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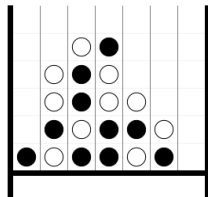
Connect Four is a two-player game played on a vertical 7-column, 6-row grid where players take turns dropping colored discs into the slots, aiming to connect four discs of their color horizontally, vertically, or diagonally.

- **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)

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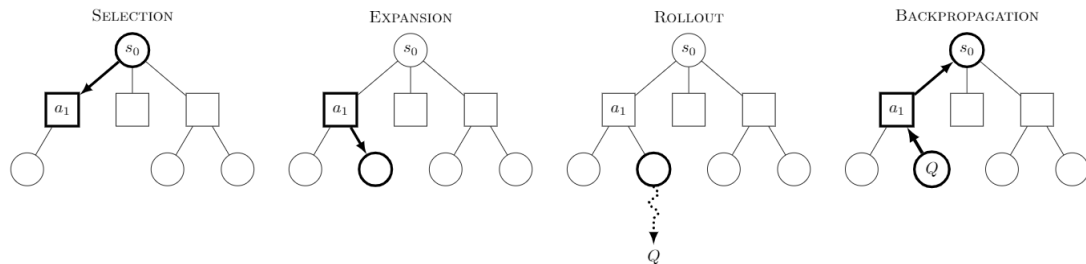
To perform a MCTS search using the MCTX library, two main functions must be provided:

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.

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:

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

Naive implementation of Monte Carlo Tree Search:

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

```
61
62 def RecurrentFn(s, a)
63     s', reward <- env_step(s, a)
64     V, pi(a|s) <- random_rollout(s), uniform_prior
65     discount <- -1.0
66     return s', pi, V, reward, discount
67
68 def RunMCTS(s)
69     root = (V, pi(a|s)) <- random_rollout(s), uniform_prior
70     policy_output <- MCTS(root, RecurrentFn, num simulation)
71     return arg max_a policy_output
72
```
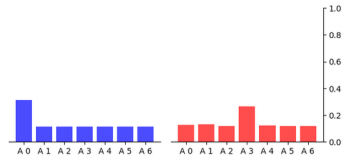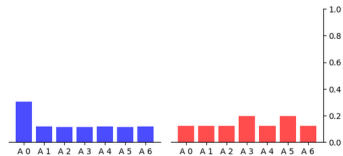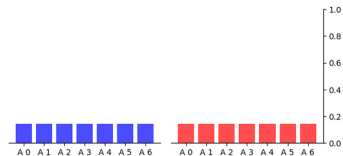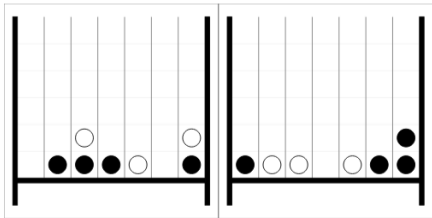
Value of the state: [-1. -1.]

Reinforce baseline algorithm:

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

```
1  Initialize: Neural Network, Optimizer, Replay Buffer
2
3  Main Loop:
4      for iteration in 1..MaxIterations:
5          # Self-play Phase
6          Collect Games:
7              for N games:
8                  def step_fn(s):
9                      pi_nn(a|s) <- nn_forward(s)
10                     a <- epsilon-greedy (pi(a|s))
11
12                     s', r <- env_step(s, a)
13
14                     return (s, a, s', r, discount=-1),
15                             s'
16
17                 s_init <- env.reset()
18                 data <- jax.scan(step_fn, s_init)
19
20                 samples <- compute_loss_input(data) # cumulative rewards
21                 Save samples to Replay Buffer # (s_t, a_t, G_t:value target)
22
23
24          Shuffle samples and make minibatches (Replay Buffer)
25
```

```
33
34
35
36          # Training Phase
37          Train Network:
38              batch ~ Replay Buffer # sample (s_t, a_t, G_t)
39              compute_loss:
40                  V_nn, pi_nn(a|s_t) <- nn_forward(s_t)
41                  A(s_t, a_t) <- G_t - V_nn(s_t) # compute advantage
42                  policy_loss <- -mean[log(pi_nn(a|s_t) * A(s_t, a_t)]
43                  value_loss  <- MSE(V_nn(s_t, G_t))
44                  entropy_loss <- entropy(pi_nn(S_t))
45
46                  loss <- policy_loss + value_loss + 0.1 entropy_loss
47
48              Compute gradient (loss)
49              Update model parameters
50
51          # Saving Phase (periodic)
52          if iteration % saving_interval == 0:
53              Save checkpoint
54
55
56
57
```

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.

Policy Loss and Value Loss Over Iterations
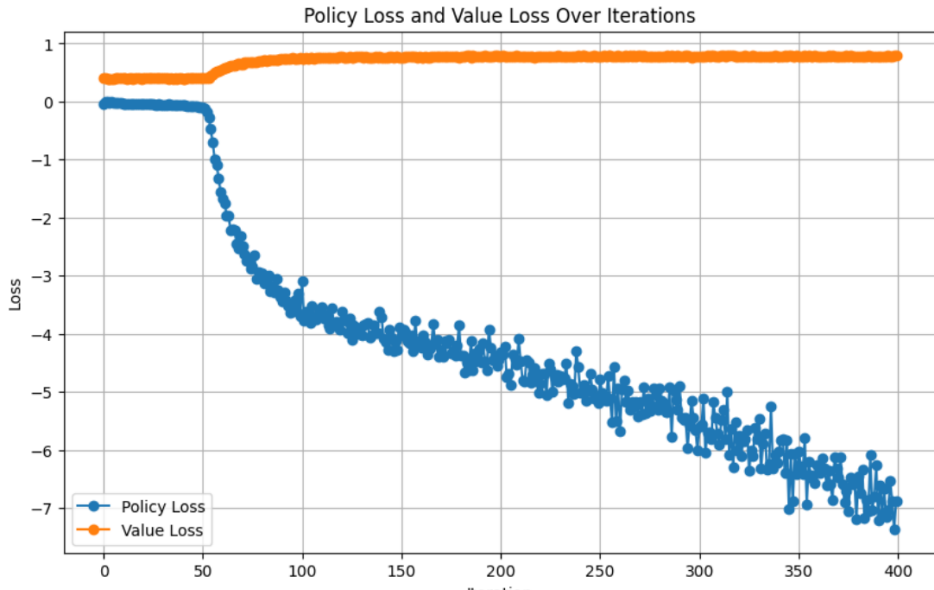
Even with loss improvements, divergence still occurs.

The Deadly Triad:

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

```
1   Initialize: Neural Network, Optimizer, Replay Buffer                     28
2                                                                            29
3   Main Loop:                                                               30
4       for iteration in 1..MaxIterations:                                   31
5           # Self-play Phase                                                32          # Training Phase
6           Collect Games:                                                   33          Train Network:
7               for N games:                                                 34              batch ~ Replay Buffer # sample (s_t, policy_out_t, G_t)
8                   def step_fn(s):                                          35              compute_loss:
9                       root = V_nn, pi_nn(a|s) <- nn_forward(s)             36                  V_nn, pi_nn(a|s_t) <- nn_forward(s_t)
10                      run MCTS(root, num_simulation):                      37                  policy_loss <- cross_entropy(pi_nn(a|s_t), policy_out)
11                          s', reward <- env                                38                  value_loss  <- MSE(V_nn(s_t, G_t))
12                          V', pi <- nn_forward                             39
13                                                                           40                  loss <- policy_loss + value_loss + entropy_loss
14                      a <- MCTS.policy_out.action                          41
15                                                                           42              Compute gradient (loss)
16                      s', r <- env_step(s, a)                              43              Update model parameters
17                                                                           44
18                      return (s, policy_out, s', r, discount=-1),.         45          # Saving Phase (periodic)
19                          s'                                               46          if iteration % saving_interval == 0:
20                                                                           47              Save checkpoint
21                  s_init <- env.reset()                                    48
22                  data <- jax.scan(step_fn, s_init)                        49
23                                                                           50
24                  samples <- compute_loss_input(data) # cumulative rewards 51
25                  Save samples to Replay Buffer # (s_t, pi_t, G_t:value target) 52
26                                                                           53
27                                                                           54
28              Shuffle samples and make minibatches (Replay Buffer)         55
29                                                                           56
```
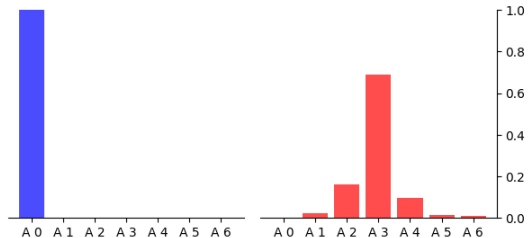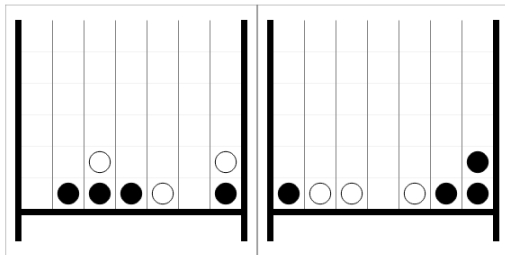
```
Array([-0.93581444,  0.85516936], dtype=float32)
```

References

Thank you for your attention!