Developing an Intelligent Agent for Nine Men's Morris Using Deep Q-Networks

Author Name
Institution Name
author.email@institution.edu

April 15, 2025

Abstract

This report presents the development of an artificial intelligence agent for Nine Men's Morris using Deep Q-Networks (DQN). A Python-based game environment was implemented, and the DQN agent was trained against a random opponent. The methodology covers game mechanics, DQN architecture, and difficulty settings for human interaction. Extensive evaluation over 700 episodes reveals a low win rate (1–6%) against a random opponent, indicating challenges in strategic learning. Analysis identifies reward structure and training limitations as key factors, offering insights for improvement. This work provides a foundation for applying deep reinforcement learning to combinatorial games.

Contents

1 Introduction

Nine Men's Morris, a two-player strategy game of ancient origin, requires players to form mills—three aligned pieces on a 24-point board—across placement, movement, and flying phases. Its strategic depth makes it suitable for reinforcement learning (RL) exploration. This study employs Deep Q-Networks (DQN) [?] to create an agent capable of playing competitively and engaging human opponents at varied difficulty levels. Objectives include developing a robust game environment, training a DQN agent, designing rewards, and evaluating performance, while advancing practical knowledge of deep RL using PyTorch.

2 Methodology

This section outlines the game environment, DQN framework, and difficulty calibration strategies.

2.1 Game Environment

The `NineMensMorrisEnv` class, implemented in Python, encapsulates game rules and dynamics.

2.1.1 Board and State Representation

The 24-point board, arranged in three concentric squares, is represented as a NumPy array:

- 0: Empty point.
- 1: Player 1 piece.
- 2: Player 2 piece (AI).

The state, normalized to [0, 0.5, 1.0], is the neural network input. Figure 1 shows the board layout.

`nmm_board.png`

Figure 1: Nine Men's Morris board with 24 points, numbered 0–23.

2.1.2 Game Phases and Rules

Gameplay spans:

1. **Placement**: Place nine pieces on empty points.

2. **Movement**: Move pieces to adjacent empty points.
3. **Flying**: Move to any empty point with three pieces.

Mechanics include:

- `get_valid_actions()`: Lists legal moves (indices or tuples).
- `check_mill()`: Detects mills, triggering piece removal.
- `remove_piece()`: Prioritizes non-mill pieces.
- `check_winner()`: Loss occurs with fewer than three pieces or no moves.

2.1.3 Action Space

Actions are phase-dependent:

- Placement: Indices 0–23.
- Movement/Flying: Tuples `(from_pos, to_pos)`.

The environment ensures rule adherence.

2.2 Deep Q-Network Design

DQN approximates $Q(s, a; \theta)$ for large state spaces [**?**].

2.2.1 Neural Network Architecture

The `DQNetwork` in PyTorch comprises:

- Input: 24 neurons.
- Hidden: Two 128-neuron layers with ReLU.
- Output: 24 Q-values.

```
class DQNetwork(nn.Module):
    def __init__(self, state_size=24, action_size=24):
        super(DQNetwork, self).__init__()
        self.fc1 = nn.Linear(state_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, action_size)
        self.relu = nn.ReLU()
    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Listing 1: DQN Model in PyTorch

2.2.2 DQN Mechanisms

- **Experience Replay**: Stores and samples transitions $(s, a, r, s', \text{done})$.
- **Target Network**: Provides stable targets, updated every 10 episodes.
- **Epsilon-Greedy**: ϵ decays from 1.0 to 0.1.

Loss is minimized via:

$$\mathcal{L}(\theta) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right]$$

using Adam.

2.2.3 Action Selection Across Phases

The 24 Q-values are mapped:

- **Placement**: Highest valid index.
- **Movement/Flying**: Highest Q-value via `(from_pos + to_pos) % 24`.

This heuristic ensures legality but simplifies movement.

2.3 Difficulty Calibration

The `select_action` function offers:

- **Easy**: 70% random, mill priority.
- **Moderate**: 30% random, blocks mills, DQN with $\epsilon = 0.2$.
- **Difficult**: Strategic checks, DQN with $\epsilon = 0.05$.

3 Experimentation

This section covers training, rewards, and computational setup.

3.1 Training Protocol

The `train_agent()` function executes:

1. Initialize environment and agent.
2. For 20,156 episodes:
   (a) Reset to state s.
   (b) Select action via ϵ-greedy.
   (c) Execute action, store transition.
   (d) Random opponent plays.
   (e) Train on mini-batch.
3. Decay ϵ, update target network, save model.

3.2 Reward Design

Rewards include:

- **Sparse**: $+10.0$ (win), -10.0 (loss).
- **Dense**:
  - $+2.0$: Piece removed.
  - $+1.0$: Mill formed.
  - $+0.05$: Central position.
  - $+0.1 \times$ count: Potential mills.
  - $+0.5 \times$ count: Blocked opponent mills.
  - $+0.2 \times$ piece_difference: Piece advantage.
  - -0.05: Per move.

3.3 Computational Setup

Training used a desktop with PyTorch. Hyperparameters:

- $\gamma = 0.95$, learning rate = 0.001, batch size = 32.
- Replay buffer = 2000, target update every 10 episodes.
- ϵ: 1.0 to 0.0997, decay = 0.995.

3.4 Ethical Considerations

Training was resource-efficient, adhering to ethical AI principles for fair gameplay.

4 Analysis and Results

This section evaluates performance based on seven evaluation runs, each comprising 100 episodes against a random opponent, using the model at episode 20,156 ($\epsilon = 0.0997$).

4.1 Performance Evaluation

Table 1 summarizes the evaluation results.

Table 1: Evaluation Results Against Random Opponent (100 Episodes per Run)

Run	Win Rate (%)	Average Reward	Average Episode Length (Moves)
1	4.00	-14.24	190.22
2	6.00	-13.64	179.06
3	5.00	-11.77	190.11
4	5.00	-13.34	191.65
5	2.00	-15.42	178.82
6	3.00	-18.80	177.37
7	1.00	-18.57	174.99
Mean	3.71	-15.11	183.17
Std. Dev.	1.70	2.62	6.83

Across 700 episodes:

- **Win Rate**: Averaged 3.71% (range: 1–6%), with 26 wins, 674 losses, and no draws.
- **Average Reward**: -15.11 (range: -18.80 to -11.77), reflecting frequent losses (-10.0) compounded by per-move penalties (-0.05).
- **Episode Length**: Averaged 183.17 moves (range: 174.99–191.65), indicating prolonged games despite losses.

The agent struggled significantly, losing 96.29% of games on average. However, qualitative observations suggest it learned to form mills and occasionally block opponent moves, though inconsistently.

4.2 Performance Analysis

The low win rate against a random opponent is unexpected, given 20,156 training episodes. Possible contributing factors include:

- **Reward Structure**: Dense rewards (e.g., +1.0 for mills, +0.05 for central positions) may overemphasize intermediate goals, diluting focus on winning (+10.0). The -0.05 per-move penalty accumulates in long games, skewing total rewards negatively.

- **Training Opponent**: Exclusive training against a random opponent may have limited strategic depth, failing to expose the agent to robust counter-strategies.
- **Action Mapping**: The heuristic for movement phases (`(from_pos + to_pos) % 24`) may misalign Q-values with optimal moves, reducing effectiveness.
- **State Representation**: Excluding phase or piece counts may hinder the agent's ability to contextualize actions.
- **Exploration**: An $\epsilon = 0.0997$ during evaluation introduces 10% randomness, potentially disrupting learned policies.

The variance in rewards (std. dev. 2.62) and win rates (std. dev. 1.70) suggests instability, possibly due to insufficient convergence or hyperparameter sensitivity.

4.3 Insights Gained

- **RL Dynamics**: Understood state-action-reward interplay and exploration challenges.
- **DQN Stability**: Confirmed the role of replay and target networks.
- **Reward Design**: Learned the need for balanced sparse and dense rewards.
- **Action Challenges**: Recognized limitations of heuristic mappings.
- **Evaluation Needs**: Highlighted the importance of diverse opponents.

4.4 Limitations

- Overreliance on random opponent training.
- Inefficient movement action mapping.
- Simplified state representation.
- Potential reward misalignment.

4.5 Future Directions

- Refine rewards to prioritize winning (e.g., reduce per-move penalty).
- Train against varied opponents or self-play [**?**].
- Enhance action representation (e.g., explicit tuple outputs).
- Include phase and piece counts in state.
- Explore advanced RL methods (e.g., Double DQN [**?**]).

4.6 Conclusion

The DQN agent for Nine Men's Morris, while demonstrating basic strategic behaviors, achieved a low win rate (3.71%) against a random opponent, highlighting challenges in reward design, training setup, and action mapping. The project offers valuable lessons in deep RL application, establishing a foundation for future enhancements to achieve competitive performance.

References