MCTS - Connect four

Markov Decision Processes and Reinforcement Learning

Samuele Angheben

Trento University

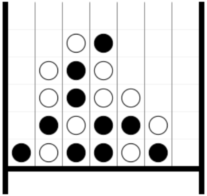
February 9, 2025

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 provides a high-performance framework for computational efficiency and flexibility, allowing for scalable and efficient development of machine learning algorithm.

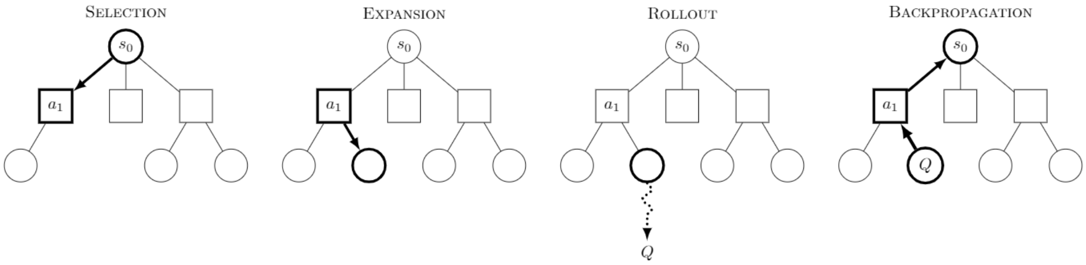
JAX Features:

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

Libraries with links

- `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:

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

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

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

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

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

Description

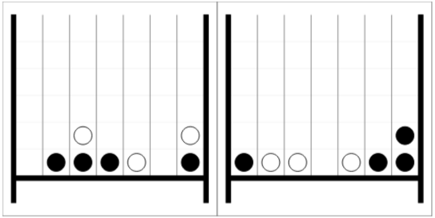
Naive implementation of Monte Carlo Tree Search:

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

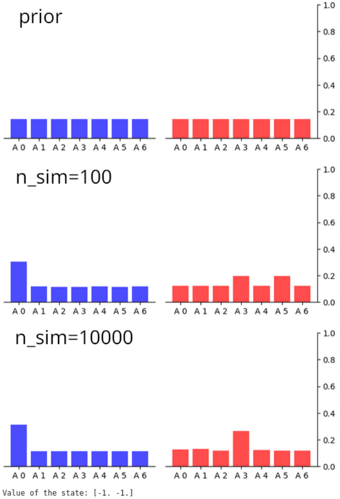
Pseudocode MCTS Inference


```
def RecurrentFn(s, a)
    s', reward <- env_step(s, a)
    V, pi <- random_rollout(s'), uniform_prior
    discount <- -1.0
    return s’, pi, V, reward, discount

def RunMCTS(s)
    root = (V, pi(a|s)) <- random_rollout(s), uniform_prior
    policy_output <- MCTS(root, RecurrentFn, num simulation)
    return arg max_a policy_output
```



Value = [-1, -1]


Policy output
prior
n_sim=100
n_sim=10000
A 0 A 1 A 2 A 3 A 4 A 5 A 6
1.0
0.8
0.6
0.4
0.2
0.0
Value of the state: [-1. -1.]

Description

Reinforce baseline algorithm:

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

```
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):
                    root = V_nn, pi_nn(a|s) <- nn_forward(s)
                    run MCTS(root, num_simulation):
                        s', reward <- env
                        V', pi <- nn_forward

                    a <- MCTS.policy_out.action

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

                    return (s, policy_out, 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, pi_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_t|s_t) * A(s_t, a_t)]
                value_loss  <- MSE(V_nn(s_t), G_t)
                entropy_loss <- entropy(pi_nn)

                loss <- policy_loss + value_loss + 0.1 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:

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

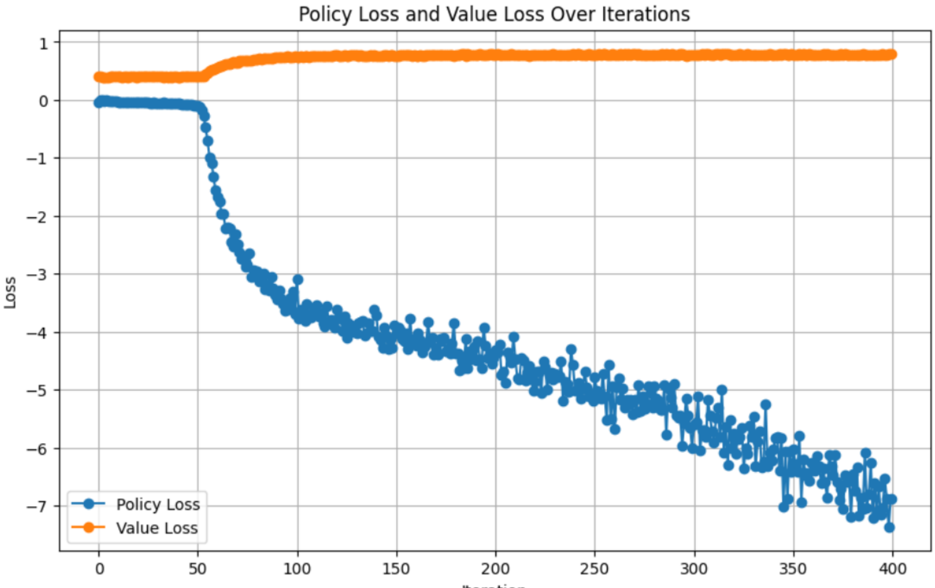
$$A(S_t, a_t) = G_t - \hat{V}(S_t)$$

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

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

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

Loss plot


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

Even with loss improvements, divergence still occurs.

The Deadly Triad:

Bootstrapping	$\times$
Function Approximation	$\checkmark$
Off-Policy (No Batch)	$\times$

Alphazero algorithm

- MCTS
  - policy: neural network policy head
  - value: neural network value head
- neural network function approximation

```
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):
                    root = V_nn, pi_nn(a|s) <- nn_forward(s)
                    run MCTS(root, num_simulation):
                        s', reward <- env
                        V', pi <- nn_forward

                    a <- MCTS.policy_out.action

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

                    return (s, policy_out, 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, pi_t, G_t:value target)


        Shuffle samples and make minibatches (Replay Buffer)





        # Training Phase
        Train Network:
            batch ~ Replay Buffer # sample (s_t, policy_out_t, G_t)
            compute_loss:
                V_nn, pi_nn(a|s_t) <- nn_forward(s_t)
                policy_loss <- cross_entropy(pi_nn(a|s_t), policy_out)
                value_loss  <- MSE(V_nn(s_t, G_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:

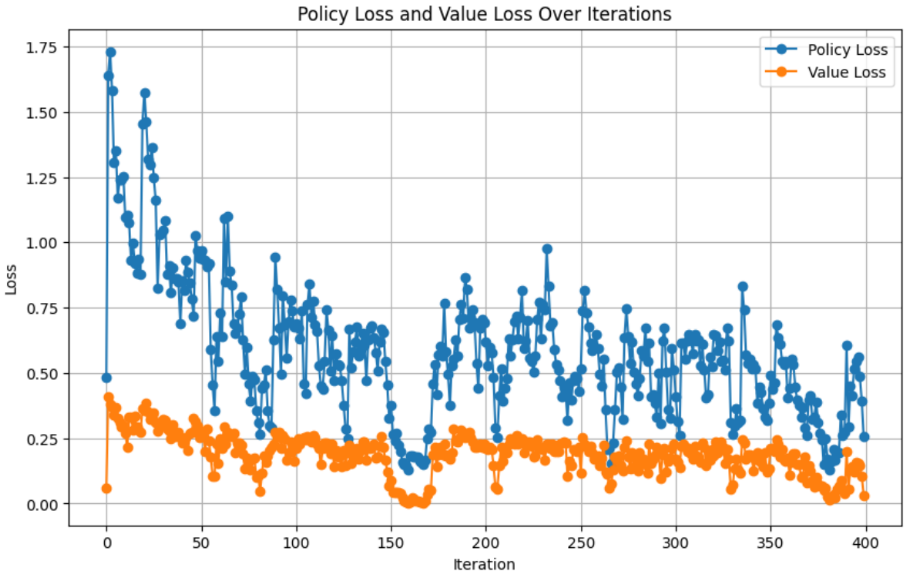
$$L = \mathcal{L}_{\text{policy}} + \mathcal{L}_{\text{value}}$$

Where:

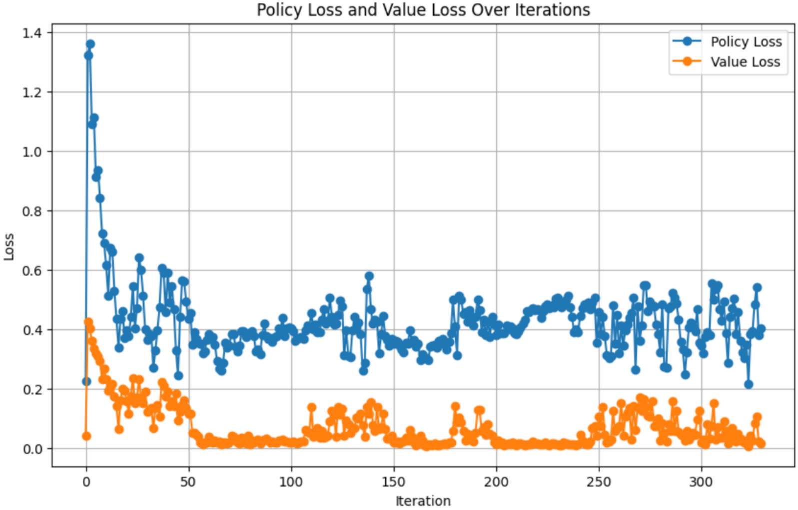
$$\mathcal{L}_{\text{policy}} = -\mathbb{E}_{a \sim policy\_out} [\log \hat{\pi}(a)]$$

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

Loss plot


Policy Loss and Value Loss Over Iterations
Policy Loss
Value Loss
Loss
Iteration
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
50
100
150
200
250
300
350
400

With better training: increase num. MCTS simulation, batch size, neural network



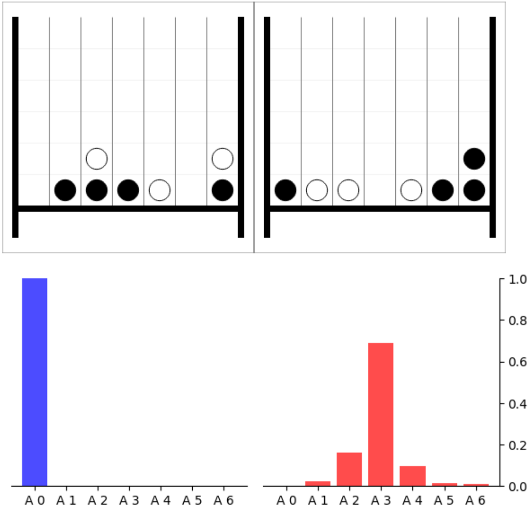
Inference MCTS vs One-shot

```
def RecurrentFn(s, a)
    s', reward <- env_step(s, a)
    V, pi <- nn_forward(s')
    discount <- -1.0
    return s', pi, V, reward, discount

def RunMCTS(s)
    root = (V, pi(a|s)) <- nn_forward(s)
    policy_output <- MCTS(root, RecurrentFn, num simulation)
    return arg max_a policy_output
```

```
def RunOneShot(s)
    pi_nn(a|s) <- nn_forward(s)
    return arg max_a pi_nn(a|s)
```

Example Network forward




```
Array([-0.93581444,  0.85516936], dtype=float32)
```

Notebook games and demo play

Future Experiments

- hyper-parameter tuning for better training
- longer training
- network architecture
  - increase capacity
  - transformer architecture
- MCTS: test different selection algorithm

References

Danihelka, I., Guez, A., Schrittwieser, J., and Silver, D. (2022).
Policy improvement by planning with gumbel.
https://www.openreview.net/forum?id=bERaNdoegnO.
Accessed: 2025-02-07.

Koyamada, S., Okano, S., Nishimori, S., Murata, Y., Habara, K., Kita, H., and Ishii, S. (2024).
Pgx: Hardware-accelerated parallel game simulators for reinforcement learning.

Silver, D., Huang, A., Maddison, C. J., and et al. (2016).
Mastering the game of go with deep neural networks and tree search.
Nature, 529(7587):484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2017a).
Mastering chess and shogi by self-play with a general reinforcement learning algorithm.

Silver, D., Schrittwieser, J., Simonyan, K., and et al. (2017b).
Mastering the game of go without human knowledge.
Nature, 550:354–359.

Thank you for your attention!