

1. Rules of MiniChess

MiniChess is a simplified version of chess played on a 5×5 board. The pieces and their movements are the same as in traditional chess, but the board is smaller, and some rules are adapted:

- Each player starts with 5 pawns and 5 pieces: king, queen, bishop, knight, and rook.
- The objective is to checkmate the opponent's king.
- Players alternate turns, with gold and silver sides.
- Pawns promote to queen automatically when they reach the other side.
- Stalemates and repetitions are treated as draws.



2. Project Objectives

The objective of this project is to develop a reinforcement learning (RL) agent that can play the game of MiniChess using the provided `MiniChessEnv` environment in the project file minichess_env.py. A sample random agent is provided in the file MiniChessRandomAgent.py. This would serve both as an example of what is required of an agent, and can be used as a baseline to train or test. The goal is to model the game using an appropriate RL architecture (e.g., Q-learning, policy gradients, actor-critic) and train the agent to play competitively.

3. Environment Overview

The MiniChess environment is implemented in Python and follows the Gymnasium API design. The game is played on a 5×5 board represented as a 2D NumPy array:

```
self.board # shape: (5, 5), dtype: int
```

Each cell of the board holds an integer corresponding to a piece or an empty square:

Value	Meaning	Color
0	Empty square	-
1–6	Pawn to King	Silver (Black)
7–12	Pawn to King	Gold (White)

The specific integer codes for each piece are as follows:

Piece Type	Silver Code	Gold Code
Pawn	1	7
Knight	2	8
Bishop	3	9
Rook	4	10
Queen	5	11
King	6	12

The environment exposes:

- self.board: current board state
- self.current_player: 1 (Gold) or -1 (Silver)
- get_legal_moves(): returns a list of valid actions for the current player
- step(action): applies a legal move and switches turns
- render(): displays the board with piece images and highlights

Actions are integers from 0 to 624, representing all possible from-to square combinations. You can decode an action like this:

```
from_idx = action // 25 # 0-24
to_idx = action % 25 # 0-24
from_row, from_col = divmod(from_idx, 5)
to_row, to_col = divmod(to_idx, 5)
```

4 Agent Interface Requirements

Each student must submit a Python file (e.g. agent_john.py) that defines a class named Agent with the following structure:

```
class Agent:
    def __init__(self):
        # (Optional) Load a model or initialize state
        pass

    def get_action(self, board, player):
        # Inputs:
        # - board: a 5x5 NumPy array of piece integers (see above)
        # - player: 1 (Gold) or -1 (Silver), indicating whose turn it is

        # Output:
        # - a single integer action in [0, 624], representing a legal move
```

Agents must only return moves that are valid in the current board state. A helper to retrieve legal moves is available via:

```
env = MiniChessEnv()
env.board = board.copy()
env.current_player = player
legal_moves = env.get_legal_moves()
```

If an agent attempts an illegal move or throws an error during execution, it will forfeit the game in the competition. Therefore, agents must be robust and self-contained.

If your agent depends on a trained model (e.g. PyTorch or TensorFlow), you must include the model file and loading logic within `__init__()`.

5. Deliverable Format

Your final deliverable code should be a .py file that contains your agent. The match_runner.py file shows how an agent will be accessed for the evaluation and competition. The code below demonstrates a game play between two random agents.

```
from match_runner import load_agent, play_game
a1 = load_agent("MiniChessRandomAgent.py")
a2 = load_agent("MiniChessRandomAgent.py")
play_game(a1,a2)
```

6. The Random Agent

The random agent in MiniChessRandomAgent.py is provided as a baseline for debugging and benchmarking. It selects a random legal move from the list returned by the environment's `get_legal_moves()` method. This ensures all selected moves are valid, but the agent has no learning or strategy. Students can use this as a starting point for interfacing with the environment and validating their implementations.

7. Project Evaluation

The final grade for the project will be based on three components:

1. Self-Play Evaluation (30%)

Each agent will play 10 self-play games against itself. Evaluation will focus on:

- Whether all moves made are legal
- How few of the games result in draws (prefer strategic resolution)

2. Round Robin Tournament and Playoffs (30%)

Each agent will compete in a round-robin tournament against all others, playing both gold and silver roles equally. Scoring will be based on wins, losses, and draws. The top four agents will advance to semi-finals, playing two games each (switching colors). If a semi-final ends in a tie, the agent with the higher round-robin score proceeds. The finals will be two games; if tied, both agents share the win.

3. Final Report (40%)

Students must submit a formal report structured like a research paper with the following sections:

- Introduction
- Related Work
- Reinforcement Learning Model (clearly described in mathematical terms)
- Training Methodology (detailed description including training plots)
- Challenges and Lessons Learned
- Case Studies of Selected Self-Play Games
- Conclusion and Future Work

The report must include analysis, screenshots of gameplay, and discussion of agent behavior.