FINAL REPORT

IMPROVING CHESS STRATEGIES WITH DEEP RL METHODS

Venkata Ramana Reddy Duggempudi Pranavi Sriya Vajha
Venkata Sai Vikas Katuru

50539685 50540897 50532536

vduggemp@buffalo.edu pvajha@buffalo.edu vkaturu@buffalo.edu

1 Problem Statement

In this project we aim to create a smart system that learns to play chess on its own using reinforcement learning methods. This system should not only grasp the basic rules of chess but also develop its own advanced strategies. It needs to be flexible enough to adapt to different opponents and situations.

The challenge involves in training two distinct agents using different machine learning techniques: Deep Q Learning (DQN) and Monte Carlo Tree Search (MCTS). These agents learn to play chess effectively through self-play, interactions with each other, or engagements with human players, without any human-derived strategies or interventions. Our goal is to enable these agents to explore a vast number of game states and tactics, pushing the boundaries of their learning capabilities to achieve, or even surpass, Grandmaster-level performance in chess.

2 Environment Setup: PettingZoo Chess Environment

2.1 Overview

PettingZoo is a multi-agent reinforcement learning platform that offers various environments for conducting complex interaction studies between agents. One such environment is the chess game provided within PettingZoo's classic collection. This setup allows for the exploration of complex agent interactions within a structured and strategic game setting, making it an ideal testbed for developing and testing advanced reinforcement learning (RL) algorithms.

2.2 Environment Details:

2.2.1 Action Space:

The environment provides a discrete action space with 4672 possible actions, reflecting the multitude of potential moves in a chess game.

2.2.2 API Type:

It utilizes a Parallel API, allowing for simultaneous agent actions within the environment. This is critical for scenarios where agent interactions can occur concurrently.

2.2.3 Control Type:

The environment does not support manual control, emphasizing its design for automated agent interaction and decision-making.

2.2.4 Agents:

There are two agents in the environment, typically representing the two players in a chess game, labelled as player_0 and player_1.

Observation Space: The observation shape is defined as (8, 8, 111), which likely represents a structured format for the chess board where each piece and possible move is encoded within this multidimensional array.

The observation values are discrete, ranging between 0 and 1, indicating the presence or absence of specific pieces or conditions at each position on the board.

Import	`from pettingzoo.classic import chess_v6`
Actions	Discrete
Parallel API	Yes
Manual Control	No
Agents	`agents= ['player_0', 'player_1']`
Agents	2
Action Shape	Discrete(4672)
Action Values	Discrete(4672)
Observation Shape	(8,8,111)
Observation Values	[0,1]

Figure 1: Environment description

3 Methodology:

To teach our agents how to play chess, we are using two main RL methods: Deep Q Learning (DQN) and Monte Carlo Tree Search (MCTS). These methods help the agents figure out the best moves to make.

3.1 Deep Q Learning (DQN):

Deep Q Learning is a RL method where the agent learns to make decisions by approximating the Q-function, which measures the expected future rewards of taking a particular action in each state. In our implementation, the DQN agent interacts with the chess environment, observing states and taking actions based on its learned Q-values. Through a combination of experience replay and target network updates, the DQN agent iteratively refines its policy to improve its performance in the chess domain.

3.2 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search is a heuristic search algorithm that builds a tree of possible game states by simulating random playouts from each state and selecting the most promising actions based on the outcomes of these simulations. Our MCTS agent utilizes this tree structure to explore the state space of the chess environment, gradually focusing its search on the most promising trajectories to improve its decision-making capabilities. By balancing exploration and exploitation, the MCTS agent effectively learns to navigate the complexities of chess gameplay.

3.3 Training Dual Agents

To facilitate a comprehensive comparison of RL techniques in the chess domain, we train two distinct agents: one utilizing MCTS and the other employing DQN. This dual-agent setup enables us to evaluate the relative strengths and weaknesses of each approach, shedding light on their respective performances and applicability in different contexts. Through extensive experimentation and analysis, we aim to glean insights into the effectiveness of these RL methodologies in training agents for strategic decision-making tasks such as chess.

3.4 Choosing Moves:

In the gameplay implementation, the strategy for selecting moves involves two distinct agents with different decision-making approaches:

3.4.1 MCTS-Based Agent:

This agent uses the Monte Carlo Tree Search to determine its moves. By analysing the MCTS tree, it identifies the most advantageous move based on the child node with the highest win rate or the most visits, depending on the selection strategy

employed. This method ensures that the decisions are backed by a thorough evaluation of potential outcomes, reflecting the most strategic and promising actions as determined by the MCTS.

3.4.2 DQN-Based Agent:

Contrasting with the MCTS-based approach, the DQN-based agent selects moves based on a policy derived from a Deep Q-Network. This network evaluates the potential reward of each legal move from the current game state, guiding the agent to choose actions that maximize expected returns based on learned value functions. While the MCTS agent relies on exploration and simulation, the DQN agent operates on learned experiences, potentially offering more consistent performance against varying strategies.

3.4.3 Function Implementation Variability:

Additionally, the play_game() function allows for flexibility in how moves are selected. For instance, moves might be chosen randomly to introduce variability into the game or to test the resilience and adaptability of each strategy under less predictable conditions. This random selection feature is essential for ensuring that both the MCTS and DQN agents can handle a wide range of game scenarios, thereby enhancing their strategic depth and learning capabilities.

4 Description of RL Methods

The methods we have used are Deep Q learning and Monte-Carlo Tree Search.

4.1 Deep Q learning:

4.1.1 Definition

It is a reinforcement learning algorithm that combines Q-learning with deep neural networks to learn action–value functions in complex environments with high dimensional state spaces.

4.1.2 DQN algorithm

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 2: DQN Algorithm

4.1.3 Handling Illegal Moves:

The algorithm is modified such a way to handle illegal moves effectively. So, an attention mask is applied to the Q-values during the training phase. This mask assigns a large negative value to the Q-values of illegal moves, discouraging the agent from selecting them during the action selection phase. This ensures that the agent learns only valid strategies and moves within the legal confines of the game.

4.1.4 Practical Implementation:

Using the DQN algorithm, a neural network is trained to navigate the strategic complexities of chess in the multi-agent setting provided by PettingZoo. The environment offers a rich array of interactions and states, ideal for testing and enhancing the robustness of reinforcement learning algorithms.

```
super(DQN, self).__init__()
self.conv1 = nn.Conv2d(111, 32, kernel_size=3, stride=1, padding=1)
self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
self.fc1 = nn.Linear(128 * 8 * 8, 1024)
self.fc2 = nn.Linear(1024, 4672)
```

Figure 3: Neural Network in DQN Algorithm

4.2 Monte-Carlo Tree Search:

4.2.1 Definition

It is a probabilistic search algorithm used mainly in large decision spaces. It is commonly used in board games and adversarial environments.

4.2.2 MCTS Algorithm:

Algorithm 1 Original MCTS

function MCTSEARCH(s_0)
  create root node v_0 with state s_0.
  while within computational budget **do**
    $v_l \leftarrow$ *TreePolicy*(v_0)
    $\Delta \leftarrow$*Simulate*($s(v_l)$)
    Backprog(Δ, v_l)
  return *Bestchild*(v_0)

Figure 4: Neural Network in DQN Algorithm

4.2.3 Handling Legal Moves:

In the implementation of the Monte Carlo Tree Search (MCTS) within the game environment, the strategy for selecting legal moves is carefully designed to ensure optimal decision-making. Moves are selected based either on the results of the MCTS or randomly, depending on the logic defined in the 'play_game()' function.

5 Experiments

5.1 Training the DQN model:

We trained both the agents on DQN model for 500 episodes. So,the training results are:

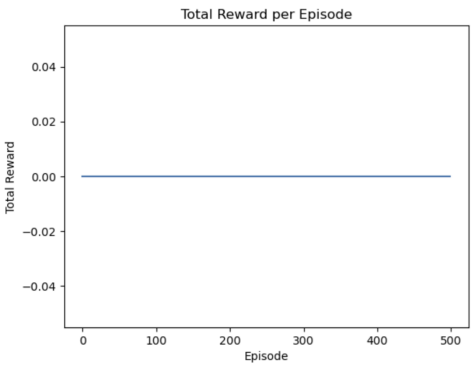

Figure 5: Training results

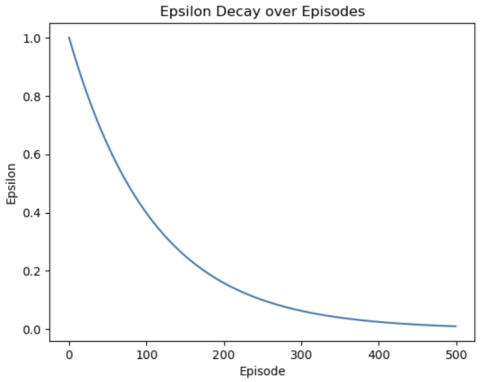

Figure 6: Epsilon curve

The total reward per episode is 0 because both agents are making the moves based on the same network which results in draw in most of the games.

Then the model is evaluated for 10 episodes where Player_1 takes random moves and PLayer_2 takes moves based on the DQN algorithm. The results are

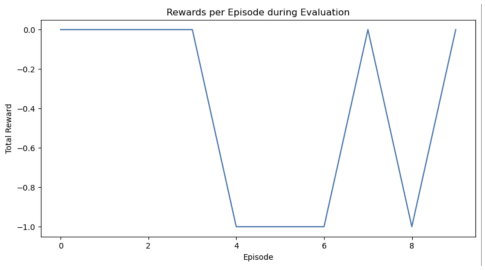

Figure 7: Evaluation results

The Player_2 can win some games which indicates that model learnt some of the best moves but unable to learn perfectly.

The main problem with the DQN model is it only estimates the next best move based on the current state but for the games like chess it should estimate the future moves and make the best possible move.

So, we decided to use MCTS algorithm.

5.2 MCTS algorithm:

We used the MCTS algorithm and used the DQN model to play the game. The Player_1 uses the DQN model to make the moves and the Player_2 uses the MCTS

algorithm to make the best move.

Then the Player_2 which uses MCTS algorithm is winning in most of the cases when played aganist Player_1 which ses DQN model to make its move.

6 Results:

The final board output after the Player_1 which uses the DQN model and the Player_2 which uses MCTS algorithm played when aganist each other.

Game 1: (iteration=100) Here the iterations is less in this case the DQN is performing better than MCTS.

Game 2: (iterations = 2000) When the depth of the tree i.e, iterations is increased the MCTS is outperforming DQN in most of the games.

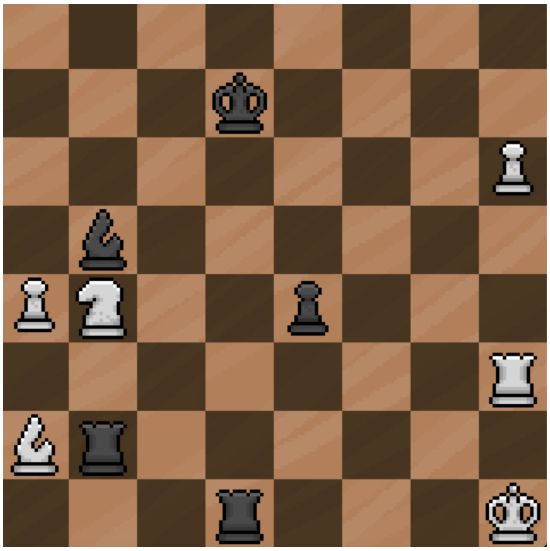

Figure 8: Black wins

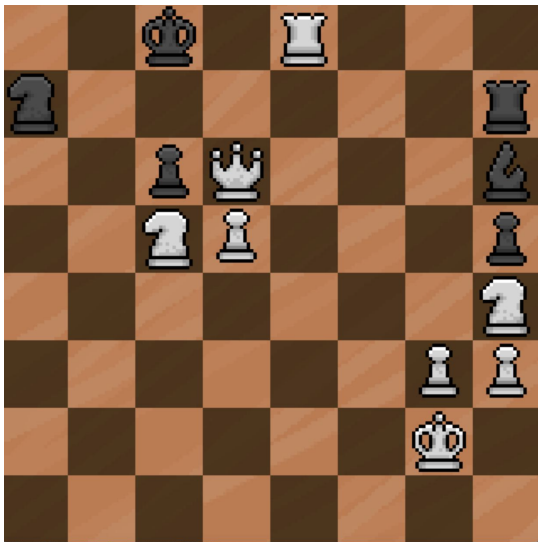

Figure 9: White wins

7 Observation:

The MCTS algorithm is performing better when compared to DQN algorithm. This because the DQN algorithm only considers the best move based on the current state where as the MCTS algorithm can simulate the future possible steps which is useful to make long-term strategic decisions. MCTS is mainly suitable for environments like chess with high branching factor and complex decision trees whereas DQN doesnot perform well and able to generalize better in these types of environments unless it is extensively trained. The DQN requires large number of

samples in these types of environments because every game may evolve differently but MCTS simulates the future states and takes best move which does not require any training data. The DQN algorithm may overfit the training data and struggles in the case of novel moves whereas MCTS adapts to every novel move and does not overfit.

8 Conclusion:

In this project, the DQN and MCTS are explored on complex environments like chess. We found that DQN can learn only basic strategies and legal moves in chess whereas MCTS outperform DQN due to its ability to simulate future moves and make the best moves for long-term outcomes. This suggests that in complex environments like chess with high branching factor MCTS outperforms DQN.

Contribution

gray!50 Member	Research	Code	Report
vduggemp	33	33	34
pvajha	34	33	33
vkaturu	33	34	33

9 References:

1. Playing Atari with Deep Reinforcement Learning
2. Human-level control through deep reinforcement learning
3. Monte Carlo Tree Search: A Review of Recent Modifications and Applications
4. Latex: For report