MCTS - Connect four

Markov Decision Processes and Reinforcement Learning

Samuele Angheben

Trento University

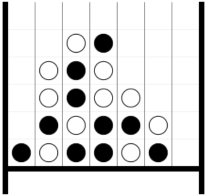
February 8, 2025

Overview

The objective of this project is to explore different reinforcement learning algorithm on the game **Connect Four**, starting from very simple mcts implementation to alphazero.

Algorithms:

- **Naive MCTS:** A basic Monte Carlo Tree Search approach without learning.
- **REINFORCE with Baseline:** A policy gradient method incorporating a baseline.
- **AlphaZero:** A self-play RL algorithm combining MCTS with deep learning.



Connect Four is a two-player game played on a vertical 7-column, 6-row grid where players take turns dropping colored discs into the slots, aiming to connect four discs of their color horizontally, vertically, or diagonally.

Properties

- **Deterministic:** Outcome fully determined by player actions. $S_{t+1}, R_t = p(S_t, a_t)$
- **Episodic:** Sequence of actions leading to a terminal state. Episode: $\{(S_1, a_1), \ldots, (S_T, a_T)\}$
- **Two-Player:** Players 1 and 2 alternate moves, strategies π_1 and π_2.
- **Zero-Sum:** Rewards sum to zero. $R_1 = -R_2$. Terminal rewards: Win (+1/-1), Draw (0/0)
- **Perfect Information:** Both players have complete knowledge of the current state.

- **Trajectory Length:** Max 42 moves (7 columns x 6 rows).
- **State Space:**
  - Naive: $3^{42} \approx 1.1 \times 10^{20}$
  - Improved: $\approx 4.5 \times 10^{12}$ states
  - Storage: 4.5×10^{12} states $\times$ 8 bytes/state $=$ 36 TB (64 bit state representation)
- **Tabular Methods:** Infeasible due to large state space.
- **Solvability:** Solved. Player 1 can always win with optimal play (Wikipedia). No readily available solvers for testing.

Framework: JAX for computational efficiency and flexibility, enabling generalization to various games. Abstraction of MCTS, network training, and game dynamics for modularity and scalability.

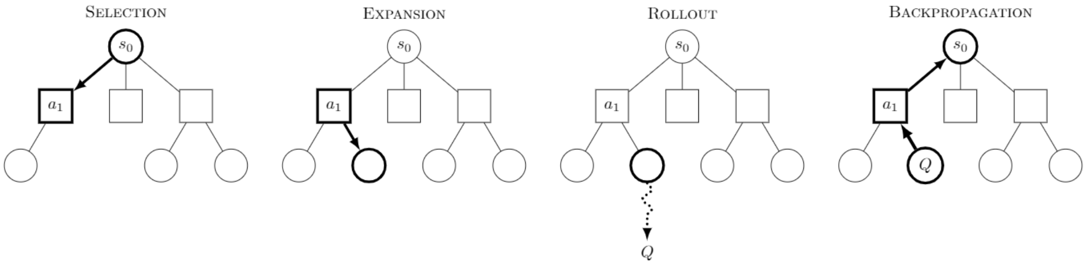
JAX Features:

- JIT Compilation
- GPU/TPU Acceleration
- Efficient Vectorization (vmap)
- Automatic Parallelization (pmap)
- Functional Programming (Pure Functions)

Libraries

- `mctx`: JAX-native MCTS implementations.
- `pgx`: Game environment simulation, diverse games, fast parallelization.
- `Haiku`: Neural network building and training in JAX.
- `optax`: Gradient processing and optimization.

Monte Carlo Tree Search is a decision-making algorithm that explores a decision space by constructing a search tree through repeated simulations. Given a computation budget it balances exploration and exploitation to refine the root node policy.



To perform a MCTS search using the MCTX library, two main functions must be provided:

RootFnOutput(state):

- Specifies the representation of the root state.
- Returns the prior logits and the estimated value of the root state.

recurrent_fn(state, action):

- Encapsulates the environment dynamics.
- Returns the reward, discount factor, and for the new state, the prior logits and value.

The search returns **action_weight**, representing the updated root node policy.

Negative Discount Factor

In alternating-turn games, a negative discount factor ($\gamma = -1$) is used to invert value estimates between players:

$$V(S_t) = r_t + \gamma \cdot V(S_{t+1}) = r_t + (-1) \cdot V(S_{t+1})$$

- S_t: Current state (Player 1's turn).
- S_{t+1}: Next state (Player 2's turn), where the value $V(S_{t+1})$ represents the opponent's optimal value.

This inversion maintains the zero-sum property by ensuring that a high value for one player corresponds to a low value for the opponent.

The neural network is a custom **ResNet**, a convolutional network with residual connections and batch normalization. It has two output heads:

$$l, v' = \text{ResNet}_\theta(S_t)$$

1. **Policy Head:** Produces a probability distribution over possible actions:

$$\pi(a \mid S_t) \approx \hat{\pi}(a \mid S_t) = \text{softmax}(l)$$

2. **Value Head:** Outputs a scalar value approximating the value function:

$$V(S_t) \approx \hat{V}(S_t) = v'$$

Description

Naive implementation of Monte Carlo Tree Search:

- **no learning**
- MCTS
  - policy: uniform
  - value: single random rollout

function $\textsc{RecurrentFn}$(s, a)

$s', reward \leftarrow env.step(s, a)$

$(logit', value') \leftarrow uniform\_prior, random\_rollout(s')$

$discount \leftarrow -1.0$

return s', logit', value', reward, discount

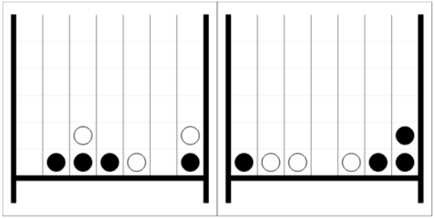
function $\textsc{RunMCTS}(s)$

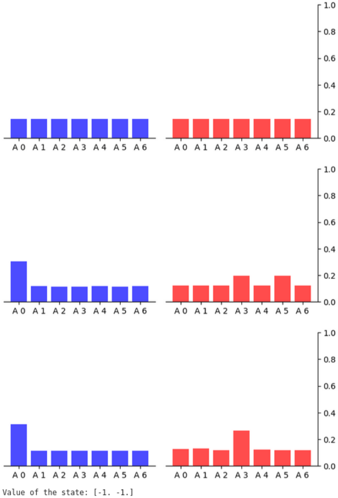
$root = (logit, value) \leftarrow uniform\_prior, random\_rollout(s)$

$policy\_output \leftarrow MCTS(root, RecurrentFn, num\_simulation)$

return $\arg\max_a policy\_output$

Example Policy_Out





Description

Reinforce baseline algorithm:

- Policy based algorithm
- no MCTS
- on policy, batch
- neural network function approximation

Pseudocode


```
Initialize: Neural Network, Optimizer, Replay Buffer

Main Loop:
    for iteration in 1..MaxIterations:
        # Self-play Phase
        Collect Games:
            for N games:
                def step_fn(s):
                    pi_nn(a|s) <- nn_forward(s)
                    action <- epsilon-greedy (pi(a|s))

                    s', r <- env_step(s, a)

                    return (s, a, s', r, discount=-1),
                            s'

                s_init <- env.reset()
                data <- jax.scan(step_fn, s_init)

                samples <- compute_loss_input(data) # cumulative rewards
                Save samples to Replay Buffer # (s_t, a_t, G_t:value target)


        Shuffle samples and make minibatches (Replay Buffer)





        # Training Phase
        Train Network:
            batch ~ Replay Buffer # sample (s_t, a_t, G_t)
            compute_loss:
                V_nn, pi_nn(a|s_t) <- nn_forward(s_t)
                A(s_t, a_t) <- G_t - V_nn(s_t) # compute advantage
                policy_loss <- -mean[log(pi_nn(a|s_t) * A(s_t, a_t)]
                value_loss  <- MSE(V_nn(s_t, G_t))
        entropy_loss <- entropy(pi_nn(S_t))

                loss <- policy_loss + value_loss + entropy_loss

            Compute gradient (loss)
            Update model parameters

        # Saving Phase (periodic)
        if iteration % saving_interval == 0:
            Save checkpoint
```

Total Loss:

$$L = \mathcal{L}_{\text{policy}} + \mathcal{L}_{\text{value}} + 0.1 \times \mathcal{L}_{\text{entropy}}$$

Where:

$$\mathcal{L}_{\text{policy}} = -\mathbb{E}\left[\log \hat{\pi}(a_t \mid S_t) \cdot A(S_t, a_t)\right]$$

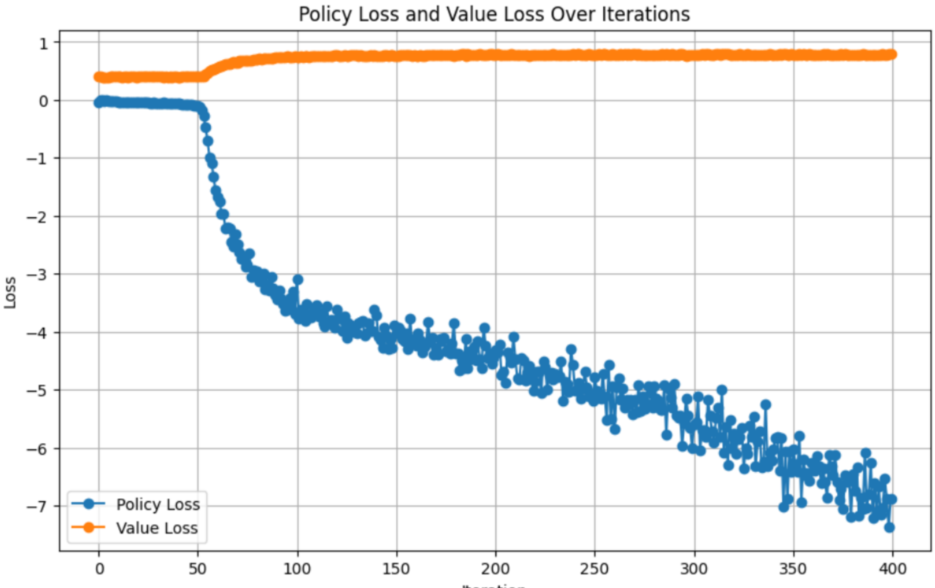
$$A(S_t, a_t) = G_t - \hat{V}(S_t)$$

$$\mathcal{L}_{\text{value}} = \mathbb{E}\left[\left(\hat{V}(S_t) - G_t\right)^2\right]$$

$$\mathcal{L}_{\text{entropy}} = -\mathbb{E}\left[\sum_a \pi(a \mid S_t) \log \pi(a \mid S_t)\right]$$

Gradient Clipping: ensures that the gradients do not exceed a specified threshold, preventing exploding gradients.

Loss plot


Policy Loss and Value Loss Over Iterations
Loss
1
0
−1
−2
−3
−4
−5
−6
−7
0
50
100
150
200
250
300
350
400
Policy Loss
Value Loss

Even with loss improvements, divergence still occurs.

The Deadly Triad:

Bootstrapping	×
Function Approximation	✓
Off-Policy (No Batch)	×

Multiple Columns

Heading

1. Statement
2. Explanation
3. Example

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer lectus nisl, ultricies in feugiat rutrum, porttitor sit amet augue. Aliquam ut tortor mauris. Sed volutpat ante purus, quis accumsan dolor.

Treatments	Response 1	Response 2
Treatment 1	0.0003262	0.562
Treatment 2	0.0015681	0.910
Treatment 3	0.0009271	0.296

Table: Table caption

Theorem

Theorem (Mass–energy equivalence)

$E = mc^2$

Figure

Uncomment the code on this slide to include your own image from the same directory as the template .TeX file.

Citation

An example of the `\cite` command to cite within the presentation:

This statement requires citation [**?**].

References

The End