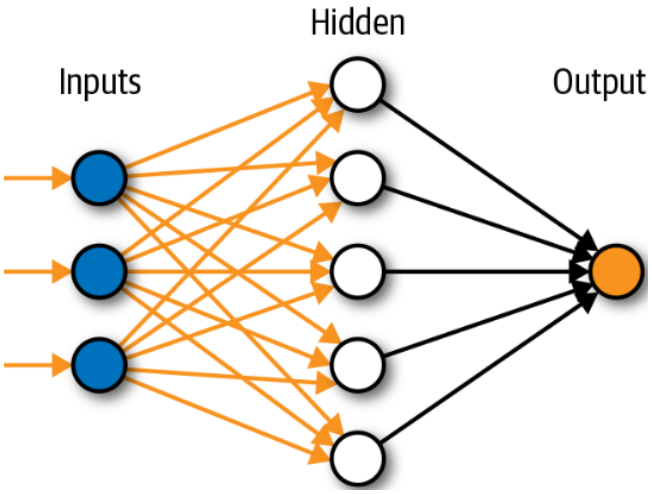


Figure 3.2: A neural network architecture [28].

3.3.1 Activation Function

One of the most important reasons this activation function is necessary is that it provides non-linearity to the output, which is otherwise fairly linear. This non-linearity makes the neural network able to learn complex and real-world patterns. The choice of activation function in a neural network is a matter of optimization, hence it falls into the list of hyperparameters. However, the nature of input data and the output we desire can help us make a good start. In this project, we use the Rectifier Linear Unit (ReLU) as the activation function in the hidden layer and the Linear activation function in the output layer.

3.4 Bellman Equation

In the Q-learning perspective, the Bellman equation serves as a cornerstone in reinforcement learning, expressing the relationship between the value of a state or state-action pair and the expected rewards obtained from that state. The value associated with each state is learned by the agent. The optimal state-action function satisfies the Bellman equation [29] as defined below:

$$Q_{\text{new}}(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma \max_{a'} Q'(s',a') - Q(s,a)]$$

, Where $Q_{\text{new}}(s,a)$ is the updated Q-value for state-action pair (s,a).$Q(s,a)$ is the current Q-value for state-action pair (s,a).α is the learning rate. $R(s,a)$ is the immediate reward of taking action a in state s.γ is the discount rate. $\max_{a'} Q'(s',a')$ represents the maximum possible Q-value in the next state s', considering all possible actions a'

There is another variant of Bellman equation for State Value functions. In this approach, the value of a state is updated based on the value of the previous state and actions. The equation is defined below:

$$\begin{aligned} v_\pi(s) &\cong \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \quad \text{for all } s \in S \end{aligned}$$

where $v_\pi(s)$ is the value function for policy π at state s, $\mathbb{E}_\pi[G_t|S_t = s]$ is the expected return at time step t given that the current state is s, and G_t is the return. γ is the discount factor, and S is the set of all possible states in the environment [30].

3.5 Deep Q-Network (DQN)

Deep Q-Network (DQN) [31] refers to the neural network utilized within the Deep Q-Learning framework to estimate the Q-function, which is used to determine the optimal action to take in a given state. The Q-function represents the expected cumulative reward of taking a certain action in a certain state and following a certain policy. In Q-Learning, the Q-function is updated iteratively as the agent interacts with the environment. During training, the DQN receives experience tuples containing the current state, action, reward, and next state. When the agent interacts with the environment,the generated tuples are stored in the replay buffer. This buffer is then used to train the DQN using an optimizer like Stochastic Gradient Descent or Adam optimizer. The target network is used to predict the maximum Q value of the next state for all actions. This estimate is used by the prediction network to update its weights. After a certain number of iterations, the weights of the prediction network are copied over to the target network. This process stabilizes the training and enables faster convergence. One of the key challenges in implementing the Deep Q-Network (DQN) is that the Q-function is typically nonlinear and can have many local minimums. The DQN algorithm [32] handles environments with many states and actions, as well as learns from high-dimensional inputs such as images or sensor data. Deep Q-Network (DQN) has been applied to a wide range of problems, including game-playing, robotics, and autonomous vehicles. For example, it has been used to train agents who can play games such as Tic-Tac-Toe, Othello, and Chess and also to control robots for tasks such as grasping and navigation.

Chapter 4

Tic-Tac-Toe game using Q-learning

In this chapter, we demonstrate how a simple reinforcement learning algorithm can be set up and also show some results. We play a simple two-player game, Tic-Tac-Toe, with the goal of teaching a Reinforcement Learning agent to play it effectively. The learning agent is based on tabular Q-Learning. We observe how quickly the agent can learn and whether it can achieve an optimal policy that maximizes the cumulative reward. We use Python as the programming language for our project.

The main reason for choosing Tic-Tac-Toe for implementing Q-Learning is its simplicity. Tic-Tac-Toe is a straightforward two-player game with well-defined rules, making it easy to formalize. Additionally, Tic-Tac-Toe has a relatively small state space due to its simplicity, which is advantageous for several reasons: it allows for the use of tables to map state-action pairs, and it ensures that the number of possible games is manageable. Q-learning is selected because it is a fundamental component of temporal difference learning, which is one of the cornerstone techniques in modern reinforcement learning. The tabular version of Q-learning is deemed appropriate for Tic-Tac-Toe due to its simplicity and suitability for the game's requirements. Furthermore, it offers computational advantages over more complex Q-learning methods, making it a pragmatic choice for implementing the Tic-Tac-Toe game.

The project consists of four parts. In part one, we set up the programming environment. In part two, we examine the code for Tic-Tac-Toe and all our Player agents. In part three, we simulate training the agent by playing games against different opponents and saving their policies for three different strategies. In part four, we evaluate and test the performance of the game with three different strategies using the Q-learning algorithm.

4.1 Setting up the Game Environment

This section focuses on the implementation of the learning agent. Tic-Tac-Toe, being a classic two-player game, can be implemented as a game environment class. The implementation of the Tic-Tac-Toe game developed for this study is available in reference [33]. The game environment class can be kept simple, with functions for creating a game board, converting a game state to a hash value,

checking legal moves (available positions), switching player turns, and determining the winner.

Initially, we set up an empty board of size 3x3 and designated two players as Player 1 (P1) and Player 2 (P2). We assume P1 plays 'X' as the first player. Each player's symbol fills the board when they take an action, updating the board state accordingly. For RL training, The board is represented with symbols: '0' for available positions, '1' for Player 1's moves ('X'), and '-1' for Player 2's moves ('O'). The board representation of P1 move is shown in Fig. 4.1 and the other board representation of P2 move is shown in Fig 4.2.

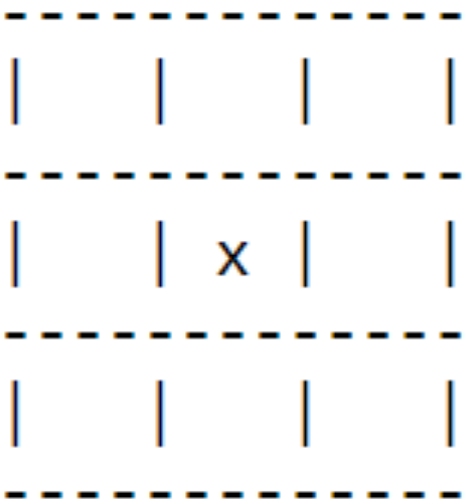


Figure 4.1: Player 1 move.

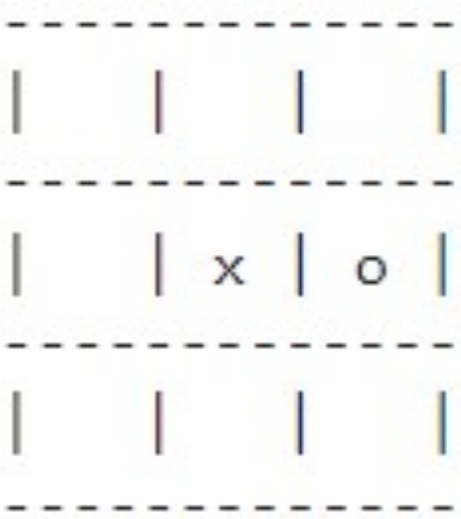


Figure 4.2: Player 2 move.

The hash function hashes the current board state, allowing it to be stored in a dictionary for value lookup. Hash values of game states are used to index data tables for specific player types. Before making a move, we check if there are any legal moves (available positions). If so, the position is updated. To determine the winner after each player's action, we need a function to continuously check if the sum of any row, column, or diagonal equals 3 or -3. If the game ends, rewards are given accordingly: P1 wins if they fill a row, column, or diagonal with their marker ('X'), earning a reward of 1 and P2 gets a reward of -1. Also, P2 wins if they fill a row, column, or diagonal with their marker ('O'), earning a reward of 1 and P1 gets a reward of -1. If there are no available positions left on the board and neither P1 nor P2 has won, the game is declared a tie and a reward of 0 is given to both players. The final board where P1 wins is shown in Fig 4.3.

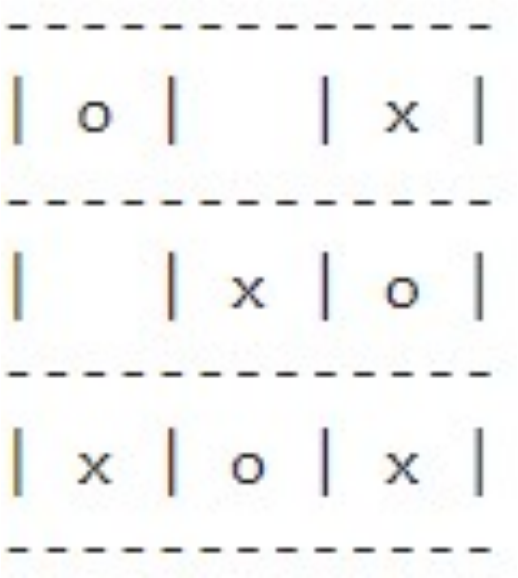


Figure 4.3: Final board game.

4.2 Training the Agent

During the training of the reinforcement learning agent for the Tic-Tac-Toe game using Q-Learning, two players, denoted as P1 and P2, initially play against each other and learn. Subsequently, the learned policy is loaded into the computer, and the agent plays against humans.

The Q-Learning player class is designed to be simple, with functions initialized by hyperparameters. These functions include looking for available positions, choosing actions, updating the board state, adding actions to player states, and determining the end of the game to assign rewards accordingly.

The hyperparameters are initialized as follows:

- **Learning Rate (α):** Controls the rate at which the Q-values are updated during training. Initialized as $\text{lr} = 0.001$.
- **Exploration Rate (ε):** Determines the probability of the agent exploring a random action versus exploiting learned knowledge. Initialized with $\text{start\_exp\_rate} = 0.3$, indicating a high exploration rate at the start.
- **Discount factor (γ):** The discount factor balancing immediate rewards versus future rewards. Initialized as $\text{decay\_gamma} = 0.9$.
- **Decaying epsilon greedy:** The epsilon decay rate is a technique that is used in training deep neural networks, which essentially focuses on setting both a starting and ending exploration rate. The main reason for using this technique is to gradually decrease the start exploration rate over time, thereby gradually increasing the percentage of exploitation over time.
- **Start Exploration Rate:** Initial value of the exploration rate ($\varepsilon = 0.3$).
- **Final Exploration Rate:** Final value of the exploration rate ($\varepsilon = 0.05$).

In the initialization function, a dictionary stores state-value pairs, and estimates are updated for all positions taken by the players during each game. These updates are reflected in a state-value dictionary. The exploration rate may decay over time according to a linear decay schedule, initially encouraging exploration and later exploitation.

For each action to be taken by a player, the agent selects the action based on an epsilon-greedy policy. Here, we set the exploration rate (ε) to 0.3. This means that 70% of the time, the agent will take a greedy action based on current estimations of state values, while 30% of the time, it will take a random action.

After each action, the agent updates its internal state representation and adds the state to its memory. Once the game ends, rewards are calculated based on the outcome:

- If P1 wins, it receives a reward of 1 and P2 gets a reward of -1.
- If P2 wins, it receives a reward of 1 and P1 gets a reward of -1.
- If the game ends in a draw, the reward is 0 for both P1 and P2.

The rewards are propagated backward through the states visited during the game, updating the state value using the Bellman equation. This process of updating the Q-values continues until convergence or a predefined number of iterations.

During training, both P1 and P2 play 50,000 games, using their policies and simultaneously updating their policies. Additionally, we observe three variants as defined below:

- P1 and P2 follow constant epsilon value of 0.3 and train.
- P1 follows decaying epsilon, with a starting exploration rate of 0.3 and an ending exploration rate of 0.05. P2 follows a constant epsilon value of 0.3 and trains.
- Both P1 and P2 follow decaying epsilon and train, with a starting exploration rate of 0.3 and an ending exploration rate of 0.05.

Additionally, we incorporate visualizations to track the training progress, presenting plots that demonstrate the win rates and ties across training batches. The training comprises 50,000 games divided into 50 batches of 1000 games each. The X-axis of the plots corresponds to the number of outer rounds, representing the batches during the reinforcement learning (RL) agent's training. On the Y-axis, we illustrate the probability of the agent in each respective figure. Moreover, we visualize the overall training progress through plots that showcase the win rates and ties throughout the entire training process, as depicted in Fig 4.4, Fig 4.5, and Fig 4.6

In the first variant shown in Fig 4.4, where both players P1 and P2 have constant decay rates, we observe that the probability of P1 starts at 0.5, and then significantly increases, reaching a high probability. For P2, the probability starts at 0.27 and gradually decreases, and is finally convergent. The tie probability for both players starts at 0.16 and gradually decreases before converging.

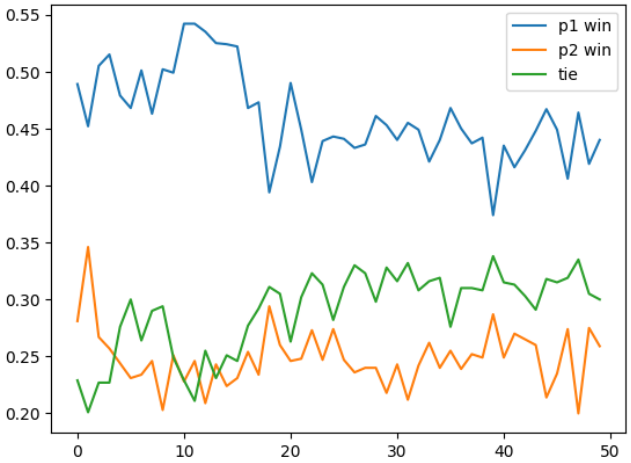


Figure 4.4: Both players P1 and P2 trained with constant epsilon. Win probabilities in each batch.

In the second variant shown in Fig 4.5, P1 has a decaying exploration rate, while P2 uses constant exploration rate. Initially, P1's win rate is 0.57, and the win rate for each batch cumulatively changes, meaning it fluctuates across batches, increasing in some batches and decreasing in others. As for P2, its exploration rate remains constant, with the win rate starting at 0.34 and gradually decreasing with each batch before converging to 0.05.

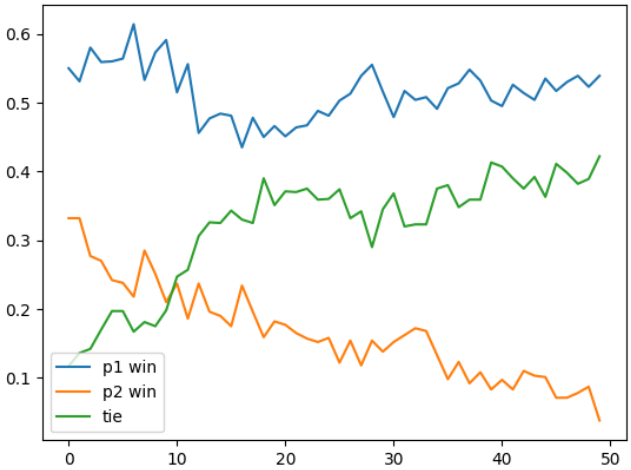


Figure 4.5: P1 trained with decay epsilon and P2 trained with constant epsilon. Win probabilities in each batch.

In the third variant shown in Fig 4.6, where both players' exploration rates decay, the win probability for P1 starts at 0.68 and gradually decreases for each batch before the end of the exploration rate. Similarly, for P2, the win rate starts at 0.2, increases in one batch, and then decreases in the remaining batches before the end of the exploration.

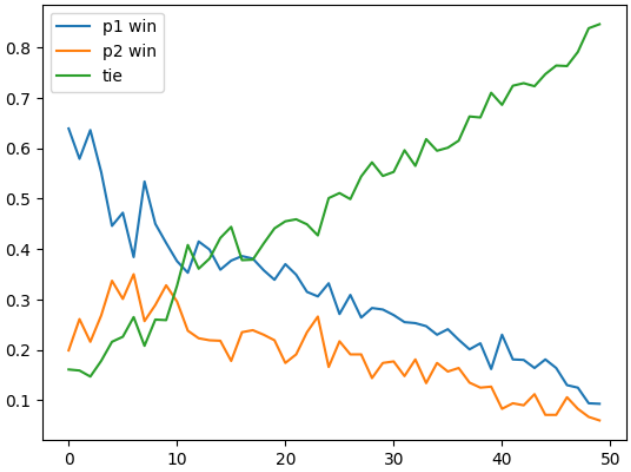


Figure 4.6: Both players P1 and P2 trained with decay epsilon. Win probabilities in each batch.

4.3 Testing the trained agent

After training the Tic-Tac-Toe game using the Q-Learning algorithm, we save the final policy for Player 1 (P1) and Player 2 (P2) for three different variants. Using these saved policies, we conduct a test game with 1000 iterations to evaluate the performance of the three different strategies:

- Two Random Players: Both players P1 and P2, make random moves. This outcome serves as a baseline, to establish the normally expected performance.
- Player 1 (P1) as Computer and Player 2 (P2) as Random.
- Player 1 (P1) as Random and Player 2 (P2) as the Computer: Player 1 makes random moves, while Player 2 follows the learned policy.

For each strategy, we record the number of wins for Player 1, the number of wins for Player 2, and the number of draws. This allows us to assess the effectiveness of the learned policies in different gameplay scenarios.

Player	**Winning Probability (%)**
Random Player (P1)	57.40
Random Player (P2)	43.60

Table 4.1: Both players P1 and P2 are random

Test	Description	P1 win prob.(%)	P2 win prob.(%)	Tie prob.(%)
1	Both players trained with constant epsilon	96.90	3.10	0.00
2	One player trained with decaying epsilon and other player trained with constant epsilon	98.40	0.00	1.60
3	Both players trained with decaying epsilon	99.10	0.00	0.90

Table 4.2: Computer player (P1) plays against a Random player (P2)

Test	Description	P1 win prob.(%)	P2 win prob.(%)	Tie prob.(%)
1	Both players trained with constant epsilon	49.20	46.00	4.80
2	One player trained with decaying epsilon and other player trained with constant epsilon	49.50	46.40	4.10
3	Both players trained with decaying epsilon	52.30	44.10	3.60

Table 4.3: Random player (P1) plays against a Computer player (P2)

4.4 Results

- **Case 1: Both random players P1 and P2 plays against each other**

  From the Table 4.1, where two random players, P1 and P2, engage in gameplay against each other, we assess the performance by considering the win rates of both players. The first player (P1) has a 57% chance of winning, while the second player (P2) has a 43% chance of winning. These outcomes serve as a baseline to establish the expected normal performance.

- **Case 2: Computer player (P1) plays against a Random player (P2)**

  From the Table 4.2, where the P1 is a computer and the P2 is a human, we observe that the average win probability of the computer player increases significantly with each exploration

method, starting from 96.90% to 99.10%, reaching an average of 98%. On the other hand, the averaging win chance of P2 is only 1.7%, which gradually decreases from 3.10% to 0.00% with each exploration method. This performance disparity highlights the effectiveness of strategic decision-making employed by the computer player (P1) against the randomness of P2's moves.

- **Case 3: Random player (P1) plays against a Computer player (P2)**
  From Table 4.3, where a random player (P1) competes against a computer player (P2), the random player (P1) exhibits an average winning probability of 52%, which increases significantly with each training method. In contrast, the computer player (P2) demonstrates an average win probability of 44%, gradually decreasing with each iteration. As second player P2, the learned player consistently outperforms the random player's moves performance observed in Table 4.1. Additionally, in these games, the player who moves first player (P1) holds an advantage, resulting in a higher probability of winning compared to second player P2.

Chapter 5

Tic-Tac-Toe game using Neural Network

5.1 Setting up the Environment

We will discuss the implementation of the Tic-Tac-Toe game using the DQN algorithm. The code developed for this is shared on GitHub at this location [34]. The first step is setting up the environment of the simple board game of Tic-Tac-Toe using neural network function approximation and it is the same as we did in case of using Q-learning. During the training of the RL agent, we utilize a neural network to predict the state value instead of relying on the tabulated dictionary like earlier. We observed how quickly the RL agent can learn using neural network function approximation and whether it can achieve an optimal policy that maximizes the cumulative reward.

5.2 Training the Agent

During the training of RL agents for Tic-Tac-Toe with a function approximation method like DQN, two players, denoted as P1 and P2, initially play against each other and train. Subsequently, the learned policy is loaded into the computer, and the agent plays against humans. The DQN agent class is designed to be simple, with functions initialized by hyperparameters. These functions include looking for available positions, choosing actions, updating the board state, adding actions to player states, and determining the end of the game to assign rewards accordingly.

In the initialization function, a neural network is instantiated to represent the state-value pair instead of a Q-value dictionary. This network learns to predict the value of a state. The exploration rate may decay over time according to an exponential decay.

In a simple board game Tic-Tac-Toe of size 3×3, which comprises a total of 9 cells, the state space size is 3^9, representing all possible combinations of X, O, or empty cells on the board. To predict the game's state value, we utilize a simple feedforward neural network architecture. Here our input layer has 9 nodes in this Tic-Tac-Toe game, which correspond to the number of cells on the board. There is one fully connected hidden layer comprised of 4 nodes. Finally, there is an output layer comprised of 1 node corresponding to the value of the state. We defined the neural network

architecture below in Fig.5.1. This architecture allows the neural network to learn and approximate the state value associated with different states in the Tic-Tac-Toe game.

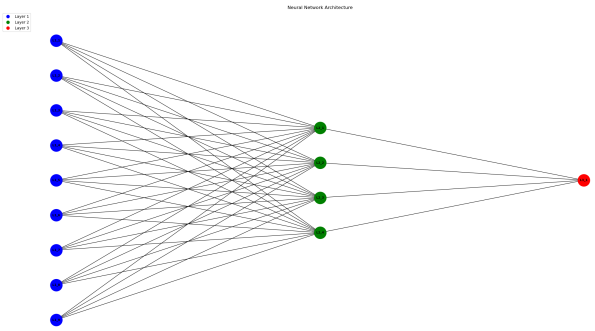


Figure 5.1: Neural network architecture of Tic-Tac-Toe game.

TensorFlow Keras framework is used to define a Sequential model and describe the model summary for the Tic-Tac-Toe game. We defined the summary in below Table 5.1.

Model: "Sequential"		
Layer (type)	**Output Shape**	**Param #**
dense (Dense)	(None, 9)	333
dense (Dense)	(None, 4)	40
dense (Dense)	(None, 1)	5
Total params:	378	
Trainable params:	378	
Non-trainable params:	0	

Table 5.1: Neural network model summary

Each RL agent has this function approximation neural network. Two networks are employed, one for each player, where each network is responsible for predicting the state value for all possible actions. Each network consists of an output layer that provides the state value. The state value is updated using the Bellman equation, which ensures that the network learns to accurately predict the expected future rewards for each action in a given state. At a given state S on the board, the potential states S' the game can transition into are limited. The number of actions available at state S is equivalent to the possible future states, as each action leads to one of these states.

During training, the agent sees 2,000 games, from which it computes the value of each state. The agent receives experience tuples containing the current state, action taken, reward, and the next state. These tuples are stored in a replay buffer, and batches of samples from the experience buffer are used to update the state value in the current state. We utilize a decaying epsilon greedy for exploration-exploitation that focuses on an exponential decay schedule to decrease the exploration rate. After

every training batch, the replay buffer is emptied, and the process is repeated. After each batch, the outcomes are recorded, and the neural network models for both players are periodically saved.

By training for 2,000 games, the DQN agent learns to make better decisions in the game of Tic Tac Toe, gradually improving its performance through reinforcement learning. This process continues until convergence. Once the game ends, rewards are calculated based on the outcome:

- If P1 wins, it receives a reward of 1 and P2 gets a reward of -1.
- If P2 wins, it receives a reward of 1 and P1 gets a reward of -1.
- If the game ends in a draw, the reward is 0 of both P1 and P2.

The backpropagation technique computes the rewards and updates the state values accordingly using the Bellman equation via the neural network available [35]. The agent trains the neural network on the replay buffer data, iterating over batches of transitions and updating the model parameters to minimize the mean squared error between predicted and target state values. During training, both P1 and P2 play according to their policy for 2,000 iterations, while also improving the policy simultaneously. We train three variants as defined below:

- P1 and P2 follow constant epsilon of 0.3 and train.
- P1 follows decaying epsilon, with a starting exploration rate of 0.3 and an ending exploration rate of 0.05. P2 follows constant epsilon of 0.3 and trains.
- Both P1 and P2 follow decaying epsilon and train, with a starting exploration rate of 0.3 and an ending exploration rate of 0.05.

The training progress is visualized using plots that display the win rates and ties over training batches. The total number of game plays by both players is 2000, which is divided into 20 batches of 100 games each. Firstly, we define the X-axis to represent the number of batches, and the Y-axis represents the probability. The ranges of the X-axis are from 0 to 20, and the Y-axis ranges from 0 to 1. Furthermore, we illustrate the comprehensive training progress by plotting the win rates and ties across each training batch. These plots are presented in Figures 5.2, 5.3, and 5.4.

In the first variant depicted in Fig. 5.2, where both players P1 and P2 have constant decay rates, we observe that the probability of P1 starts at 0.6 and then significantly increases, reaching a high probability. For P2, the probability starts at 0.1 and significantly increases to approximately 0.5 before converging. The tie probability for both players starts at 0.3 and then gradually decreases and finally converges.

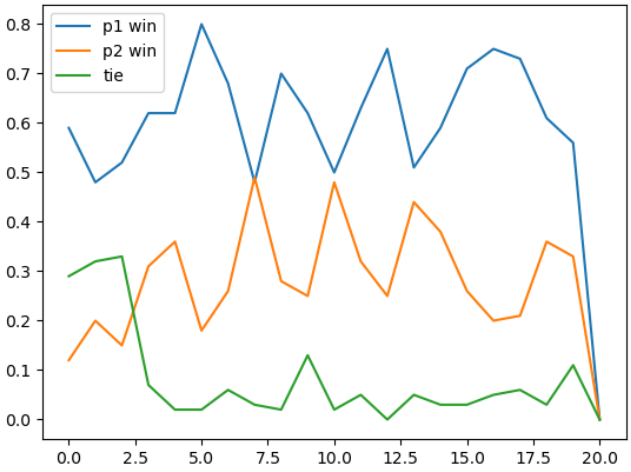


Figure 5.2: Both players P1 and P2 trained with constant epsilon. Win probabilities in each batch.

In the second variant shown in Fig. 5.3, where P1 has a decaying exploration rate while P2 has a constant exploration rate. For P1, the probability starts at 0.78 and gradually increases before converging. As for P2, it starts with a probability of 0.2 and experiences a slight increase, followed by a decrease before converging. In each batch, the exploration rate for P1 decreases and converges to 0.05 at the end of all rounds.

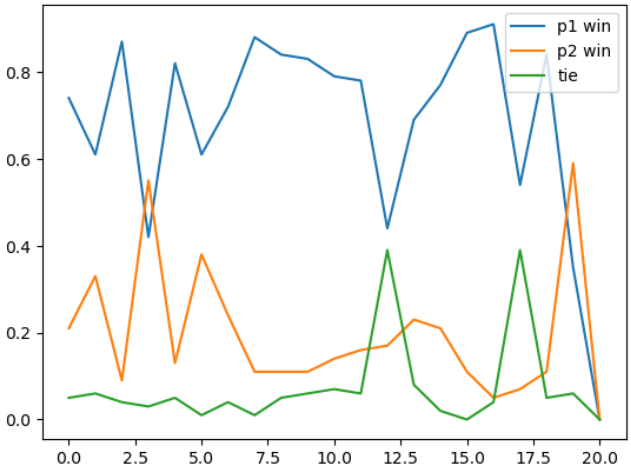


Figure 5.3: P1 trained with decay epsilon and P2 trained with constant epsilon. Win probability in each training batch.

In the third variant depicted in Fig. 5.4, both P1 and P2 follow decay schedules. The initial exploration rate is 0.3, which converges for both players after completing all batches to an exploration rate of 0.05. Win probabilities of P1 and P2 are correlated, with P1's probability increasing while P2's decreases symmetrically.

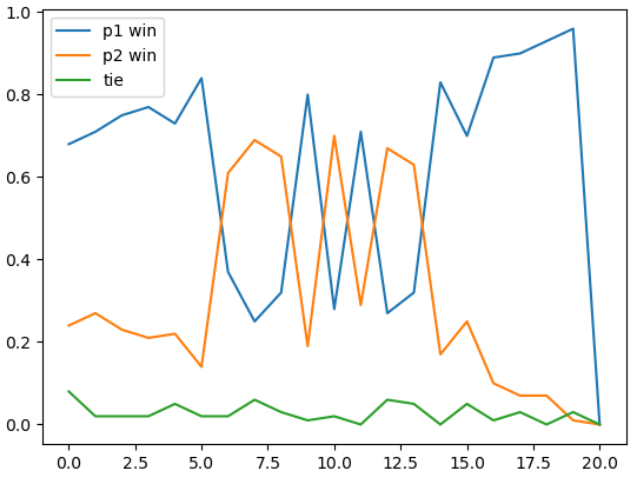


Figure 5.4: Both players P1 and P2 trained with decay epsilon. Win probabilities in each batch.

5.3 Testing trained agent

After training the Tic-Tac-Toe game using the Deep Q-Network (DQN) algorithm, we save the final policy for Player 1 (P1) and Player 2 (P2) for three different variants. Using these saved policies, we conduct a test game with 100 iterations to evaluate the performance of the three different strategies:

- Two Random Players: Both players make random moves. This outcome serves as a baseline, to establish the normally expected performance. The win rate of the first random player (P1) is 57.40%, and the win rate of the second random player (P2) is 43.60%.
- P1 is computer player and P2 is random player.
- Player 1 (P1) as Random and Player 2 (P2) as the Computer: Player 1 makes random moves, while Player 2 follows the learned policy.

For each strategy, we record the number of wins for P1, the number of wins for P2, and the number of draws. This approach enables us to evaluate the effectiveness of the learned policies across various gameplay cases.

Test	Description	P1 win prob.(%)	P2 win prob.(%)	Tie Prob.(%)
1	Both players P1 and P2 trained with constant epsilon	80.00%	10.30%	9.70%
2	P1 trained with decaying epsion and P2 trained with constant epsilon	80.20%	12.40%	7.40%
3	Both players P1 and P2 trained with decaying epsilon	76.20%	17.50%	6.30%

Table 5.2: Computer player (P1) plays against a Random player (P2)

Test	Description	P1 win prob.(%)	P2 win prob.(%)	Tie prob.(%)
1	Both players trained with constant epsilon	36.80%	52.30%	10.90%
2	One player P1 trained with decaying epsilon and other player P2 trained with constant epsilon	40.10%	52.20%	7.70%
3	Both players P1 and P2 trained with decaying epsilon	38.80%	51.80%	9.40%

Table 5.3: Random player (P1) plays against a Computer player (P2)

5.4 Results

- **Case1: Computer player (P1) plays against a Random player (P2)**

  In this case, where the first player (P1) uses trained policy while the second player (P2) plays randomly, we observed that the average win probability of the computer player (P1) is 78.47%, whereas the random player (P2) has an average winning chance of only 13.90%. This performance is average across the three strategies shown in Table 5.2. These results indicate that the trained computer player has a significantly higher chance of winning.

- **Case2: Random player (P1) plays against a Computer player (P2)**

  In the second case presented in Table 5.3, where a random player (P1) competes against a computer player (P2), the random player (P1) achieves an average winning probability of 39%,

while the computer player (P2) has a significantly higher average win probability compared to P1.

In all the scenarios described above, each trained player has demonstrated a higher probability of winning against the random player in the first or second position. For example, when the computer player acted as the first player and a random player took the second position, the computer player exhibited the highest winning probability. Similarly, when the random player played first and a computer player second, the advantage shifted to the computer player because the RL-trained player, whether acting as Player 1 or Player 2, performed better than the random player.

Chapter 6

Othello game using Deep Q-Network

We will discuss a bigger game, Othello. A schematic of the Othello board on a 6x6 grid with 36 squares is shown in Fig.6.1. Each square can be empty or occupied by either player and hence has 3 possible outcomes, resulting in a total of 3^{36} possible states. Othello is a two-player game where players are denoted by the symbols Black ('B') and White ('W'). The starting position of the board is two black coins and two white coins arranged in a 2×2 square at the center of the board. The goal of Othello is to win by having more coins of their color (either black or white) on the board than their opponent. The state representation involves the current board configuration, with actions consisting of legal moves to place color coins. Agents receive rewards based on game outcomes, with winning yielding positive rewards and losing resulting in negatives. Agents learn through gameplay iterations, adjusting strategies based on outcomes and updating their knowledge using the Bellman equation.

The game of Othello is indeed more complex than Tic-Tac-Toe yet simpler than Chess, positioning itself between the two in terms of complexity and computational expense. Its larger state space compared to Tic-Tac-Toe demands considerable memory resources, making it challenging to handle with traditional methods. To address this issue, a neural network-based function approximation is employed, allowing for efficient handling of the extensive state space. The primary motivation behind studying reinforcement learning with simpler games is to reduce the computational power required for training. With Othello's vast state space of approximately 3^{36} entries, it surpasses the commonly available machine memory capacity of necessitating innovative approaches for decision-making processes.

One way to handle large state-action spaces is by using a method called function approximation, like a neural network. Instead of using the Q value dictionary for every single state-value pair, we train a function to approximate the value of the state. This helps save a lot of storage space, making it possible to store the state value for large number of states. The usage of function approximation for state value has been demonstrated for Tic-Tac-Toe in the Chapter 5.

The project comprises four parts, similar to the approach used in the Tic-Tac-Toe section. We train the two agents using 3,000 games and the Epsilon decay approach. Over time, agents improve their performance by learning from experience, ultimately becoming proficient Othello players.

6.1 Setting up the Environment

Othello belongs to the family of two-player board games that were derived from Chess. It consists of a 6 x 6 board with black and white coins. Player 1 plays the black color coin denoted as 'B' and player 2 plays the white color coin 'W'. The implementation of the Othello game code developed for this study is available here [36]. In the board game, we defined the symbol of column and row, each column from left to right with an alphabet starting with 'A'. Each row is numbered from top to bottom starting with 1. This is the notation that shall be followed throughout the rest of the chapter. Each player is associated with a coin color, either white or black. The board has 36 vacant cells that can be occupied by either player. Also game defines the eight possible directions to explore when searching for valid moves- up, down, right, left, and the four diagonals. White owned C4 and D3 denoted the symbols as 'W', while black own C3 and D4 denoted the symbols as 'B'. Black always moves first. The starting position of the board is shown in Fig.6.1.

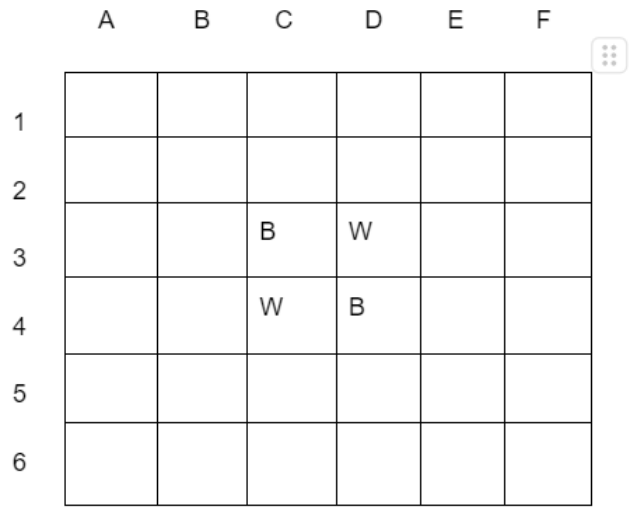


Figure 6.1: Initial Othello Board.

The game environment class can be kept simple, with functions for creating a game board, converting a game state to a hash value, checking legal moves, switching player turns, executing moves, and finally determining the winner outcome.

The get legal moves function is responsible for generating legal moves for a given player's color. It iterates over each cell on the board, checking if it contains a coin of the specified color. For each such cell, determine all possible legal moves from that position in eight possible directions.

The execute move function handles the execution of a move on the game board. It takes as input the coordinates of the move and the player's color. The method first checks if the move is valid to ensure that flipping the opponent's coin is possible in at least one direction. If the move is valid, it flips the opponent's coin and updates the board state accordingly.

The game progresses as each player makes moves. A move is made by placing a coin in an empty square. When a player does so, all of the opponent's coins bracketed between the newly placed coin and another coin of the same color get flipped into the color of the newly played coin. A move is valid only if it flips at least one of the opponents' coins. When such a move does not exist, the player skips a turn, and the opponent makes a move.

For example, we consider the position in the below figure with the black player to move. Its legal moves, which are marked with a dot, are B4, C2, C5, E1, E3, and E5. If the black player plays move B4, the white coins on C4 and D4 will be turned over and become black. The board is shown below in Fig.6.2 and Fig.6.3.

	A	B	C	D	E	F
1				W	*	
2			*	W		
3			B	W	*	
4		*	W	W	B	
5			*		*	B
6						

Figure 6.2: Othello game: black to move.

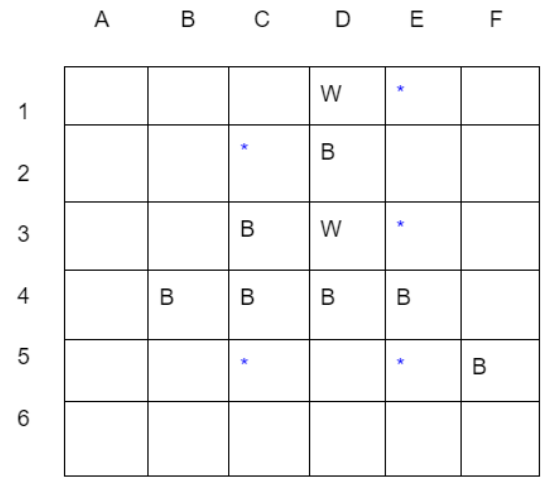


Figure 6.3: Othello game: position after move B4.

The game-over function checks when at least one of the following conditions is true: 1) The board is full, with no empty spaces. 2) The board has coins of only one color. 3) When no player has a valid move.

The get reward function calculates the reward for the current state of the game. If one player has more pieces than the other, they are rewarded accordingly. If both players have an equal number of pieces, the game is considered a draw, and a neutral reward is returned.

6.2 Othello game Strategies using RL

In Othello, the game's state space of 36 squares can be divided into three distinct phases: During the opening game, Player 1 (black) starts with four feasible positions: D2, C5, B4, and E3. The maximum state value among those feasible states is at position B4. It's important to note that symmetric positions have identical Q values. Hence, positions C5 and E3 are considered identical with similar values. During the first ten moves (moves 1-10), both players adhere to strategies determined by the best policies learned by the agent.

In the middle game (moves 11-21), the agent makes the next 10 moves and identifies better positions. The objective is to strategically position colored coins on the board to convert them into stable coins during the end game, coins that cannot be flipped back by the opponent's color. There are two primary middle-game strategies: the positional strategy and the mobility strategy [37].

The positional strategy emphasizes the significance of specific color coins placed on the board, prioritizing corners and edges due to their inherent value. Obtaining a corner coin early on or during the middle game is advantageous as it often leads to acquiring numerous stable coins. Players employing the positional strategy seek to maximize their valuable coins while minimizing those of the opponent.

Also, when considering mobility as a fundamental strategy, the following rules of thumb for openings in Othello emerge: Aim to have fewer pieces than your opponent. Try to control the center of the board, especially the four squares in the middle, early in the game.

Finally, in the end game (moves 22-32), with only a few moves remaining, both players aim for optimal positions, focusing on maximizing their own color coins while minimizing those of the opponent. Obtaining corner coins in the beginning of the game, or in the middle game, usually means that it will be possible to use those corner coins to get many more stable color coins.

The state values assigned by the trained agent represent the likelihood of each position being strategically advantageous. These values play a crucial role in guiding the players' decisions throughout the game. Typically, corners are considered good, while squares adjacent to corners are less desirable. For instance, corners may hold greater significance during the opening and early mid-game phases compared to the endgame.

6.3 Training the Agent

During the training of RL agents for Othello on a 6x6 board using function approximation methods like DQN, two players, designated as P1 and P2, engage in initial gameplay against each other to train.

The DQN agent class is kept simple, with functions initialized via hyperparameters. These functions involve searching for legal moves, selecting actions, updating the board state, adding actions to player states, and determining the end of the game to assign rewards accordingly.

The hyperparameters are initialized as follows:

- **Learning Rate (α):** Controls the rate at which the state values are updated during training. Initialized as $\text{lr} = 0.001$.
- **Exploration Rate (ε):** Determines the probability of the agent exploring a random action versus exploiting learned knowledge. Initialized with $\text{start\_exp\_rate} = 0.2$, indicating a high explo-

ration rate at the start.

- **Discount factor (γ):** The discount factor balancing immediate rewards versus future rewards. Initialized as decay_gamma $= 0.9$.
- **exponential decay exploration rate :** The exponential decay exploration rate is a powerful technique used for training deep neural networks that focuses on starting and ending exploration rates. By gradually decreasing the starting exploration rate over time the rate converges to the ending exploration rate.
- **Start Exploration Rate:** Initial value of the exploration rate ($\varepsilon = 0.2$).
- **Final Exploration Rate:** Final value of the exploration rate ($\varepsilon = 0.02$).
- **Decay rate:** The decay rate refers to the speed at which the influence or importance of past experiences diminishes over time. (decay_rate $= 0.005$).

In the initialization function, a state value neural network is created, which estimates the state value using neural network function approximation updated for all positions taken by the players during each game. These updates are reflected in a state-value neural network.

The game of Othello contains a 6×6 board size with a total of 36 squares. The state space size is 3^{36}, representing all possible combinations of black (B) coins, white (W) coins, or empty cells on the board. To predict the state value, we utilize a simple feedforward neural network architecture.

Here our input layer has 36 nodes for the Othello game, which correspond to the number of cells in our environment. It receives as input the encoded state-input vector. There are three fully connected hidden layers, comprising of 36 nodes for the first hidden layer, 18 nodes for the second hidden layer, and 9 nodes for the third hidden layer. Finally, there is an output layer of 1 node corresponding to the state value. This neural network architecture is shown in figure 6.4. This architecture allows the neural network to learn and approximate the state value associated with different states and actions in the board game Othello.

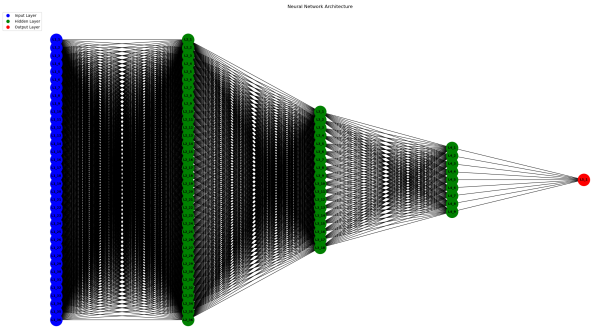


Figure 6.4: Neural Network architecture for Othello Game.

The TensorFlow Keras framework is used to define a Sequential model and describe the model summary for the Othello game. This summary is shown in Table 6.1.

Model: "Sequential"		
Layer (type)	**Output Shape**	**Param #**
dense (Dense)	(None, 36)	1332
dense (Dense)	(None, 18)	666
dense (Dense)	(None, 9)	171
dense (Dense)	(None, 1)	10
Total params:	2179	
Trainable params:	2179	
Non-trainable params:	0	

Table 6.1: Neural network model summary

Each RL agent has this function approximation neural network. Two networks are employed, one for each player, where each network is responsible for predicting the state value for all possible actions. Each network consists of an output layer that provides the state value.

At a given state (S) on the board, there are only a few possible states (S') into which the game can proceed. The number of actions possible in state S is equal to the number of possible future states, as each action takes the game into one of these states. The agent's action selection technique involves the game environment and selecting actions either randomly with a probability determined by the exploration rate or greedily based on the predicted state values using neural network function approximation. The action selection function implements this exploration-exploitation strategy. After each action, the agent stores the current state and chosen action in the replay buffer.

Here, we set the exploration rate ε to 0.2, which means that 80% of the time, the agent will take a greedy action based on current estimations of state values, while 20% of the time, it will take a random action.

During training, both players P1 and P2 trained with 1,000 games and 3,000 games, receiving experience tuples containing the current state, action taken, reward, and the next state. These tuples are stored in a replay buffer, and batches of samples from the buffer are used to predict the state value for all actions in the current state. An exponential decay schedule is utilized to decrease the exploration rate of both players, with the exploration rate being evaluated using the starting exploration rate and end exploration rate.

The training process involves dividing 1000 games into 100 batches, each containing 10 games, to constrain the replay buffer size. Similarly, for 3000 games, the training is partitioned into 300 batches, each comprising 10 games. Subsequently, the replay buffer is sampled, and the state value networks are trained using this data. After each batch, the replay buffer is cleared, and the process repeats. Following each round, the outcomes are recorded, and the neural network models for both players are periodically saved. Finally, we compare the performance of these two iterations, one with 1000 games and the other with 3000 games.

The DQN agent learns to make better decisions in the larger state space of the Othello game. This iterative process continues, and rewards can be calculated for both players P1 and P2 based on the outcomes, as discussed in the earlier chapter. The agent trains the neural network on the replay buffer data, iterating over batches of transitions and updating the model parameters to minimize the mean squared error between predicted and target state values. Through iterative gameplay and neural network training, the DQN agent learns to make better decisions in the game of Othello, gradually improving its performance through reinforcement learning. This process continues until convergence.

The backpropagation technique computes the rewards and updates the state values accordingly using the Bellman equation. The state value function in reinforcement learning represents how good a particular state is under a specific policy. The updated state value is defined using the Bellman equation.

The training progress is depicted through plots showcasing win rates and ties across training batches of 1000 games. For the 1000 games scenario, the data is divided into 100 batches, each containing 10 games. The X-axis denotes the number of batches through which the RL agent learned during training, while the Y-axis represents the probability, ranging from 0 to 1. The visualization of the overall training process is shown in Fig. 6.5.

The first variant of 1000 games is visualized as shown in Fig. 6.5, where both players, P1 and P2, maintain constant decay rates. The probability of success for P1 initially starts at 0.4 and experiences a notable surge, eventually reaching a considerably high probability. Conversely, for P2, the probability begins at 0.6 and steadily ascends to approximately 0.8 before stabilizing. Regarding tie probability, it starts at 0.15 for both players and gradually diminishes before stabilizing.

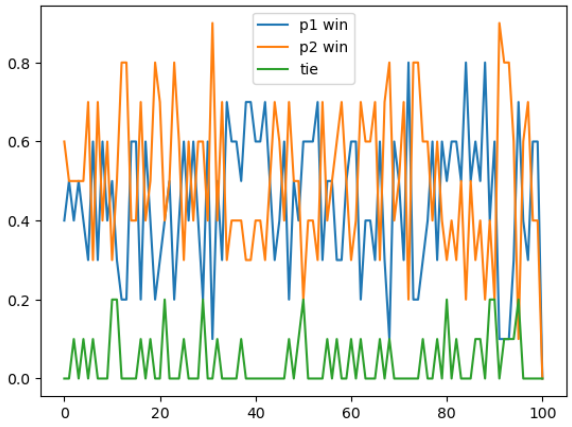


Figure 6.5: Both players P1 and P2 trained with constant epsilon. Win probabilities in each batch.

When both players, P1 and P2, are trained with a constant epsilon, we visualize the cumulative win rates of both players and ties. In this visualization, the total number of games is 500 which is divided by 50 batches each of 10 games. The X-axis represents the batches of the games, while the Y-axis represents the probability ranging from 0 to 1. The visualization is shown in Fig. 6.6. In the initial batch, P1's cumulative win probability stands at 0.2. Over subsequent batches, such as by batch 14, this probability climbs to 0.51 before experiencing a slight decrease and stabilizing at 0.4. Similarly, P2's cumulative win rate begins at 0.8 in the first batch. After several batches, like by batch 14, it decreases and then gradually rises again, eventually stabilizing at 0.5. Additionally, the cumulative tie probability for both players starts at 0 in the first batch but increases to 0.1 by, say, batch 4. This is followed by a minor decrease, ultimately smoothing out at 0.007.

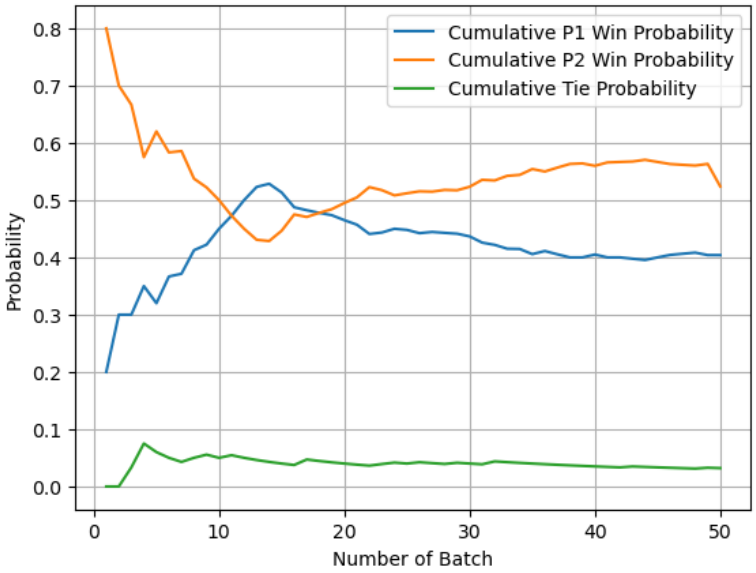


Figure 6.6: Cumulative probability where both players P1 and P2 trained with constant epsilon

6.4 Testing the trained agent

After training the Othello game using the Deep Q-Network (DQN) algorithm, we save the best policy for Player 1 (P1) and Player 2 (P2) for three different variants.

Using the saved policies, we conduct a test game with 100 iterations to evaluate the performance of the three different strategies:

- Two random players: Both players P1 and P2, make random moves. This outcome serves as a baseline to establish the normally expected performance.
- P1 as the Computer and P2 as Random: P1 follows the learned policy, while P2 makes random moves.
- P1 as Random and P2 as the Computer: P1 makes random moves, while P2 follows the learned policy.

For each strategy, we record the number of wins for Player 1, the number of wins for Player 2, and the number of draws. This approach enables us to assess the effectiveness of the learned policies in various gameplay scenarios. The results for different scenarios are presented in Tables 6.2 to 6.6

Player	Winning Probability (%)
Random Player (P1)	43.00
Random Player (P2)	52.00

Table 6.2: Both players P1 and P2 are random

Test	Description	P1 win prob.(%)	P2 win prob.(%)	Tie prob.(%)
1	Both players trained with constant epsilon	19.00%	79.00%	2.00%
2	One player P1 trained with decaying epsilon and other player P2 trained with constant epsilon	25.00%	72.00%	3.00%
3	Both players P1 and P2 trained with decaying epsilon	25.00%	73.00%	2.00%

Table 6.3: First player as Random and Second player as Computer for 1000 games

Test	Description	P1 win prob.(%)	P2 win prob.(%)	Tie prob.(%)
1	Both players trained with constant epsilon	53.00%	40.00%	4.00%
2	One player P1 trained with decaying epsilon and other player P2 trained with constant epsilon	73.00%	24.00%	3.00%
3	Both players P1 and P2 trained with decaying epsilon	68.00%	29.00%	3.00%

Table 6.4: First player as Computer and Second player as Random for 1000 games

Test	Description	P1 win prob.(%)	P2 win prob.(%)	Tie prob.(%)
1	Both players trained with decaying epsilon	23.00%	76.00%	1.00%

Table 6.5: First player as Random and Second player as Computer for 3000 games

Test	Description	P1 win prob.(%)	P2 win prob.(%)	Tie prob.(%)
1	Both players trained with decaying epsilon	72.00%	23.00%	5.00%

Table 6.6: First player as Random and Second player as Computer for 3000 games

6.5 Results

- **Case 1: Both random players P1 and P2 play against each other**

  When two random players, P1 and P2, engage in gameplay against each other, we assess the performance by considering the win rates of both players. From Table 6.2, we observe that player 1 (P1) has a 43% chance of winning, while player 2 (P2) has a 52% chance of winning. These outcomes serve as a baseline to establish the expected normal performance.

- **Case 2: Computer player (P1) plays against a Random player (P2)**

  In an Othello game where player 1 (P1) uses learned policy while the player 2 (P2) plays randomly, we observe that the average win probability of the computer player increases significantly with each iteration, whereas the random player (P2) struggles to maintain a win rate, averaging only 26%, which gradually decreases with each iteration. This performance occurs when both players train with a decaying epsilon of 3000 games. This disparity in performance underscores the efficacy of strategic decision-making employed by the computer player (P1) against the randomness of P2's moves. The results are shown in Table 6.7, and these results are plotted in Fig 6.7.

Test	Iterations	Player 1 Win Rate (%)	Player 2 Win Rate (%)
1	200	60	34
2	1000	68	29
3	3000	72	23

Table 6.7: P1 as computer and P2 as random

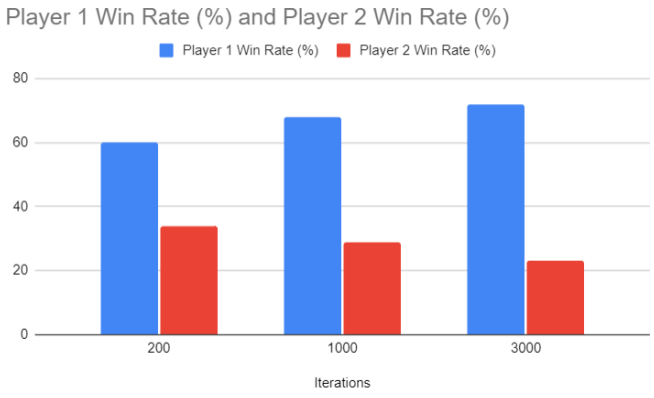


Figure 6.7: Win rate of P1 and P2.

- **Case 3: Random player (P1) plays against a Computer player (P2)**

  In this scenario, where a random player (P1) plays against a computer player (P2), the random

player (P1) has an average winning probability of 23%, which gradually decreases with each given iteration. In contrast, the computer player (P2) demonstrates an average win probability of 74%, which significantly increases with each iteration and is much higher than the random player. This performance occurs when both players train with a decaying epsilon of 3000 games. The results are shown in Table 6.8 and these results are plotted in Fig. 6.8.

Test	Iterations	Player 1 Win Rate (%)	Player 2 Win Rate (%)
1	200	24	72
2	1000	25	73
3	3000	23	76

Table 6.8: P1 as Random and P2 as Computer

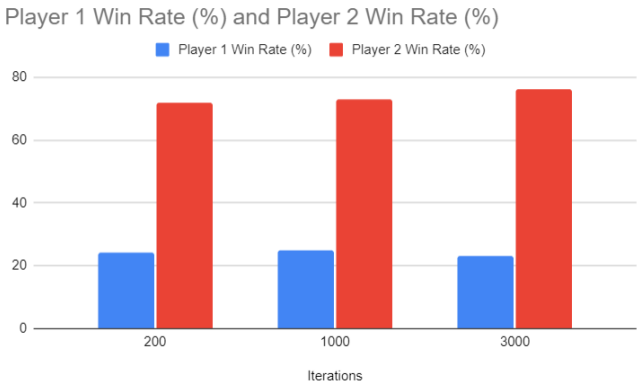


Figure 6.8: Win rate of P1 and P2.

Also, we have observed that in all the above cases, the RL trained players, either player 1 (P1) or player 2 (P2), have an advantage compared to untrained/random players, and the probability of winning is higher compared to the baseline for P1 and P2. These findings indicate that Reinforcement Learning helps that players learn strategies that improve performance.

Chapter 7

Conclusions

Two-player games are useful in various industries, such as online or digital advertising and e-commerce platforms, etc. Here, two players are involved: companies and users/customers. Companies can leverage these games to enhance user understanding, enabling them to make more accurate predictions and effectively target offers.

In this work, we created an RL agent to learn the game of Tic-Tac-Toe using the Q-learning algorithm. The agent quickly learned optimal strategies across various scenarios. We compared three different strategies in a series of games. The RL agent accumulates game knowledge and stores the best policies for evaluation. In this simple board game, we demonstrate that RL trained players perform better than random players. For both first and second player, the RL agent's winning probabilities are much higher than those of random players. We then expanded our approach to Tic-Tac-Toe by implementing function approximation, particularly utilizing neural networks within the TensorFlow Keras framework. Evaluating the performance of this approach across multiple games allowed us to assess the effectiveness of the three different strategies mentioned earlier. Furthermore, we addressed the complexity of Othello, a larger game with a state space surpassing typical machine memory capacities. Handling these larger spaces involved employing function approximation techniques such as Deep Q-Networks (DQN), a neural network-based approach. The RL agent participated in numerous games, enabling us to compare strategies and evaluate their performance. We observed that the learned player, whether P1 or P2, had a higher chance of winning because reinforcement learning assists players in learning strategies that enhance their performance.

7.1 Future Work

In our future work, we will apply RL to even larger games, such as Chess, with an 8x8 board and a state space of 10^{44}. Chess presents significant challenges in terms of computational complexity. We will explore deep reinforcement learning techniques like SARSA and PPO etc. To address the complexities of these board games, thereby advancing our understanding and capabilities in strategic decision-making environments. During the training process of RL agents, we will leverage useful techniques such as Monte Carlo Tree Search (MCTS) and human-expert knowledge to enhance the

agent's performance. Given that Chess is inherently adversarial, with two players competing against each other, we will investigate adversarial training methods, where RL agents train against each other or against human players, to discover novel strategies and improve existing ones. Moreover, evaluating the performance of RL agents in Chess will require sophisticated metrics beyond simple win-loss ratios.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.

[2] "Epsilon-greedy exploration - papers with code," https://paperswithcode.com/method/epsilon-greedy-exploration.

[3] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *2020 Reinforcement Learning Book from The MIT Press*, 2020, retrieved September 23, 2021, from http://incompleteideas.net/book/RLbook2020.pdf.

[4] "Reinforcement learning: Playing tic-tac-toe," https://www.researchgate.net/publication/369096697_Reinforcement_Learning_Playing_Tic-Tac-Toe.

[5] "Q-learning introduction," https://rdcu.be/dGl4P.

[6] "Deep reinforcement learning: Unleashing the power of ai in decision-making," https://doi.org/10.60087/jaigs.v1i1.36.

[7] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, and I. Osband, "Deep Q-learning from demonstrations," in *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*. Menlo Park, Calif.: AAAI Press, 2018, pp. 3223–3230.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv*, 2013, preprint. [Online]. Available: https://arxiv.org/abs/1312.5602

[9] P. Baum, "Tic-tac-toe," Dec 1975, thesis for: Master of Science; Advisor: Professor Kenneth J. Danhof.

[10] M. Buro, "How machines have learned to play othello," *IEEE Intelligent Systems*, vol. 14, no. 6, pp. 12–1, 1999.

[11] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-learning algorithms: A comprehensive classification and applications," *IEEE*, 2024.

[12] "Reinforcement learning explained visually part 5: Deep q-networks step by step," https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b.

[13] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-70911-1_14

[14] D. O. Hebb, "Donald o. hebb and the organization of behavior," *Molecular Brain*, vol. 13, no. 1, p. 77, 2020. [Online]. Available: https://doi.org/10.1186/s13041-020-00567-8

[15] *Supervised Learning*. Springer, 2010. [Online]. Available: https://doi.org/10.1007/978-1-4419-1428-6_451

[16] "Supervised learning image," https://www.erswf.buzz/products.aspx?cname=types+of+supervised+machine+learning&cid=95.

[17] "Unsupervised learning image," https://www.geeksforgeeks.org/ml-types-learning-part-2/.

[18] F. Olaoye and K. Potter, "Reinforcement learning and its real-world applications," *Machine Learning*, March 2024. [Online]. Available: https://www.researchgate.net/publication/379025393_Reinforcement_Learning_and_its_Real-World_Applications

[19] "Reinforcement learning image," https://www.guru99.com/reinforcement-learning-tutorial.html.

[20] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *arXiv preprint arXiv:2002.00444*, Feb 2020.

[21] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, "How to train your robot with deep reinforcement learning: Lessons we've learned," *Journal of Robotics Research (IJRR)*, February 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2102.02915

[22] M. M. Afsar, T. Crump, and B. Far, "Reinforcement learning based recommender systems: A survey," *arXiv preprint arXiv:2101.06286*, Jan 2021.

[23] C. Yu, J. Liu, and S. Nemati, "Reinforcement learning in healthcare: A survey," *arXiv preprint arXiv:1908.08796*, Aug 2019.

[24] M. van Otterlo and M. A. Wiering, *Reinforcement Learning and Markov Decision Processes*. Berlin, Heidelberg: Springer, Jan 2012.

[25] Reinforcement learning tutorial. [Online]. Available: https://www.guru99.com/reinforcement-learning-tutorial.html

[26] U. Desai, "Mastering q-learning: Hands-on examples and key concepts," https://utsavdesai26.medium.com/mastering-q-learning-hands-on-examples-and-key-concepts-5e610d91a12b, 2020.

[27] Goives. Reinforcement learning explained visually part 4: Q-learning step by step. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-explained-visually-part-4-q-learning-step-by-step-b65efb731d3e

[28] "Neural network architecture," https://stackoverflow.com/questions/65600387/when-to-use-a-neural-network-with-just-one-output-neuron-and-when-with-multiple.

[29] T. Kamihigashi, "Elementary results on solutions to the bellman equation of dynamic programming: Existence, uniqueness, and convergence," *Economic Theory*, vol. 56, no. 2, June 2014.

[30] M. van Otterlo and M. Wiering, *Reinforcement Learning and Markov Decision Processes*. Springer, 2012.

[31] "Concept of DQN in reinforcement learning," https://jonathan-hui.medium.com/rl-dqn-deep-q-network-e207751f7ae4.

[32] M. Wang, "Deep q-learning tutorial: mindqn," *Towards Data Science*, Nov 2020. [Online]. Available: https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc

[33] "Implementation of Tic-Tac-Toe game using Q-learning," https://github.com/jaluatumpa/Reinforcement-learning-TicTacToee-Game.

[34] "Implementation of Tic-Tac-Toe game using DQN algorithm," https://github.com/jaluatumpa/Tic-Tac-Toe-using-DQN-in-RL.

[35] N. Krichen Masmoudi, "Two coupled neural-networks-based solution of the hamilton-jacobi-bellman equation," *Applied Soft Computing*, vol. 11, no. 3, pp. 2946–2963, April 2011.

[36] "Implementation of Othello game using DQN algorithm," https://github.com/jaluatumpa/Othello-using-RL.

[37] D. Billman and D. Shaman, "Strategy knowledge and strategy change in skilled performance: a study of the game othello," *The American Journal of Psychology*, vol. 103, no. 2, pp. 145–166, Summer 1990.