Computational Intelligence Activity Log

Giulio Maselli, s306125

A.A. 2023/24

1 Lab 1: Set Covering A*

This section provides a detailed look at the Python implementation of the set-covering problem using the A* search algorithm.

```python
import numpy as np
from queue import PriorityQueue
from random import random
from functools import reduce
from collections import namedtuple

PROBLEM_SIZE = 50
NUM_SETS = 100
SETS = tuple(np.array([random() < 0.2 for _ in range(PROBLEM_SIZE)])
             for _ in range(NUM_SETS))
State = namedtuple('State', ['taken', 'not_taken'])

# Stima del numero di elementi che devono essere ancora coperti dai set nello stato attuale
def h(state):
    return PROBLEM_SIZE - sum(
        reduce(
            np.logical_or,
            [SETS[i] for i in state.taken],
            np.array([False for _ in range(PROBLEM_SIZE)]),
        )
    )

# Numero di elementi coperti dai set nello stato attuale
def g(state):
    covered_elements = sum(
        reduce(
            np.logical_or,
            [SETS[i] for i in state.taken],
            np.array([False for _ in range(PROBLEM_SIZE)]),
        )
    )
    return PROBLEM_SIZE - covered_elements

# Verifica se lo stato attuale comprende tutti gli elementi
def goal_check(state):
    return np.all(reduce(np.logical_or, [SETS[i] for i in state.taken], np.array([False for _ in range(PROBLEM_SIZE)])))

assert goal_check(State(set(range(NUM_SETS)), set())), "Problem not solvable"

frontier = PriorityQueue()
initial_state = State(set(), set(range(NUM_SETS)))
frontier.put((h(initial_state), initial_state))

counter = 0
_, current_state = frontier.get()


```

```python
# A*
while not goal_check(current_state):
    counter += 1
    for action in current_state.not_taken:
        new_state = State(
            current_state.taken ^ {action},
            current_state.not_taken ^ {action})
        frontier.put((g(new_state) + h(new_state), new_state))
    _, current_state = frontier.get()
print(f"Solved in {counter} steps ({len(current_state.taken)} tiles)")
print(current_state)

# Solved in 6 steps (6 tiles)
# State(taken={1, 70, 50, 51, 53, 86}, not_taken={0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15...}
```

Here the A* algorithm initializes the search with an empty '**taken**' set, and all sets in '**not_taken**', then uses a priority queue to manage states, prioritizing them based on the sum of the cost so far ('**g**') and the estimated cost ('**h**')

2 Lab 2: Nim with ES

That's my implementation of a game-playing agent for the game of Nim, using Evolution Strategies (ES). Nim is a mathematical game of strategy in which two players take turns removing objects (matches) from distinct piles.
The backbone code already had three implemented strategies, *pure_random*, *optimal* and *gabriele*, which I transformed into *evolvable_based_on_ES_1_lambda*.

1. The 'select_parent' function randomly chooses a 'parent' from the current population, which will then be used to generate new moves (offspring). This selection is done in such a way that moves with a better 'fitness' (in this context a measure of how close a move is to winning the game) are more likely to be chosen.

```
def select_parent(population, tournament_size=2):
    return min(random.choices(population, k=tournament_size),
    key=lambda i: i.fitness)
```

2. The '*evolve_one_plus_lambda*' function is responsible for creating new offspring from the selected parents. This is done by applying the '*mutation*' function to the parents to create a new set of moves.

```
    def evolve_one_plus_lambda(parent, mutation_rate, state):
    child = mutation(parent, state)
    return child
```

3. The '*mutation*' function introduces variation into the population. It takes a parent move and creates a new move (child) by randomly altering the parent's move. This could involve changing the row from which objects are removed or the number of objects taken. Mutation introduces new strategies into the population and is essential for exploring the space of possible strategies.

```
    def mutation(parent, nim):
    if parent is None:
        # in assenza di genitore, genera una mossa casuale
        row = random.choice([r for r, c in enumerate(nim.rows)
     if c > 0])
        num_objects = random.randint(1, nim.rows[row])
        a = nim_sum(nim)
        offspring = Move(row, num_objects, a)
    else:
        if nim.k is None:
            elements = random.randrange(1, nim.rows[parent.row
    ] + 1)
        else:
            elements = min(nim.k, random.randrange(1, nim.rows
    [parent.row] + 1))
        temp_rows = nim.rows.copy()
        temp_rows[parent.row] -= elements
        offspring = Move(parent.row, elements, nim_sum(
    temp_rows))

    return offspring
```

4. The 'evolvable_based_on_ES_1_lambda' function implements an evolutionary strategy (ES) specifically the (1+lambda) strategy, to evolve game moves for playing Nim. The process aims to discover an effective move by simulating evolution over a series of generations.

```python
def evolvable_based_on_ES_1_lambda(self, state: Nim, alpha = 0.5):
    # Popolazione iniziale di possibili mosse
    population = [mutation(None, state) for _ in range(POPULATION_SIZE)]

    for g in range(N_GENERATIONS):
        offspring = []

        for i in range(LAMBDA):
            # Selezione del genitore (nel tuo caso, potrebbe essere tramite il torneo)
            parent = select_parent(population)

            # Generazione del figlio mediante mutazione del genitore
            child = mutation(parent, state)

            # Aggiornamento del genitore con alpha
            parent.row = int(parent.row + alpha * (child.row - parent.row))
            parent.num_objects = int(parent.num_objects + alpha * (child.num_objects - parent.num_objects))

            # Aggiungi il figlio alla lista degli offspring
            offspring.append(child)

        # Unisci genitori e figli per formare la nuova popolazione
        population += offspring

        # Ordina la popolazione in base al fitness e seleziona i primi POPULATION_SIZE individui
        population = sorted(population, key=lambda i: i.fitness, reverse=False)[:POPULATION_SIZE]
        logging.debug(f"Actual best {population[0]}")

    # Restituisci la migliore mossa
    best_move = Nimply(population[0].row, population[0].num_objects)
    return best_move
```

3 Lab 9: Genome and Evolutionary Strategy

This code outlines an implementation of an evolutionary algorithm designed to solve an optimization problem, with the ultimate goal of minimizing the number of fitness function calls. The optimization problem in question involves finding a binary genome sequence that achieves the highest fitness according to a given fitness function.

The key components are as usual the 'parent selection', 'crossover' and 'mutation'.

```

class EvolutionaryAlgorithm:
    def __init__(self, problem_instance, genome_length, population_size=50, generations=100, mutation_rate=0.01, elitism_percentage=0.1):
        self.problem = problem_instance
        self.genome_length = genome_length
        self.population_size = population_size
        self.generations = generations
        self.mutation_rate = mutation_rate
        self.elitism_percentage = elitism_percentage
        self.best_fitness_history = []  # To store the best fitness in each generation

    def initialize_population(self):
        return [choices([0, 1], k=self.genome_length) for _ in range(self.population_size)]

    def crossover(self, parent1, parent2):
        # xover
        crossover_point = randint(1, self.genome_length - 1)
        child = parent1[:crossover_point] + parent2[crossover_point:]
        return child

    def mutate(self, genome):
        # mutation
        mutated_genome = [bit ^ (random() < self.mutation_rate) for bit in genome]
        return mutated_genome

    def run_algorithm(self):
        population = self.initialize_population()
        best_fitness_per_generation = []

        for generation in range(1, self.generations + 1):
            # Evaluate fitness for each individual
            fitness_values = [self.problem(ind) for ind in population]

            # Store the best fitness in each generation
            best_fitness_per_generation.append(max(fitness_values))

            # top individuals
            sorted_indices = sorted(range(len(fitness_values)), key
```

```
=lambda k: fitness_values[k], reverse=True)
        selected_parents = [population[i] for i in
sorted_indices[:self.population_size // 2]]

        # offspring creation
        offspring = []
        for i in range(0, len(selected_parents) - 1, 2):
            parent1 = selected_parents[i]
            parent2 = selected_parents[i + 1]
            child = self.crossover(parent1, parent2)
            child = self.mutate(child)
            offspring.append(child)

        # if the population size is odd, add the last ind
        if len(selected_parents) % 2 == 1:
            offspring.append(selected_parents[-1])

        population = selected_parents + offspring

        # best fitness in each generation
        best_fitness = max(fitness_values)
        print(f"Generation {generation}, Best Fitness: {
best_fitness:.2%}")

    # best ind in the final gen
    best_index = max(range(len(fitness_values)), key=lambda k:
fitness_values[k])
    best_individual = population[best_index]

    # Store the best fitness history for each generation
    self.best_fitness_history = best_fitness_per_generation

    return best_individual

def get_best_fitness_history(self):
    return self.best_fitness_history
```

Best Solution Overall:

- Population size: *1000*
- Generations: *100*
- Mutation rate: *0.01*
- Elitism percentage: *0.3*
- Fitness: *75.00%*
- Number of fitness calls: *75253*

4 Lab 10: Tic-Tac-Toe Q-Learning Project

This code outlines an implementation of a **Q-Learning algorithm** for playing Tic-Tac-Toe against a random opponent. The implementation involves training a Q-Learning agent (**QLearningPlayer**) to learn optimal strategies through repeated gameplay against a **RandomPlayer**, who selects moves randomly. The process aims to reinforce winning strategies and penalize losing ones, leveraging the Q-Learning framework to iteratively improve the agent's performance.

4.1 Core components

- TicTacToe: Represents the game environment with methods to make moves, check for wins or draws, and get available moves. The game board is a 3x3 numpy array, and players are represented by 1 (X) and -1 (O).
- RandomPlayer: An agent that selects moves randomly from the available spots on the board.
- QLearningPlayer: A reinforcement learning (RL) agent that uses Q-Learning to decide on the best moves. It maintains a Q-table mapping states to action values, updated based on the outcome of each game.

```
    class QLearningPlayer:
def __init__(self, symbol, alpha=0.1, gamma=0.9, epsilon=0.2):
    self.symbol = symbol  # 1 per X, -1 per O
    self.alpha = alpha    # learning rate
    self.gamma = gamma    # discount factor
    self.epsilon = epsilon  # exploring prob (epsilon greedy)
    self.q_table = {}  # state -> action -> value

def get_state(self, game):
    return str(game.board.reshape(9))

def choose_move(self, game):
    state = self.get_state(game)
    if np.random.rand() < self.epsilon:
        # exploring
        return random.choice(game.get_available_moves())
    else:
        # exploiting
        self.q_table.setdefault(state, {})
        if not self.q_table[state]:
            # if none -> rand
            return random.choice(game.get_available_moves())
        best_move = max(self.q_table[state], key=self.q_table[state].get)
        return eval(best_move)

def update_q_values(self, prev_state, action, reward, next_state):
    self.q_table.setdefault(prev_state, {})
    self.q_table.setdefault(next_state, {})
    prev_q = self.q_table[prev_state].get(str(action), 0)
```

```
    max_next_q = max(self.q_table[next_state].values(), default=0)
    self.q_table[prev_state][str(action)] = prev_q + self.alpha * (reward + self.gamma * max_next_q - prev_q)

def update_q_table(self, game_history):
    # game_history = list (state, action, reward, next_state)
    for i in range(len(game_history) - 1):
        state, action, reward, next_state = game_history[i]
        self.update_q_values(state, action, reward, next_state)

    # update q-values for final_state
    final_state, final_action, final_reward, _ = game_history[-1]
    self.update_q_values(final_state, final_action, final_reward, final_state)

def receive_reward(self, game):
    # rewars values
    if game.is_winner(self.symbol):
        return 1  # win
    elif game.is_draw():
        return 0  # draw, maybe -0.5
    else:
        return -1  # loss

def save_q_table(self, filename='q_table.pkl'):
    with open(filename, 'wb') as f:
        pickle.dump(self.q_table, f)
    print(f"Q-table saved in {filename}.")

def load_q_table(self, filename='q_table.pkl'):
    with open(filename, 'rb') as f:
        self.q_table = pickle.load(f)
    print(f"Q-table loaded from {filename}.")
```

- Training:

```
def play_game(q_player, random_player, episodes=100000):
for episode in range(episodes):
    game = TicTacToe()
    game_history = []  # init game history

    while True:
        current_player = q_player if game.current_player == q_player.symbol else random_player
        move = current_player.choose_move(game)
        if move is None:  # if none break loop
            break

        # save current_state if qlplayer
        if isinstance(current_player, QLearningPlayer):
            prev_state = current_player.get_state(game)

        game.make_move(*move)

```

```python
        if isinstance(current_player, QLearningPlayer):
            next_state = current_player.get_state(game)
            reward = 0
            game_history.append((prev_state, move, reward, next_state))

        # winner check
        if game.is_winner(q_player.symbol) or game.is_winner(random_player.symbol) or game.is_draw():
            final_reward = q_player.receive_reward(game)  # final reward

            # update reward for qlplayer
            if isinstance(current_player, QLearningPlayer) and game_history:
                prev_state, move, _, next_state = game_history[-1]
                game_history[-1] = (prev_state, move, final_reward, next_state)
            break

    # update qtable with complete history
    if game_history:
        q_player.update_q_table(game_history)

    game.reset()  # next game

print("Learning completed after {} episodes".format(episodes))
```

- Testing win rate on 1000 games: [00:01¡00:00, 756.95it/s]
  Win rate: 77.60%
  Total games: **1000**, Wins: **776**, Draws: **59**, Losses: **165**

5 Quixo

5.1 Introduction:

The essence of this project is to develop a framework where different types of AI players, including a Deep Q-Learning (DQL) agent and a Minimax agent, can learn and compete in the game of Quixo.

5.2 Key Components

5.2.1 Game Environment (Game.py)

Central to the project, this class manages the game state, enforces the rules of Quixo, and determines the winner. It allows moves to be made on the board, checks for win conditions, and ensures the game progresses logically from start to finish.

5.2.2 Players

- **RandomPlayer**: Serves as a baseline opponent, making moves by randomly selecting from available options.

```python
class RandomPlayer(Player):
    def __init__(self) -> None:
        super().__init__()

    def make_move(self, game) -> tuple[tuple[int, int], Move]:
        from_pos = (random.randint(0, 4), random.randint(0, 4)
    )
        move = random.choice([Move.TOP, Move.BOTTOM, Move.LEFT
    , Move.RIGHT])
        return from_pos, move

```

- **Minimax Player**: Implements the Minimax algorithm with a specified depth of lookahead. It evaluates game states to make the most advantageous moves, considering the possible responses of the opponent. This player introduces a strategic challenge, forcing learning agents to develop more sophisticated strategies to win. The class includes methods for initializing the player, evaluating the game state, running the Minimax algorithm to choose the best move, and executing the selected move.

```python
class MiniMaxPlayer(Player):
    def __init__(self, depth: int, player_index) -> None:
        super().__init__()
        self.depth = depth
        self.player_index = player_index

```

```
    def evaluate_game_state(self, board):
        player_score, player_4_in_row = count_aligned(board, self.player_index, 4)
        opponent_score, opponent_4_in_row = count_aligned(board, 1 - self.player_index, 4)

        # evaluation of important pieces (corners) and alligned pieces

        strategic_value = 0
        corners = [(0, 0), (0, len(board)-1), (len(board)-1, 0), (len(board)-1, len(board)-1)]
        for x, y in corners:
            if board[x, y] == self.player_index:
                strategic_value += 1
            elif board[x, y] == 1 - self.player_index:
                strategic_value -= 1

        # critical situations
        if opponent_4_in_row:
            return -10  # avoid opponent's win
        if player_4_in_row:
            return 10   # win the game

        return player_score - opponent_score + strategic_value


    def minimax(self, board, depth, alpha, beta, maximizing_player):

        if depth == 0:
            return self.evaluate_game_state(board)

        if maximizing_player:
            max_eval = -float('inf')
            for move in get_possible_moves(board, self.player_index):
                # apply the move and compute new board state
                new_board = apply_move(board, move, self.player_index)
                eval = self.minimax(new_board, depth - 1, alpha, beta, False)
                # print(f"Valutazione mossa {move}: {eval}") # Debugging
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
            return max_eval
        else:
            min_eval = float('inf')
            for move in get_possible_moves(board, 1 - self.player_index):
                # apply the move and compute new board state
                new_board = apply_move(board, move, 1 - self.player_index)
                eval = self.minimax(new_board, depth - 1,
```

```
alpha, beta, True)
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    break
            return min_eval


    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        best_score = -float('inf')
        best_action = None  # tuple (from_pos, move)

        for action in get_possible_moves(game.get_board(), self.player_index):
            # apply the move to obtain the new board
            new_board = apply_move(game.get_board(), action, self.player_index)
            score = self.minimax(new_board, self.depth, -float('inf'), float('inf'), True)
            if score > best_score:
                best_score = score
                best_action = action

        # print(f"MiniMaxPlayer best_action: from_pos={best_action[0]}, move={best_action[1]}")
        # print_board(new_board)
        return (best_action[0][1], best_action[0][0]), best_action[1] if best_action is not None else ((0, 0), Move.TOP)

'''
depth = 5 has a great win rate (between 75% and 95%), almost 20 seconds per game

depth = 5 vs. Random
100%
Percentuale vittorie: 94.0%
Percentuale pareggi: 0.0%
Percentuale sconfitte: 6.0%

depth = 3 vs. Random
100%
Percentuale vittorie: 86.0%
Percentuale pareggi: 0.0%
Percentuale sconfitte: 14.0%
'''

```

- **Q-Learning Player**: designed to implement a reinforcement learning agent using the Q-Learning algorithm. This agent is capable of learning optimal moves and strategies by interacting with the game environment and adapting based on the outcomes of its actions.

Initialization (__init__): Sets up the learning agent with initial parameters for learning rate (alpha), discount factor (gamma), exploration rate (epsilon), and initializes the Q-table.
Q-Table Persistence (load_q_table, save_q_table): These functions handle the loading and saving of the Q-table to disk, allowing the agent to retain its learned knowledge between game sessions and continue improving over time.

```python
class QLearningPlayer(Player):
    def __init__(self, player_index, alpha=0.2, gamma=0.9, epsilon=0.2, epsilon_decay=1, epsilon_min=0, preload=True):
        super().__init__()
        self.alpha = alpha  # Learning rate
        self.gamma = gamma  # Discount factor
        self.epsilon = epsilon  # Exploration probability
        self.epsilon_decay = epsilon_decay  # Epsilon decay rate
        self.epsilon_min = epsilon_min  # Minimum epsilon
        self.q_table = {}  # Initialize Q-table
        self.moves_history = []  # Track moves
        self.player_index = player_index
        self.win_count = 0
        self.draw_count = 0
        self.loss_count = 0

        if preload:
            self.q_table = self.load_q_table()

    def load_q_table(self):
        try:
            with open("q_table.pickle", "rb") as f:
                print("File Q-table caricato.")
                return pickle.load(f)
        except FileNotFoundError:
            print("File Q-table non trovato, inizializzazione di una nuova Q-table.")
            return {}
    def save_q_table(self):
        try:
            with open("q_table.pickle", "wb") as f:
                pickle.dump(self.q_table, f)
            print("Q-table salvata con successo.")
        except Exception as e:
            print(f"Errore nel salvataggio della Q-table: {e}")

```

State Representation (state_representation): Converts the current game board into a standardized format that serves as a key for accessing the Q-table, ensuring that the agent can evaluate and learn from specific game states.

Move Selection (make_move): Determines the agent's actions within the game by either exploring new moves randomly or exploiting known moves with high Q-values, based on the epsilon-greedy strategy. This function encapsulates the core of the agent's decision-making process.

```
def state_representation(self, board):
    return str(board.reshape(-1))

def make_move(self, game) -> tuple[tuple[int, int], Move]:
board = game.get_board()
state = self.state_representation(board)

possible_moves = get_possible_moves(board, self.player_index)

if random.random() < self.epsilon or state not in self.q_table:
    selected_move = random.choice(possible_moves)
else:
    max_q_value = float("-inf")
    selected_action_key = None
    for move in possible_moves:
        action_key = self.move_to_key(move)
        q_value = self.q_table.get(state, {}).get(action_key, float("-inf"))
        if q_value > max_q_value:
            max_q_value = q_value
            selected_action_key = action_key

    # Se nessuna mossa ha Q superiore, scelta casuale tra le mosse
    if selected_action_key is None:
        selected_move = random.choice(possible_moves)
    else:
        selected_move = self.key_to_move(selected_action_key)

# Simula mossa selezionata per valutare il suo effetto
simulated_board = apply_move(deepcopy(board), selected_move, self.player_index)
simulated_state = self.state_representation(simulated_board)
reward = self.evaluate_move_effect(game, selected_move)

# Calcola il miglior Q per il prossimo stato simulato
next_max = max(self.q_table[simulated_state].values(), default=0) if simulated_state in self.q_table else 0
self.update_q_table(state, selected_move, reward, next_max)
self.epsilon = max(self.epsilon * self.epsilon_decay, self.epsilon_min)
#print(f"Epsilon: {self.epsilon}")

self.moves_history.append((state, selected_move, simulated_state))

```

```
#print(f"{board}")
#print(f"Selected Move: {selected_move[0][0], selected_move[0][1]}")
#print(f"Possible Moves {possible_moves}")
selected_move = ((selected_move[0][1], selected_move[0][0]), selected_move[1])

return selected_move


def move_to_key(self, move):
    # Converte la mossa in una chiave utilizzabile nella Q-table
    return (move[0], move[1].value)

def key_to_move(self, key):
    # Converte una chiave della Q-table in una mossa
    return (key[0], Move(key[1]))

```

Learning from Experience (update_q_table, learn): Updates the Q-values in the Q-table based on the agent's experiences, applying the Q-Learning formula to adjust values towards optimal strategies over time. **Evaluating Actions (evaluate_move_effect)**: Assesses the immediate impact of actions to assign rewards, guiding the learning process by providing feedback on the effectiveness of moves in achieving favorable outcomes or preventing unfavorable ones.

```
def update_q_table(self, state, action, reward, next_max):
    if state not in self.q_table:
        self.q_table[state] = {}

    action_key = self.move_to_key(action)
    if action_key not in self.q_table[state]:
        self.q_table[state][action_key] = 0  # Inizializza se non presente

    # Aggiorna Q usando next_max, ovvero il massimo valore Q per il prossimo stato
    self.q_table[state][action_key] += self.alpha * (reward + self.gamma * next_max - self.q_table[state][action_key])

'''
def adjust_epsilon_dynamically(self):
    # Riduce epsilon piu' lentamente se l'agente sta vincendo o numero di pareggi alto
    if self.win_count > self.loss_count or self.draw_count > (self.win_count + self.loss_count):
        self.epsilon *= (self.epsilon_decay ** 0.5)  # Riduzione piu' lenta
    elif self.loss_count >= self.win_count:
        # Aumenta epsilon leggermente se ci sono molte sconfitte, esploraz. piu' ampia
        self.epsilon = min(self.epsilon / (self.epsilon_decay ** 0.5), 1.0)

```

```
    self.epsilon = max(self.epsilon, self.epsilon_min)
'''

def evaluate_move_effect(self, game, move):
    current_board = game.get_board()
    ql_max_elements_before, ql_otk_before = count_aligned(current_board, self.player_index)
    opp_max_elements_before, opp_otk_before = count_aligned(current_board, 1 - self.player_index)

    # Simula la mossa
    new_board = apply_move(deepcopy(current_board), move, self.player_index)
    ql_max_elements_after, ql_otk_after = count_aligned(new_board, self.player_index)
    opp_max_elements_after, opp_otk_after = count_aligned(new_board, 1 - self.player_index)

    # reward basato su differenza e prevenzione di mosse critiche
    reward = 0

    # Miglioramento della posizione
    if ql_max_elements_after > ql_max_elements_before:
        reward += (ql_max_elements_after - ql_max_elements_before) * 2

    # Blocco dell'avversario
    if opp_otk_before and not opp_otk_after:
        reward += 2

    # Vittoria imminente
    if ql_otk_after and not ql_otk_before:
        reward += 3

    # Vittoria
    if ql_max_elements_after == 5:
        reward += 6

    # Avversario vicino alla vittoria
    if not opp_otk_before and opp_otk_after:
        reward -= 2

    # Sconfitta
    if opp_max_elements_after > 4:
        reward -= 5

    # mosse neutre
    if ql_max_elements_after <= ql_max_elements_before and not ql_otk_after:
        reward -= 1

    return reward

    def learn(self, game, move, reward):
    board = game.get_board()
    state = self.state_representation(board)
```

```
    next_state = deepcopy(board)
    apply_move(next_state, move, self.player_index)  # Simula la mossa sullo stato

    next_state_rep = self.state_representation(next_state)
    next_max = max(self.q_table[next_state_rep].values(), default=0) if next_state_rep in self.q_table else 0
    self.update_q_table(state, move, reward, next_max)

```

Game Finalization (finalize_game): Processes the result of a game to update learning parameters and the Q-table based on the outcome, reinforcing successful strategies and penalizing unsuccessful ones. This step ensures that the agent learns from the entire game, not just individual moves.

```
def finalize_game(self, result):
if result == "win":
    self.win_count += 1
    reward = 10  # vittoria
elif result == "draw":
    reward = 1  # pareggio
    self.draw_count += 1
else:  # "lose"
    reward = -10  # sconfitta
    self.loss_count += 1
if (self.win_count + self.draw_count + self.loss_count) % 10 == 0:
    self.win_count, self.draw_count, self.loss_count = 0, 0, 0
# Aggiorna la Q-table retroattivamente sulle mosse effettuate durante la partita
for state, action, next_state in reversed(self.moves_history):
    current_q_value = self.q_table.get(state, {}).get(self.move_to_key(action), 0)
    next_max = 0
    if next_state in self.q_table:
        next_max = max(self.q_table[next_state].values())
    self.q_table.setdefault(state, {})[self.move_to_key(action)] = current_q_value + self.alpha * (reward + self.gamma * next_max - current_q_value)
# Resetta la history delle mosse per la prossima partita
self.moves_history.clear()

```

- **Deep-QLearning Player**: This agent leverages deep learning to approximate the optimal action-value function, enabling it to make strategic decisions based on the current state of the game.

  Model Construction (_build_model): Builds the neural network model that approximates the Q-function, determining the value of taking a particular action in a given state. This network is the core of the DQN, guiding the agent's decision-making process.
  Initialization (__init__): Sets up the agent with necessary parameters, including the size of the state and action spaces, and initializes the replay memory for experience replay, which is critical for stabilizing and improving learning outcomes.

```python
class DQNAgent(Player):
    def __init__(self, player_index, state_size, action_size):
        super(DQNAgent, self).__init__()
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=2000)    # Buffer di riproduzione per memorizzare esperienze passate
        self.gamma = 0.95  # discount rate
        self.epsilon = 0.1  # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 1
        self.learning_rate = 0.001  # Tasso di apprendimento per l'ottimizzatore
        self.player_index = player_index
        self.model = self._build_model()

    def _build_model(self):
        # Costruisce il modello della rete neurale con struttura lineare
        model = nn.Sequential(
            nn.Linear(self.state_size, 24),
            nn.ReLU(),
            nn.Linear(24, 24),
            nn.ReLU(),
            nn.Linear(24, self.action_size)
        )
        self.optimizer = optim.Adam(model.parameters(), lr=self.learning_rate)
        return model

```

  Experience Storage (remember): Records experiences (state, action, reward, next state, done) in the replay memory, enabling the agent to learn from a diverse set of experiences by revisiting past states and outcomes.
  Replay (replay): Samples a batch of experiences from the replay memory to update the neural network. This process involves calculating target Q-values and optimizing the network's parameters to reduce the discrepancy between predicted and target Q-values, facilitating learning from past actions.

```python
def remember(self, state, action, reward, next_state, done):
    # Memorizza l'esperienza passata nel buffer di riproduzione
    self.memory.append((state, action, reward, next_state, done))

def replay(self, batch_size, gamma):
    # Effettua l'aggiornamento del modello utilizzando il buffer di riproduzione
    minibatch = random.sample(self.memory, batch_size)
    for state, action, reward, next_state, done in minibatch:
        # Convert state and next_state to PyTorch tensors
        state_tensor = torch.FloatTensor(state).unsqueeze(0)  # Adds a batch dimension
        next_state_tensor = torch.FloatTensor(next_state).unsqueeze(0)  # Adds a batch dimension

        target = reward
        if not done:
            # Use next_state_tensor instead of next_state
            target = (reward + gamma * torch.max(self.model(next_state_tensor)).item())

        # Use state_tensor instead of state
        target_f = self.model(state_tensor)
        target_f[0][action] = target

        self.optimizer.zero_grad()
        # Calculate loss using the tensor, not the NumPy array
        loss = nn.MSELoss()(target_f, self.model(state_tensor))
        loss.backward()
        self.optimizer.step()

```

Action Selection (act): Decides on actions based on the current state, employing an epsilon-greedy strategy that balances exploration (choosing random actions) and exploitation (choosing the best-known action according to the neural network).

Move Translation (index_to_move): Translates action indices determined by the DQN into specific moves within the game, ensuring that actions align with the game's rules and dynamics.

```python
def act(self, state):
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)
    state = torch.FloatTensor(state).unsqueeze(0)
    act_values = self.model(state)
    return np.argmax(act_values.detach().numpy()[0])  # returns action

 def index_to_move(self, action_index, game):
    # Converte un indice di azione in una mossa specifica, assicurandosi che la mossa sia valida.
    possible_moves = get_possible_moves(game.get_board(), self.player_index)
```

```
    if action_index >= 0 and action_index < len(possible_moves):
        return possible_moves[action_index]
    else:
        return random.choice(possible_moves)

```

Model Saving and Loading
Progress Tracking

```
def save_model(self):
    torch.save(self.model.state_dict(), "dql_model.pth")

def save_progress(self):
    progress = {
        "epsilon": self.epsilon,
        "memory": self.memory,
        # Aggiungi altri attributi se necessario
    }
    with open("dql_progress.pickle", 'wb') as f:
        pickle.dump(progress, f)

def load_model(self):
    self.model.load_state_dict(torch.load("dql_model.pth"))
    self.model.eval()  # Imposta il modello in modalita' di valutazione

def load_progress(self):
    with open("dql_progress.pickle", 'rb') as f:
        progress = pickle.load(f)
        self.epsilon = progress["epsilon"]
        self.memory = progress["memory"]

```

Move Making (make_move): Integrates the DQN's action selection and the game's rules to choose and execute moves during gameplay. This function represents the agent's interaction with the game environment, applying its learned strategies to compete.
Reward Calculation (calculate_reward): Assesses the outcomes of actions to assign rewards, guiding the agent's learning by providing feedback on the effectiveness of its decisions relative to the game's outcome.

```
def make_move(self, game):
    state = normalize_state_simple(game.get_board(), game.get_current_player())
    action_index = self.act(state)
    chosen_move = self.index_to_move(action_index, game)
    (x, y), direction = chosen_move
    swapped_position = (y, x)
    swapped_move = (swapped_position, direction)
    return swapped_move

 def calculate_reward(self, game):
    winner = game.check_winner()
    if winner == self.player_index:
        return 1
```

```
    elif winner == -2:
        return 0
    elif winner != -1:
        return -1
    else:
        return 0

```

Step Simulation (step): A hypothetical function that would simulate the effect of an action on the game state, returning the new state, reward, and whether the game has ended. This functionality is essential for evaluating potential moves and their outcomes within the learning process.

```
def step(board, action, player_index):

    # Esegue un passo nel gioco data un'azione e restituisce il nuovo stato, la ricompensa e se c'e' un vincitore

    new_board = apply_move(deepcopy(board), action, player_index)
    winner = check_winner(new_board)

    if winner == player_index:
        reward = 2
        done = True
    elif winner == -2:
        reward = 0
        done = True
    elif winner != -1:
        reward = -1
        done = True
    else:
        reward = 0
        done = False

    return new_board, reward, done

```

5.3 Results

Despite the struggle and initially unconvincing results, after all night long training sessions, seasoned with parameter tuning, the best agent to use for training agents with RL is Minimax with Depth 1 or 3, due to the very long processing times.

```
        -Testing DQL versus RandomPlayer

        Test completato dopo 100 partite
        Vittorie: 78 (78.00%)
        Pareggi: 0 (0.00%)
        Sconfitte: 22 (22.00%)


        -Training DQL versus QLearningPlayer

        Test completato dopo 100 partite
        Vittorie: 74 (74.00%)
        Pareggi: 0 (0.00%)
        Sconfitte: 26 (26.00%)

```