Final Project CM3070

Preliminary Report

Game Playing AI: Kane and Abel

Student: Amit Dubey

Project Template: CM 3020 - Artificial Intelligence

Project Idea Title 1: Kane and Abel: AIs that play games

The idea for the project (based on the template): The game I've chosen is Tic-Tac-Toe where I plan to build two systems. Kane would be the rule based gameplay and Abel would be AI driven.

Tic Tac Toe Basics:

- Grid: 3x3
- Players: X and O
- Goal: Get three of your marks in a horizontal, vertical, or diagonal row.

Motivation for the project: I'm motivated by the desire to explore the contrasts and potential synergies between rule-based AI (e.g. in IBM Deep Blue) and machine learning AI (e.g. AlphaGo and OpenAI's Dota 2 AI) within a simple, accessible game environment

TABLE OF CONTENTS

Section 1: Background

1.1 Introduction to Game-Playing AI

The field of Artificial Intelligence (AI) has long been fascinated with the challenge of creating systems capable of playing games at a high level. Game-playing AI serves as a crucial domain for developing and testing AI methodologies due to the well-defined rules and objectives of games. This area of study not only contributes to advancing AI technology but also offers insights into strategic thinking and decision-making processes that can be applied to broader real-world problems.

Historical Context

One of the earliest and most notable examples of game-playing AI is IBM's Deep Blue, a chess-playing computer that went on to defeat the reigning world chess champion, Garry Kasparov, in 1997. This event marked a significant milestone in AI research, showcasing the potential of AI systems in handling complex tasks. Another landmark achievement was Google DeepMind's AlphaGo, which in 2016 became the first program to defeat a professional human player in the ancient board game Go, a feat previously thought to be a long long time away due to the game's profound complexity.

Evolution of AI Techniques

These accomplishments were made possible through varying AI techniques. Deep Blue relied heavily on brute-force computation and sophisticated handcrafted rules, while AlphaGo leveraged advanced machine learning algorithms, including deep neural networks and reinforcement learning. This evolution from rule-based systems to learning-based approaches signifies a shift in AI research, emphasising the importance of adaptable and self-improving systems.

1.2 Project Focus

This project aims to explore these two distinct methodologies within the context of a simpler yet strategically rich game: Tic Tac Toe. By developing two AI systems - Kane AI and Abel AI - I aim to compare the traditional rule-based approach with modern machine learning techniques. Kane AI will be built using a finite state machine or similar predetermined

strategies, whereas Abel AI will employ statistical machine learning methods to learn the game's strategies through experience.

Objectives

The primary objectives of this project are:

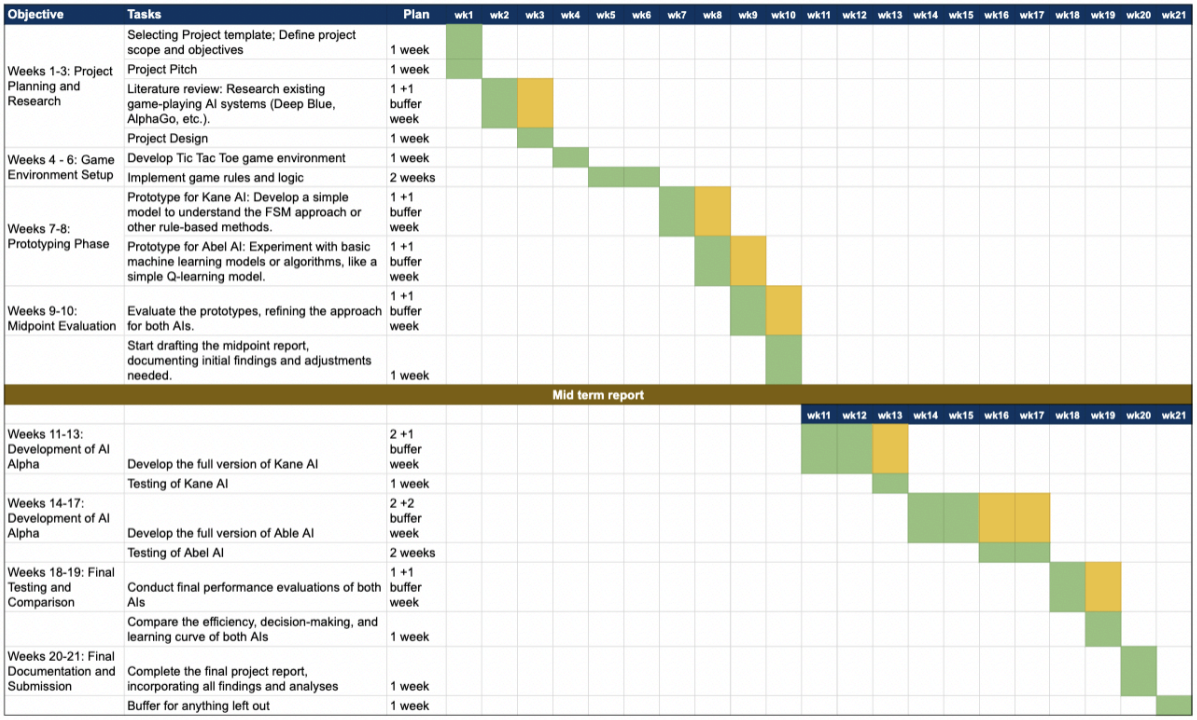
- To develop and integrate two distinct AI players within the Tic Tac Toe game environment.
- To evaluate and compare the performance and learning capabilities of a rule-based AI (Kane AI) and a machine learning-based AI (Abel AI).
- To contribute to the understanding of how different AI approaches can be applied to game-playing scenarios and the implications of these approaches in broader AI applications.

By achieving these objectives, this project aims to not only advance the understanding of game-playing AI but also to shed light on the strengths and limitations of different AI methodologies in a controlled, competitive setting.

Section 2: Planning and Research

2.1 Planning Phase

The planning phase of the project involved establishing a clear and feasible timeline including appropriate buffers, selecting appropriate tools, and outlining the key milestones. The project was divided into several phases: initial research, game environment development, AI development (for both Kane AI and Abel AI), training and testing, and final evaluation. The project has been planned to span over 21 weeks, with a midpoint report to assess progress and make necessary adjustments.



Objective	Tasks	Plan
Weeks 1-3: Project Planning and Research	Selecting Project template; Define project scope and objectives	1 week
	Project Pitch	1 week
	Literature review: Research existing game-playing AI systems (Deep Blue, AlphaGo, etc.).	1 +1 buffer week
	Project Design	1 week
Weeks 4 - 6: Game Environment Setup	Develop Tic Tac Toe game environment	1 week
	Implement game rules and logic	2 weeks
Weeks 7-8: Prototyping Phase	Prototype for Kane AI: Develop a simple model to understand the FSM approach or other rule-based methods.	1 +1 buffer week
	Prototype for Abel AI: Experiment with basic machine learning models or algorithms, like a simple Q-learning model.	1 +1 buffer week
Weeks 9-10: Midpoint Evaluation	Evaluate the prototypes, refining the approach for both AIs.	1 +1 buffer week
	Start drafting the midpoint report, documenting initial findings and adjustments needed.	1 week
Mid term report		
Weeks 11-13: Development of AI Alpha	Develop the full version of Kane AI	2 +1 buffer week
	Testing of Kane AI	1 week
Weeks 14-17: Development of AI Alpha	Develop the full version of Able AI	2 +2 buffer week
	Testing of Abel AI	2 weeks
Weeks 18-19: Final Testing and Comparison	Conduct final performance evaluations of both AIs	1 +1 buffer week
	Compare the efficiency, decision-making, and learning curve of both AIs	1 week
Weeks 20-21: Final Documentation and Submission	Complete the final project report, incorporating all findings and analyses	1 week
	Buffer for anything left out	1 week

2.2 Literature and Research

The research phase focused on studying existing game-playing AI systems to understand their architecture, strengths, and limitations. This informed my approach to developing the two AI players for Tic Tac Toe. Key studies included:

2.2.1 Case Study 1: IBM's Deep Blue

Murray Campbell, A.Joseph Hoane, Feng-hsiung Hsu, Deep Blue, Artificial Intelligence, Volume 134, Issues 1–2, 2002, Pages 57-83, ISSN 0004-3702, https://doi.org/10.1016/S0004-3702(01)00129-1.

Background

Developed by IBM, Deep Blue represents a landmark achievement in the field of Artificial Intelligence and game theory. In 1997, it became the first computer chess-playing system to win both a single game and a chess match against a reigning world champion, Garry Kasparov, under standard chess tournament time controls. This event marked a significant moment in AI history, showcasing the potential of artificial systems to excel in tasks that require deep intellectual strategy and calculation.

Technical Foundations

Deep Blue's design was a testament to the power of specialised hardware and

software tailored for the specific task of playing chess at the highest level. Key technical aspects include:

- Brute Force Search: Deep Blue utilised an advanced brute force algorithm that could evaluate 200 million possible chess positions per second. This allowed the system to explore and analyse a vast number of potential moves and their implications, far beyond human capability.
- Parallel Processing: The system used a custom-built set of parallel processors. Each processor was dedicated to calculating chess moves, enabling simultaneous and rapid evaluation of different game scenarios.
- Heuristic Evaluation Function: Deep Blue employed a sophisticated heuristic algorithm to evaluate the chess board. This function was fine-tuned by chess grandmasters and incorporated a vast array of chess knowledge, including openings, endgames, and positional strategies.
- Database of Games: The system had access to a massive database of historical chess games, allowing it to reference past games and strategies employed by grandmasters.

Limitations

Despite its groundbreaking success, Deep Blue had significant limitations:

- Lack of Generality: Deep Blue was highly specialised for chess. Its algorithms and hardware were custom-built for chess analysis, making the system inflexible and non-transferable to other domains or problems.
- No Learning Capability: Unlike modern AI systems, Deep Blue couldn’t learn from experience. It couldn’t improve its play or adapt its strategy beyond its pre-programmed capabilities.

Influence on Modern AI Projects

Deep Blue's approach contrasts sharply with contemporary AI systems, which emphasise adaptability, learning, and generalisation:

- Learning from Experience: Modern AI systems, including my Kane AI project, aim to learn from interactions within their environment. This learning-based approach allows for continuous improvement and adaptation, a capability that Deep Blue lacked.
- Flexibility and Scalability: Modern AI systems are designed to be more flexible and scalable, often using neural networks that can be trained for different tasks and adapt to new data or environments.

Motivation for my Project:

- Unlike Deep Blue, my AI Abel aims to learn from playing rather than relying on pre-programmed expertise, offering a more flexible approach that can generalise across different games.

2.2.2 Case Study 2: Google's AlphaGo

Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961

Background

AlphaGo, developed by Google DeepMind, marked a significant breakthrough in Artificial Intelligence. In 2016, it became the first computer program to defeat a professional human Go player, Lee Sedol. This achievement was particularly notable due to Go's reputation as a game of profound complexity and subtlety, with a vast number of possible positions and moves that far exceed those in games like Chess.

Technical Foundations

AlphaGo's success was built upon several innovative approaches that combined traditional AI methods with cutting-edge machine learning techniques:

- Deep Neural Networks: AlphaGo utilized deep convolutional neural networks to evaluate Go board positions and predict the most likely moves. These networks were trained on thousands of human amateur and professional Go games, learning patterns and strategies used by human players.
- Monte Carlo Tree Search (MCTS): Alongside neural networks, AlphaGo employed MCTS for decision-making. MCTS is a heuristic search algorithm for decision processes that can provide a strong strategy even with the vast search space presented in Go.
- Reinforcement Learning: AlphaGo was trained using reinforcement learning, where the system played millions of games against itself, learning from its mistakes and improving over time.
- Combination of Supervised and Unsupervised Learning: The initial training of AlphaGo's neural networks was supervised, using a database of human games. It was followed by unsupervised learning through self-play, enabling the program to surpass human-level strategies.

Limitations

While groundbreaking, AlphaGo had its limitations:

- High Computational Demands: AlphaGo's training and gameplay required significant computational resources, including sophisticated GPUs and TPUs, making it resource-intensive.

- Specificity to Go: The architecture and training of AlphaGo were specifically tailored for Go. Its techniques, while powerful, were not directly applicable to problems outside the game or to games with different structures.

Influence on Contemporary AI Projects: Advancing Machine Learning: AlphaGo's use of deep learning and reinforcement learning has inspired a wide range of applications in other complex systems.

Motivation for my Project:

- My project seeks to implement machine learning techniques in a more resource-constrained environment, making it more accessible.
- It also contrasts a rule-based AI with a learning AI in the same game, which AlphaGo does not do.

2.2.3 Case Study 3: OpenAI's Dota 2 AI

Berner, Christopher & Brockman, Greg & Chan, Brooke & Cheung, Vicki & Dębiak, Przemysław & Dennison, Christy & Farhi, David & Fischer, Quirin & Hashme, Shariq & Hesse, Chris & Józefowicz, Rafal & Gray, Scott & Olsson, Catherine & Pachocki, Jakub & Petrov, Michael & Pinto, Henrique & Raiman, Jonathan & Salimans, Tim & Schlatter, Jeremy & Zhang, Susan. (2019). Dota 2 with Large Scale Deep Reinforcement Learning.

Achievement: OpenAI Five demonstrated the ability to compete with and beat professional human players in the complex, multi-agent environment of Dota 2.

Limitations:

- Relied on massive amounts of training data and computing power.
- The approach was designed for real-time strategy games and may not translate to turn-based games without significant modification.

Motivation for my Project:

- I’m exploring how simpler machine learning models can perform in a turn-based setting, which requires different strategies and learning techniques.
- I’m also examining the educational aspect by developing one AI by hand, which Dota 2 AI doesn't address.

2.2.4 Case Study 4: Stockfish (Chess)

- Reference: Official Stockfish Website.
- Achievement: Stockfish is a powerful open-source chess engine widely considered among the best for analysing chess games.
- Limitations:
    - Does not "learn" in the way machine learning systems do. It relies on sophisticated search algorithms and evaluation functions.
    - While open-source and runnable on conventional computers, it doesn't showcase the learning process from a novice level to expertise.
- Motivation for My Project:
    - My project aims to demonstrate a learning curve, with Abel AI starting from scratch and improving over time, offering insights into the machine learning process.
    - It allows direct comparison between traditional AI and ML approaches within the same game environment.

2.2.5 Case Study 5: Monte Carlo Tree Search (MCTS) in Game AI

- Reference: Browne, C. et al. A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games, 2012.
- Achievement: MCTS has been pivotal in advancing AI capabilities in complex board games, significantly contributing to the success of AI in games like Go and Chess.
- Limitations:
    - MCTS can be computationally expensive and may not always converge to the optimal strategy in games with a vast state space.
    - It requires careful tuning and domain-specific adaptations for optimal performance.
- Motivation for My Project:
    - Investigating the applicability of MCTS concepts in a simpler domain like Tic Tac Toe, potentially leading to a hybrid approach in Abel AI.
    - Understanding MCTS helps in contrasting the decision-making process in rule-based systems and learning-based systems.

2.3 Approach for Kane and Abel Game Playing AI

The findings from this research phase laid the foundation for the development of the game environment and the two AI agents, aligning with the project's goals to compare a rule-based AI system with a learning-based AI system in the context of Tic Tac Toe

Kane - Rule based game

Finite State Machine (FSM):

- States can be based on the strategic scenarios in the game. For example:
  - Opening Move
  - Offensive (2 of wer symbols in a row, column, or diagonal)
  - Defensive (2 opponent's symbols in a row, column, or diagonal)
  - Neutral (No immediate threat or winning move)
- Transitions are based on the game board after each move.

Abel - AI based game

Reinforcement Learning (RL) would be an apt approach for Tic Tac Toe.

Q-learning:

- States: Every possible board configuration
- Actions: Any open cell on the board
- Rewards:
  - +1 for a win
  - -1 for a loss
  - 0 for a draw or any other non-terminal move

Let Abel AI play thousands of games against a random policy or itself. Over time, the Q-values will converge, and the agent will have learned the optimal policy for Tic Tac Toe.

Training:

For Abel AI:

- Will play games against a random agent.
- We will try exploration vs. exploitation using an epsilon-greedy policy: Initially, I will let the AI explore more (random moves), but as it learns, let it exploit its knowledge more.
- As it plays, it updates its Q-values based on the rewards it gets.
- Periodically, I will test Abel AI against Kane AI or human players to see its progress.

Evaluation & Comparison:

- Against Humans: We will try volunteers to play against both AIs. Track win, lose, and draw rates.
- Against Each Other: Make Kane AI and Abel AI play against each other. This will give insights into how well the learned strategy of Abel AI compares against the pre-programmed strategy of Kane AI.

Fine-tuning Abel AI :

If Abel AI's performance is unsatisfactory, consider:

- Adjusting the learning rate in Q-learning.
- Changing the exploration vs. exploitation balance.
- Training for more games.

Extensions (That I might give a try):

Advanced Techniques:

- Will try to employ a Deep Q Network (DQN), where a neural network approximates the Q-values. This might be too much for a simple game like Tic Tac Toe but should be a good learning experience.

Adversarial Training:

- Introduce rule variations or handicaps to see how the AIs adapt.

By the end of this project, I should have a clear understanding of traditional game-playing algorithms (as demonstrated by Kane AI) and a modern machine learning approach (Abel AI). The comparison between them in the context of Tic Tac Toe will provide valuable insights into the strengths and weaknesses of each method.

Section 3: Game environment setup

The development of the game environment is a crucial step in my project, providing the foundation upon which both Kane AI and Abel AI interact and learn. For this purpose, I chose Python as my primary programming language due to its simplicity and readability, plus I studied this in my last semester in the AI course. The development process was carried out in a Jupyter Notebook, an ideal environment for such a task given its interactive nature and ability to mix code, output, and documentation.

3.1 Development Process

Creating the Tic Tac Toe Board:

- Started by implementing the basic structure of Tic Tac Toe, a 3x3 grid. This was represented as a list of nine elements, each corresponding to a cell on the board.

- Functions were created for essential operations like displaying the board, checking for wins or draws, and updating the board with players' moves.

Game Mechanics:

- The core game mechanics, such as switching turns between two players and determining the game's outcome (win, lose, or draw), were implemented.
- Additional utility functions were added to check for game-specific conditions like a full board or a winning move.

User Interface:

- Although primarily a backend project, a basic user interface in the Jupyter Notebook was implemented to facilitate manual gameplay and testing. This included functions to accept user input for moves and display the current state of the game.

Testing the Game Environment:

- Rigorous testing was conducted to ensure that the game logic was sound. This included testing edge cases, such as handling invalid inputs and checking the correctness of win/draw conditions.

3.2 Challenges and Solutions

- One of the initial challenges involved ensuring that the game logic accurately recognized all possible win conditions. This was addressed by thorough testing and iterating over the win-checking algorithm.
- Another challenge was managing the state of the game effectively, especially when planning to integrate the AI players. This was resolved by implementing clear state management within the game's architecture.

3.3 Integration with AI Players

- The game environment was designed with placeholders for AI player integration. This structure allows for the seamless introduction of both Kane AI and Abel AI into the game for playing and learning.
- I’m trying for the game environment to be set up to easily toggle between human players and AI players, enabling a flexible testing and demonstration setup.

In Summary

The successful setup of the game environment in Python within a Jupyter Notebook marks a significant milestone in my project. It lays the groundwork for the subsequent phases involving the development, integration, and testing of the AI players. The interactive nature of the Jupyter Notebook will also facilitate future demonstrations and presentations of the project's outcomes.

Section 4: Prototyping and Iteration

The prototyping and iteration phase was crucial for exploring initial concepts for both Kane AI and Abel AI. This phase allowed me to test fundamental strategies and algorithms and to refine the approach before full-scale development.

4.1 Kane AI Prototype

For Kane AI, the prototype focused on implementing a basic rule-based strategy. The objective was to create an AI that could make decisions based on the current state of the board, using a simple set of rules.

Prototype Code for Kane AI:

Prototype for Kane AI

Tic Tac Toe Game Class

In [7]:

```
class TicTacToe:
    def __init__(self):
        self.board = [' ' for _ in range(9)]  # 3x3 board
        self.current_winner = None  # Track the winner!

    def print_board(self):
        for row in [self.board[i*3:(i+1)*3] for i in range(3)]:
            print('| ' + ' | '.join(row) + ' |')

    def available_moves(self):
        return [i for i, x in enumerate(self.board) if x == ' ']

    def empty_cells(self):
        return ' ' in self.board

    def make_move(self, square, letter):
        if self.board[square] == ' ':
            self.board[square] = letter
            if self.check_winner(square, letter):
                self.current_winner = letter
            return True
        return False

    def check_winner(self, square, letter):
        # Check row
        row_ind = square // 3
        row = self.board[row_ind*3:(row_ind+1)*3]
        if all([s == letter for s in row]):
            return True

        # Check column
        col_ind = square % 3
        column = [self.board[col_ind + i*3] for i in range(3)]
        if all([s == letter for s in column]):
            return True

        # Check diagonals
        if square % 2 == 0:
            diagonal1 = [self.board[i] for i in [0, 4, 8]]
            if all([s == letter for s in diagonal1]):
                return True
            diagonal2 = [self.board[i] for i in [2, 4, 6]]
            if all([s == letter for s in diagonal2]):
                return True

        return False
```

Kane AI Prototype Class

In [8]:

```
class KanePrototype:
    def __init__(self, letter):
        self.letter = letter

    def get_move(self, game):
        square = random.choice(game.available_moves())
        for move in game.available_moves():
```

```
            game.make_move(move, self.letter)
            if game.current_winner == self.letter:
                square = move  # winning move found
                break
            game.board[move] = ' '  # reset the move
        return square
```

Human Player Class

This class is for playing against the AI manually:

In [9]:

```
class HumanPlayer:
    def __init__(self, letter):
        self.letter = letter

    def get_move(self, game):
        valid_square = False
        while not valid_square:
            square = input(f"{self.letter}'s turn. Input move (0-8): ")
            try:
                val = int(square)
                if val not in game.available_moves():
                    raise ValueError
                valid_square = True
            except ValueError:
                print("Invalid square. Try again.")

        return val
```

Game Play Function

This function manages the game play, alternating between players:

In [12]:

```
import time
import random

def play(game, player_x, player_o, print_game=True):
    if print_game:
        game.print_board()

    letter = 'X'  # Starting letter
    while game.empty_cells():
        if letter == 'O':
            square = player_o.get_move(game)
        else:
            square = player_x.get_move(game)

        if game.make_move(square, letter):
            if print_game:
                print(f"{letter} makes a move to square {square}")
                game.print_board()
                print('')  # Empty line for readability

            if game.current_winner:
                if print_game:
                    print(f"{letter} wins!")
                return letter  # Ends the loop and exits the game
            letter = 'O' if letter == 'X' else 'X'  # Switches player

        time.sleep(0.8)

    if print_game:
        print("It's a tie!")
```

Putting it all together

Now I can run a game by creating an instance of TicTacToe, KanePrototype and HumanPlayer.

Here's an example of a gameplay between AI (X) and Human (0).

In [15]:
```
if __name__ == '__main__':
    x_player = KanePrototype('X')
    o_player = HumanPlayer('O')
    t = TicTacToe()
    play(t, x_player, o_player, print_game=True)
```

```
|   |   |   |
|   |   |   |
|   |   |   |
X makes a move to square 2
|   |   | X |
|   |   |   |
|   |   |   |

O's turn. Input move (0-8): 0
O makes a move to square 0
| O |   | X |
|   |   |   |
|   |   |   |

X makes a move to square 5
| O |   | X |
|   |   | X |
|   |   |   |

O's turn. Input move (0-8): 8
O makes a move to square 8
| O |   | X |
|   |   | X |
|   |   | O |

X makes a move to square 3
| O |   | X |
| X |   | X |
|   |   | O |

O's turn. Input move (0-8): 4
O makes a move to square 4
| O |   | X |
| X | O | X |
|   |   | O |

O wins!
```

In [ ]:

Opportunities still working on

- Some challenges that I still need to fix with this code is that the AI isn't trying to win when I give it the opportunity to win. So I need to try and iterate on that part of the code
- Also, I need to have it play against other AI
- Detailed testing plan which is part of the Development process
- Further opportunities to enhance Kane AI, like implementing strategic opening moves, blocking opponent wins or even adding different levels of difficulty, keeping a score count for each of the matches etc

4.2 Abel AI Prototype

For Abel AI, a rudimentary machine learning model was prototyped, using a basic form of reinforcement learning. The goal was to allow Abel AI to learn from each game by updating its strategy based on the outcome.

Prototype Code for Abel AI:

Importing necessary libraries and creating the DQN Agent class

In [1]:
```
pip install tensorflow
```

In [3]:
```
import numpy as np
import random
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
```

In [29]:
```
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = []
        self.gamma = 0.95  # discount rate
        self.epsilon = 1.0  # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = self._build_model()

    def _build_model(self):
        # Neural Net for Deep-Q learning Model
        model = Sequential()
        model.add(Dense(24, input_dim=self.state_size, activation
='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mse', optimizer=Adam(learning_rate=sel
f.learning_rate))
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, don
e))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        act_values = self.model.predict(state)
        return np.argmax(act_values[0])  # returns action

    def replay(self, batch_size):
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target = reward
            if not done:
                target = reward + self.gamma * np.amax(self.model.p
redict(next_state)[0])
            target_f = self.model.predict(state)
            target_f[0][action] = target
            self.model.fit(state, target_f, epochs=1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

Creating the Tic Tac Toe game class

In [30]:
```
class TicTacToe:
    def __init__(self):
        self.board = [' ' for _ in range(9)]  # 3x3 board
        self.current_winner = None  # Track the winner!

    def print_board(self):
        for row in [self.board[i*3:(i+1)*3] for i in range(3)]:
            print('| ' + ' | '.join(row) + ' |')

    def available_moves(self):
        return [i for i, x in enumerate(self.board) if x == ' ']

    def empty_cells(self):
        return ' ' in self.board

    def make_move(self, square, letter):
        if self.board[square] == ' ':
            self.board[square] = letter
            if self.check_winner(square, letter):
                self.current_winner = letter
            return True
        return False

    def check_winner(self, square, letter):
        # Check row
        row_ind = square // 3
        row = self.board[row_ind*3:(row_ind+1)*3]
        if all([s == letter for s in row]):
            return True

        # Check column
        col_ind = square % 3
        column = [self.board[col_ind + i*3] for i in range(3)]
        if all([s == letter for s in column]):
            return True

        # Check diagonals
        if square % 2 == 0:
            diagonal1 = [self.board[i] for i in [0, 4, 8]]
            if all([s == letter for s in diagonal1]):
                return True
            diagonal2 = [self.board[i] for i in [2, 4, 6]]
            if all([s == letter for s in diagonal2]):
                return True

        return False

    def get_state(self):
        state = []
        for cell in self.board:
            if cell == 'X':
                state.append(1)
            elif cell == 'O':
                state.append(-1)
            else:
                state.append(0)
        return np.array([state])
```

Training the DQN Agent

In [31]:
```
def train_dqn_agent(episodes, batch_size):
    agent = DQNAgent(state_size=9, action_size=9)
    for e in range(episodes):
        # Initialize the Tic Tac Toe game
        game = TicTacToe()
        state = game.get_state()

        for time in range(9):  # Maximum of 9 moves in Tic Tac Toe
            action = agent.act(state)
            if action not in game.available_moves():
                reward = -10  # Penalty for invalid move
                done = True
            else:
                game.make_move(action, 'X')  # DQN agent plays as
'X'
                next_state = game.get_state()
                reward = 0
                done = game.current_winner is not None

                if not done:
                    # Random opponent move
                    opponent_move = random.choice(game.available_mo
ves())
                    game.make_move(opponent_move, 'O')
                    next_state = game.get_state()
                    done = game.current_winner is not None
                    if done:
                        reward = -10  # Penalty if opponent wins

                if done and game.current_winner == 'X':
                    reward = 10  # Reward for winning

            agent.remember(state, action, reward, next_state, done)
            state = next_state

            if done:
                break

        agent.replay(batch_size)

    # Save the model weights
    agent.model.save_weights('dqn_agent_weights.h5')
    return agent
```

Small example of training on a 10 episodes and batch size of 2 (these are really small numbers just to see how this will go in prototype; when I do the full development it would be a much larger number

In [42]:
```
if __name__ == "__main__":
    episodes = 10  # Number of games to play
    batch_size = 2  # Training batch size
    train_dqn_agent(episodes, batch_size)
```

```
1/1 [==============================] - 0s 51ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 17ms/step
```

Initialise the DQN agent

In [33]:
```
# Initialize a new DQN agent
agent_for_playing = DQNAgent(state_size=9, action_size=9)

# Load the previously saved weights
agent_for_playing.model.load_weights('dqn_agent_weights.h5')

# Now we can use agent_for_playing to play the game
```

Defining a random move player

In [34]:
```
class RandomPlayer:
    def __init__(self, letter):
        self.letter = letter

    def get_move(self, game):
        return random.choice(game.available_moves())
```

Setting up the game play function

In [37]:
```
def play_game(dqn_agent, game, verbose=False):
    current_player = 'O'  # Start with player 'O', which is the ran
dom player

    while game.empty_cells():
        if game.current_winner:
            # Game has been won
            if verbose:
                print(f"Player {game.current_winner} wins!")
            return game.current_winner

        # Determine the move based on the current player
        if current_player == 'X':
            # DQN agent's turn
            state = game.get_state()
            action = dqn_agent.act(state)
        else:
            # Random player's turn
            action = RandomPlayer('O').get_move(game)

        # Check if the move is valid
        if action not in game.available_moves():
            # Invalid move by DQN agent, random player wins
            if current_player == 'X':
                return 'O'
            else:
                # Invalid move by random player, DQN agent wins
                return 'X'

        # Make the move
        game.make_move(action, current_player)

        # Switch turns
        current_player = 'X' if current_player == 'O' else 'O'

        # Optionally, print the board
        if verbose:
            game.print_board()

    if verbose:
        print("It's a tie!")

    return 'Draw'
```

Sample game play!

```
In [40]: if __name__ == "__main__":
             dqn_agent = agent_for_playing
             game = TicTacToe()
             result = play_game(dqn_agent, game, verbose=True)
             print("Game result:winner is ", result)
```

```
|   |   | O |
|   |   |   |
|   |   |   |
|   |   | O |
|   | X |   |
|   |   |   |
| O |   | O |
|   | X |   |
|   |   |   |
| O |   | O |
|   | X |   |
|   |   | X |
| O | O | O |
|   | X |   |
|   |   | X |
Player O wins!
Game result:winner is  O
```

Some ideas on further development for Abel AI

- Performance Evaluation: How well the DQN agent performs against the random player. Does it win more often than it loses? Does it appear to be learning and improving its strategy over time?
- Further Training: If the agent's performance isn't satisfactory, it may benefit from additional training.
- Hyperparameter Tuning: Experimenting with different settings for hyperparameters like the learning rate, discount factor (gamma), and exploration rate (epsilon).
- Complexity Increase: If the agent masters Tic Tac Toe, I could consider increasing the complexity of the game or trying a different game. This can be a good test of the adaptability of my AI model.
- Code Optimization: Optimization might be required for more complex scenarios.

4.3 Overall Iteration Process

Both prototypes were integrated into the Tic Tac Toe game environment. Initial testing involved having Kane AI and Abel AI play against a human player or random agent. These tests provided insights into the effectiveness of Kane AI's rule-based strategy and the potential learning curve for Abel AI.

Challenges and Adjustments:

- For Kane AI, balancing the decision-making process to avoid overly predictable gameplay was challenging. Iterations focused on refining the rule set to make gameplay more dynamic.
- For Abel AI, the primary challenge was setting up a framework for learning and decision-making. The initial prototype used random moves, but this would evolve into a more sophisticated learning algorithm in the development phase.

In Summary

The prototyping and iteration phase was instrumental in laying the groundwork for both Kane AI and Abel AI. Lessons learned from this phase were critical in informing the subsequent development process, including feasibility, also ensuring that the foundation for both AIs would be robust and capable of being built upon.

Section 5: Development and Testing

5.1 Test Plan for Kane AI

Testing Kane AI, especially in a prototype phase, is crucial to ensure that it behaves as expected. Here are the steps and methods I can use to test the Kane AI prototype in the Tic Tac Toe environment:

1. Unit Testing

- Basic Functionality: Test individual functions like get_move to ensure they return valid moves.
- Winning Move Detection: Create scenarios where Kane AI has a winning move available. The AI should always take this move.
- Blocking Moves: Set up scenarios where the opponent (another AI or a human) is about to win. Ensure Kane AI blocks these moves if possible.

2. Integration Testing

- Game Integration: Integrate Kane AI into the game environment and observe its performance in full games.
- Opponent Variations: Have Kane AI play against different types of opponents: a random-move AI, a human player, or even itself. This tests its adaptability and robustness.

3. Manual Play-Throughs

- Human vs. AI: Play games manually against Kane AI. This allows us to test its decision-making in various game states and also its reactions to unexpected or sub-optimal human moves.

4. Automated Game Simulations

- High-Volume Testing: Automate a large number of games between Kane AI and other opponents. Analyse the win/loss/draw statistics to gauge overall performance.
- Consistency Check: Ensure Kane AI consistently makes the best moves in known scenarios. For instance, if the centre square is available on the first move, Kane AI should choose it.

5. Debugging and Logging

- Implement logging within the AI's decision-making process. This can help trace and understand the AI's choices in each move, which is invaluable for diagnosing issues or unexpected behaviours.

6. Performance Analysis

- Speed: Measure the time Kane AI takes to make decisions. It should be reasonably fast, without unnecessary computation.
- Memory Usage: Ensure that the AI's implementation does not lead to excessive memory use, which could be a concern in more complex games.

5.2 Test Plan for Abel AI

Overall outline for a test plan:

1. Unit Testing

- Basic Functionality: Test individual functions

2. Baseline Performance Testing

- Objective: Establish the initial performance level of the AI.
- Method:
    - Have the AI play a set number of games (e.g., 100) against a random-move opponent.
    - Record win/loss/draw rates.
- Expected Outcome:
    - Baseline statistics against which future improvements can be measured.

3. Learning and Adaptation Testing

- Objective: Assess the AI's ability to learn and adapt over time.
- Method:

        - Conduct training sessions over increasing numbers of games (e.g., 100, 500, 1000 games).
        - After each training session, measure performance against the same random-move opponent.
    - Expected Outcome:
        - Improvement in performance, evidenced by an increased win rate and decreased loss rate over time.

4. Strategy and Decision-making Analysis

- Objective: Analyse the strategic development of the AI.
- Method:
    - Review game logs to assess the moves made by the AI.
    - Identify patterns, strategies, common moves in winning games, and mistakes in losing games.
- Expected Outcome:
    - Increased sophistication in the AI's gameplay and decision-making process.

5. Robustness and Consistency Testing

- Objective: Evaluate the AI's consistency and robustness across different scenarios.
- Method:
    - Test the AI against various opponents with different strategies (aggressive, defensive, random).
    - Run repeated sessions to check for consistent performance.
- Expected Outcome:
    - Consistent performance regardless of opponent's strategy.

6. Stress Testing

- Objective: Determine the limits of the AI's performance under extreme conditions.
- Method:
    - Increase the speed of play, limit thinking time, or have the AI play a very high number of games in a row.
- Expected Outcome:
    - Identification of performance thresholds and potential breakdown points.

7. Comparison with Kane AI

- Objective: Compare Abel AI's performance with that of Kane AI (the rule-based system).
- Method:
    - Have both AIs play against each other and against common opponents.
    - Analyse differences in strategies and outcomes.
- Expected Outcome:
    - Clear understanding of the strengths and weaknesses of each approach.

Section 6: Final evaluations

At a high level the plan is to do a complete evaluation of how successful both the AI's are and then write a report including statistical results for the final report.

Section 7: References

- **IBM's Deep Blue**

Murray Campbell, A.Joseph Hoane, Feng-hsiung Hsu, Deep Blue, Artificial Intelligence, Volume 134, Issues 1–2, 2002, Pages 57-83, ISSN 0004-3702, https://doi.org/10.1016/S0004-3702(01)00129-1

- **Google's AlphaGo**

Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961

- **OpenAI's Dota 2 AI**

Berner, Christopher & Brockman, Greg & Chan, Brooke & Cheung, Vicki & Dębiak, Przemysław & Dennison, Christy & Farhi, David & Fischer, Quirin & Hashme, Shariq & Hesse, Chris & Józefowicz, Rafal & Gray, Scott & Olsson, Catherine & Pachocki, Jakub & Petrov, Michael & Pinto, Henrique & Raiman, Jonathan & Salimans, Tim & Schlatter, Jeremy & Zhang, Susan. (2019). Dota 2 with Large Scale Deep Reinforcement Learning.

- **Stockfish chess engine**

Open source chess engine (no date) *Stockfish*. Available at: https://stockfishchess.org/ (Accessed: 07 January 2024)

- **MCTS in Game Playing AI**

  Browne, Cameron & Powley, Edward & Whitehouse, Daniel & Lucas, Simon & Cowling, Peter & Rohlfshagen, Philipp & Tavener, Stephen & Perez Liebana, Diego & Samothrakis, Spyridon & Colton, Simon. (2012). A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games. 4:1. 1-43. 10.1109/TCIAIG.2012.2186810.

- **Tic Tac Toe - AI**
  - Website: https://www.tensorflowtictactoe.co/#
  - Laborde, G. (n.d.). *Watch an AI learn from your tic tac toe games*. Tic Tac Toe - AI. https://www.tensorflowtictactoe.co/#
  - https://github.com/GantMan/tictactoe-ai-tfjs

- CM 3020 Artificial Intelligence - Goldsmiths University of London (coursera)