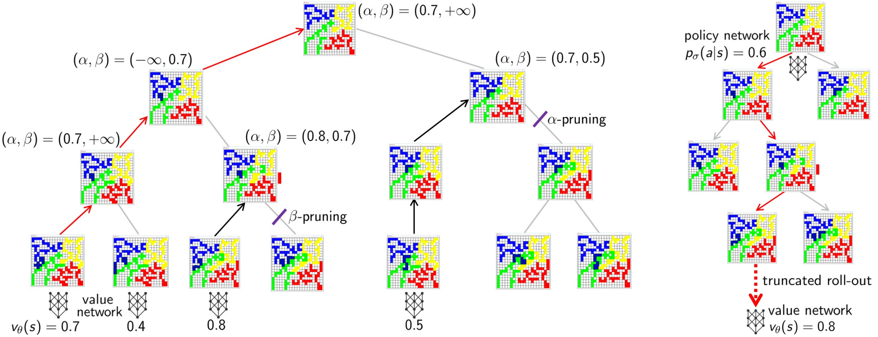
Machine Learning in Practice
#8-1: Simple-Go Project

Sang-Hyun Yoon

Summer 2019

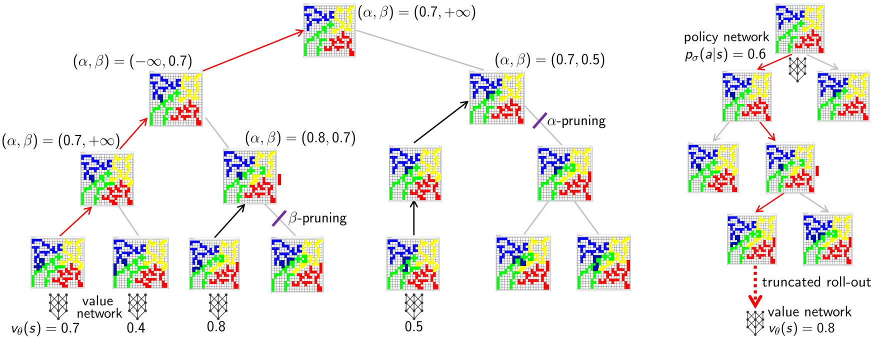
Outline

α-β/MTCS with Value Function?

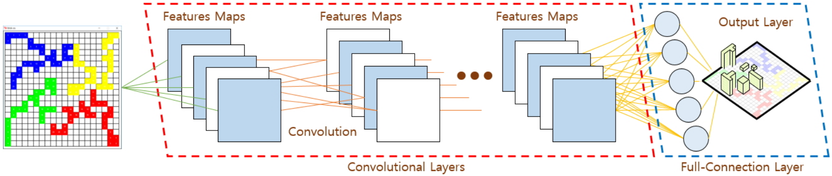


estimated winning probability as value function?

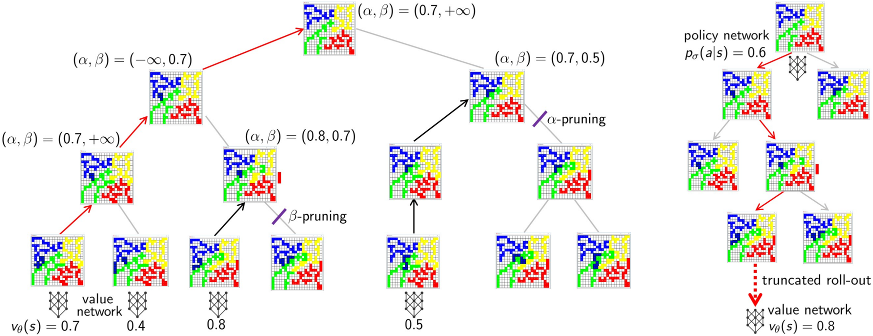
α-β/MTCS with CNN



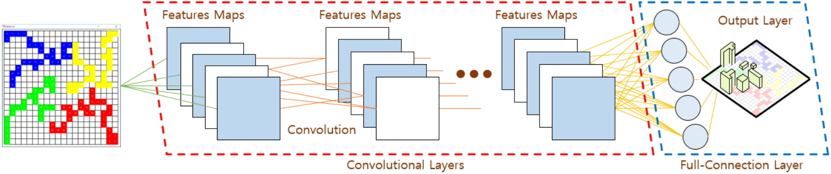
- Value function computed by CNN
- input/output: configuration / estimated winning probability



α-β/MTCS with CNN

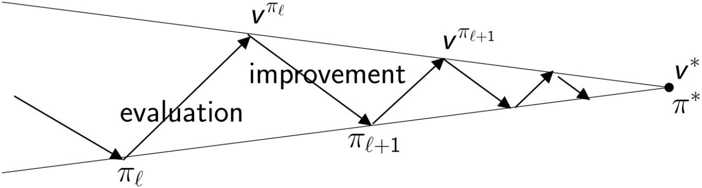


- Value function computed by CNN $\Rightarrow$ **how to train?**
- input/output: configuration / estimated winning probability



Recall: Basic Methods for Reinforcement Learning

- Policy iteration
  - MDP known: dynamic programming
  - MDP unknown: Monte-Carlo



- Temporal-difference learning
  - on-policy: SARSA
  - off-policy: Q-learning

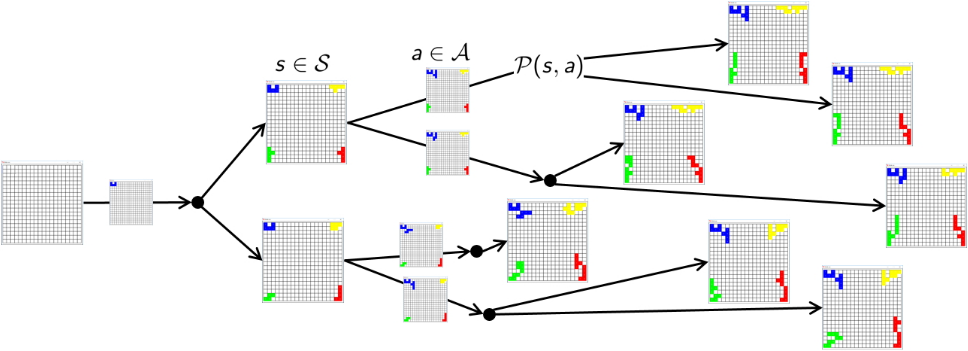
Monte-Carlo policy iteration best suitable for training those CNNs

Combinatorial Game as MDP

Definition (Markov Decision Process)

A Markov decision process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$ where

- $\mathcal{S}$: finite set of states / $\mathcal{A}$: finite set of actions
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to (\mathcal{S} \to [0, 1])$: probability dist. of next states
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$: reward function / $\gamma \in [0, 1]$: discount factor



Recall: Optimal Policy of an MDP

Definition (Markov Decision Process)

A Markov decision process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$ where

- $\mathcal{S}$: finite set of states / $\mathcal{A}$: finite set of actions
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to (\mathcal{S} \to [0,1])$: probability dist. of next states
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$: reward function / $\gamma \in [0,1]$: discount factor

Definition (Policy)

A policy is a (state-to-action) function $\pi : \mathcal{S} \to \mathcal{A}$

Definition (Value Function)

Given $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$ and π, the corresponding value function is

- $v^\pi : \mathcal{S} \to \mathbb{R}$; $v^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v^\pi(S_{t+1}) \mid S_t = s]$

Theorem (Existence of an Optimal Policy for any MDP)

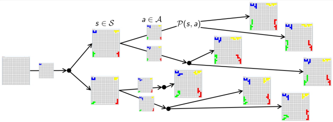
π_{opt} *s.t.* $\boxed{\forall s \in \mathcal{S},\ v^{\pi_{opt}}(s) = \max\{v^\pi(s) \mid \textit{policy}\ \pi\}}$ *always exists*

Winning Strategy of Combinatorial Game = Optimal Policy of MDP

Definition (Markov Decision Process)

A Markov decision process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$ where

- $\mathcal{S}$: finite set of states / $\mathcal{A}$: finite set of actions
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to (\mathcal{S} \to [0,1])$: probability dist. of next states
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$: reward function / $\gamma \in [0,1]$: discount factor



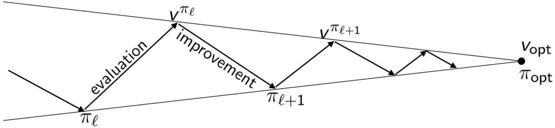
Combinatorial game $(\mathcal{C}, F_1, F_2, m_1, m_2) \longmapsto$ MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$

- $\mathcal{S} = \mathcal{A} := \mathcal{C}$
- $\mathcal{P}(c, c') :=$ pdf of next state after opponent's (optimal) move
- $R(c, c') :=$ average winning probability at next state $\mathcal{P}(c, c')$

Then, winning strategy of the game = optimal policy of MDP

- An optimal policy of MDP can be learnt by policy iteration

Recall: Policy Iteration



Recall: Value function for a given policy

Given $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma)$ and π, the corresponding value function is

- $v^\pi : \mathcal{S} \to \mathbb{R}$; $v^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v^\pi(S_{t+1}) \,|\, S_t = s]$

Equivalently, it can be recursively defined by

- $v^\pi(s) = R^\pi(s) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^\pi(s, s') \cdot v^\pi(s')$

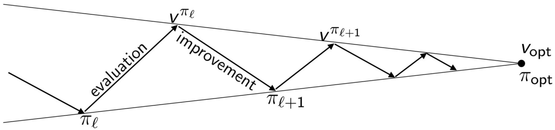
Policy **evaluation**: Given π, find the corresponding v^π

- $v_{k+1}(c) := R^\pi(c) + \gamma \sum_{c' \in \mathcal{C}} \mathcal{P}^\pi(c, c') \cdot v_k(c')$ for $k = 1, 2, \cdots$

Policy **improvement**: Given v, find the corresponding π

- $\pi(c) := \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left(R(c, a) + \gamma \sum_{c' \in \mathcal{C}} \left(\mathcal{P}(c, a, c') \cdot v(c') \right) \right)$

Recall: Policy Iteration



π_0 := `random policy` $\in \mathcal{C} \to \mathcal{A}$

`for` $(\ell := 0$ `to` $\ell_{\mathtt{th}} - 1)$

  v_0 := `random value function` $\in \mathcal{C} \to \mathbb{R}$

  `# policy evaluation`

  `foreach` $(k := 0$ `to` $k_{\mathtt{th}} - 1)$

    `foreach` $(c \in \mathcal{C})$

      $v_{k+1}(c) := R^{\pi_\ell}(c) + \gamma \sum_{c' \in \mathcal{C}} \mathcal{P}^{\pi_\ell}(c, c') \cdot v_k(c')$

  $v^{\pi_\ell} := v_{k_{\mathtt{th}}}$

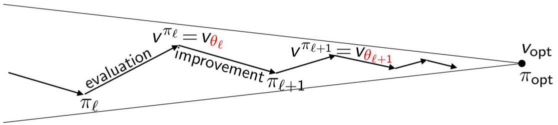
  `# policy improvement`

  `foreach` $(c \in \mathcal{C})$

    $$\pi_{\ell+1}(c) := \operatorname*{argmax}_{a \in \mathcal{A}} \left(R(c, a) + \gamma \sum_{c' \in \mathcal{C}} \left(\mathcal{P}(c, a, c') \cdot v^{\pi_\ell}(c') \right) \right)$$

`return` $\pi_{\ell_{\mathtt{th}}}$

Monte-Carlo Policy Iteration with Value Network



π_0 := random policy $\in \mathcal{C} \to \mathcal{A}$

for ($\ell := 0$ to $\ell_{\text{th}} - 1$)

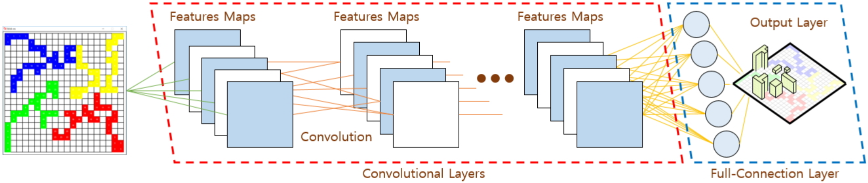
  v_0 := random value function $\in \mathcal{C} \to \mathbb{R}$

  # policy evaluation

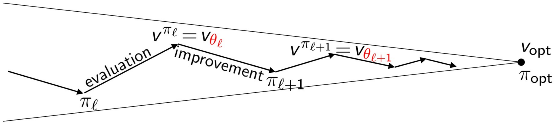
  generate data $T = \{(c, z)\}$ from self-play with policy π_ℓ

  train value network θ_ℓ s.t. $\sum_{(c,z)\in T} \left(v_{\theta_\ell}(c) - z\right)^2$ is min.

    by gradient descent $\theta_\ell \leftarrow \theta_\ell - \eta \sum_{(c,z)\in T} \left(\left(v_{\theta_\ell}(c) - z\right) \frac{\partial v_{\theta_\ell}(c)}{\partial \theta_\ell} \right)$

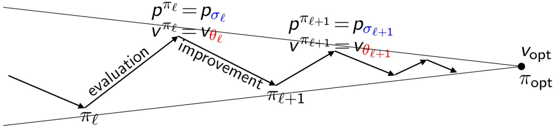


Monte-Carlo Policy Iteration with Value Network



π_0 := random policy $\in \mathcal{C} \to \mathcal{A}$
for $(\ell := 0$ to $\ell_{\text{th}} - 1)$
  v_0 := random value function $\in \mathcal{C} \to \mathbb{R}$
  # policy evaluation
  generate data $T = \{(c, z)\}$ from self-play with policy π_ℓ
  train value network θ_ℓ s.t. $\sum_{(c,z)\in T} \left(v_{\theta_\ell}(c) - z\right)^2$ is min.
    by gradient descent $\theta_\ell \leftarrow \theta_\ell - \eta \sum_{(c,z)\in T} \left(\left(v_{\theta_\ell}(c) - z\right) \frac{\partial v_{\theta_\ell}(c)}{\partial \theta_\ell}\right)$
  v^{π_ℓ} := v_{θ_ℓ}
  # policy improvement
  $\pi_{\ell+1}$:= ε-greedy/α-β/MCTS with value function v^{π_ℓ}
return $\pi_{\ell_{\text{th}}}$

Monte-Carlo Policy Iteration with Value Network & Policy Network



π_0 := `random policy` $\in \mathcal{C} \to \mathcal{A}$

`for` ($\ell := 0$ `to` $\ell_{\text{th}} - 1$)

  v_0 := `random value function` $\in \mathcal{C} \to \mathbb{R}$

  `# policy evaluation`

  `generate data` $T = \{(c, z)\}$ `from self-play with policy` π_ℓ

  `train value network by` $\theta_\ell \leftarrow \theta_\ell - \eta \sum_{(c,z)\in T} \left((v_{\theta_\ell}(c) - z) \frac{\partial v_{\theta_\ell}(c)}{\partial \theta_\ell} \right)$

  `train policy network by` $\sigma_\ell \leftarrow \sigma_\ell + \alpha \sum_{(c,a,z)\in T} \left(\frac{\partial \log p_{\sigma_\ell}(a|c)}{\partial \sigma_\ell} z \right)$

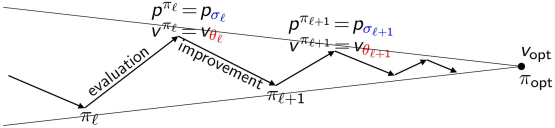
  $v^{\pi_\ell}, p^{\pi_\ell} := v_{\theta_\ell}, p_{\sigma_\ell}$ `(policy gradient)`

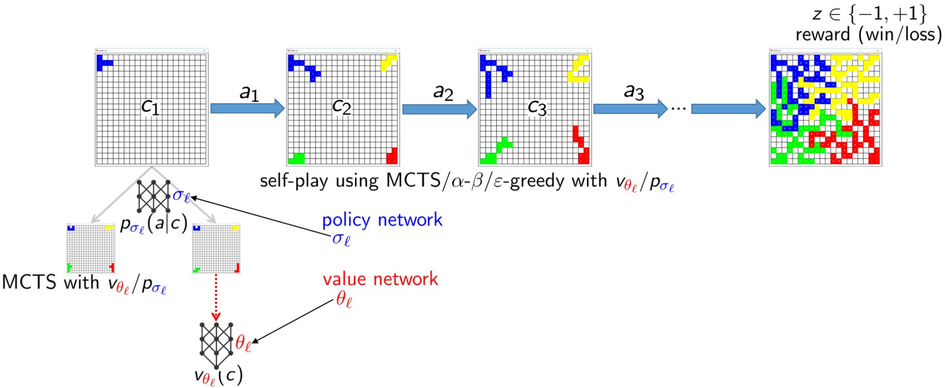
  `# policy improvement`

  $\pi_{\ell+1}$:= ε`-greedy/`α`-`β`/MCTS with` v^{π_ℓ} `&` p^{π_ℓ}

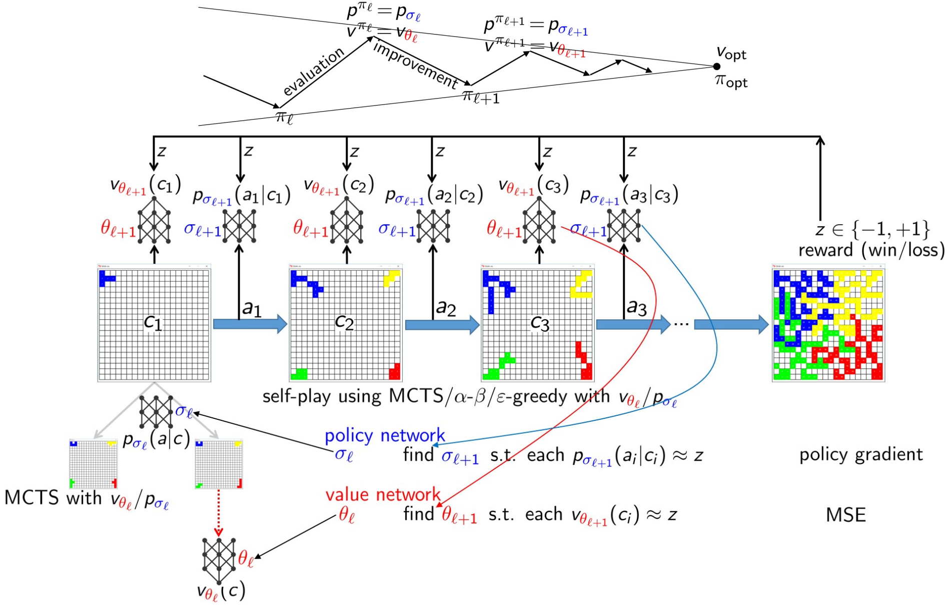
`return` $\pi_{\ell_{\text{th}}}$

Monte-Carlo Policy Iteration with Value Network & Policy Network

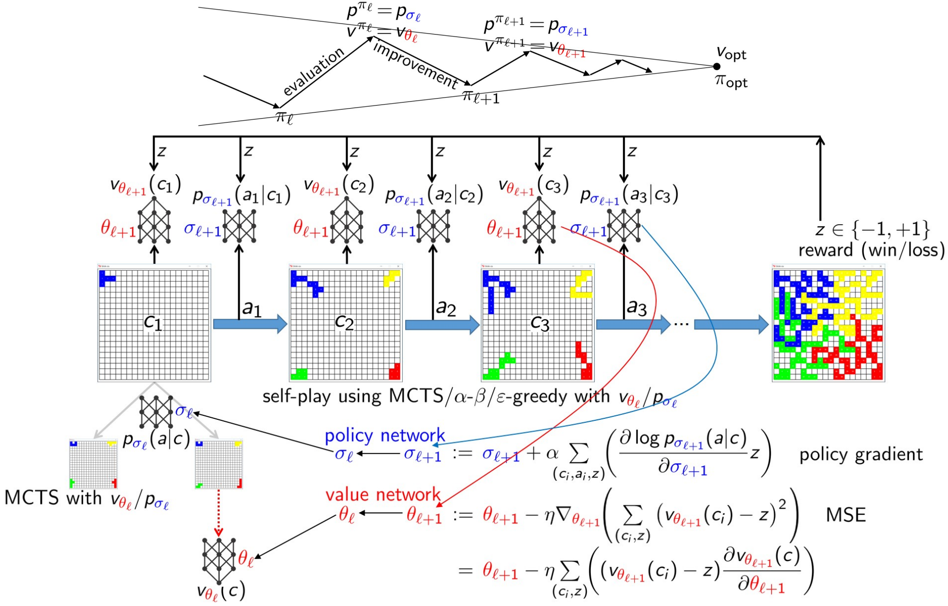




Monte-Carlo Policy Iteration with Value Network & Policy Network



Monte-Carlo Policy Iteration with Value Network & Policy Network

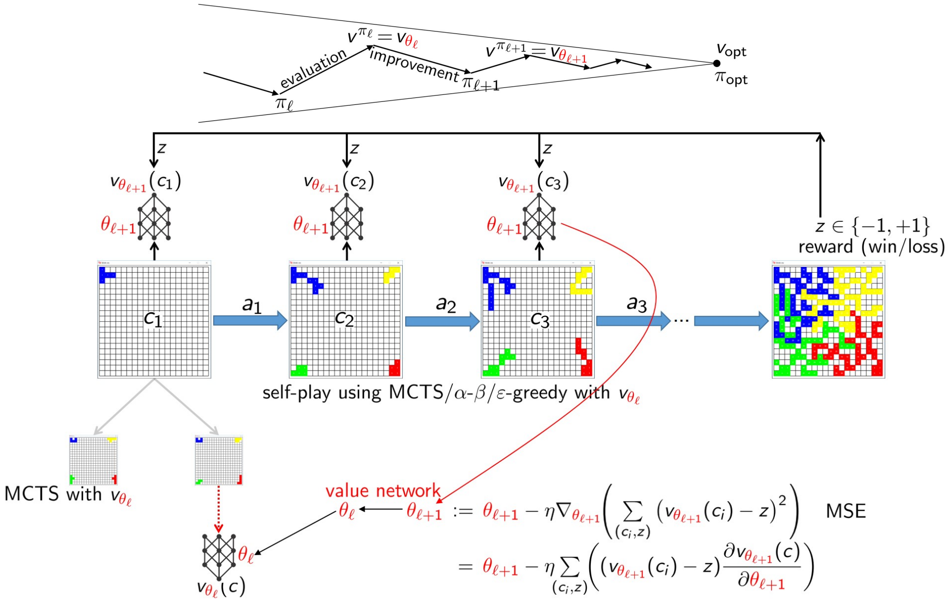


$$\sigma_\ell \longleftarrow \sigma_{\ell+1} := \sigma_{\ell+1} + \alpha \sum_{(c_i,a_i,z)} \left(\frac{\partial \log p_{\sigma_{\ell+1}}(a|c)}{\partial \sigma_{\ell+1}} z \right) \quad \text{policy gradient}$$

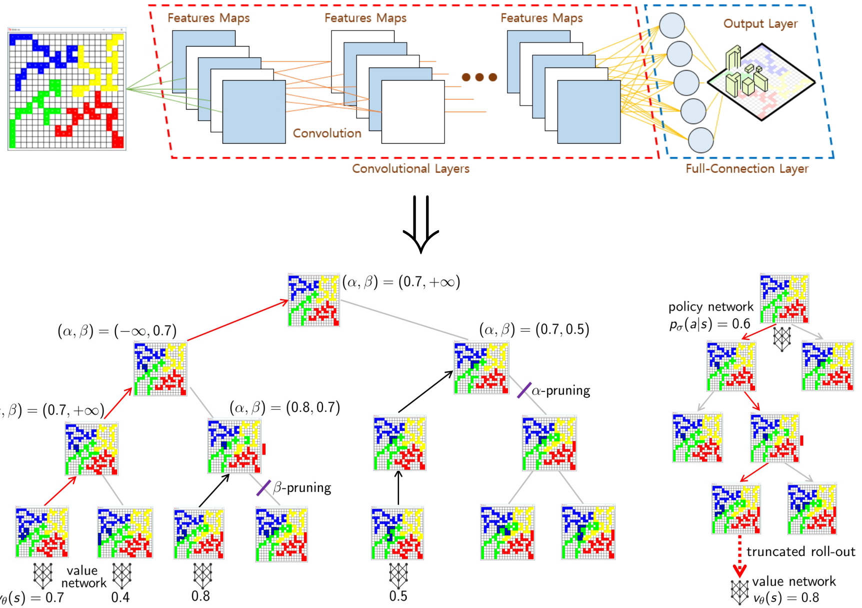
$$\theta_\ell \longleftarrow \theta_{\ell+1} := \theta_{\ell+1} - \eta \nabla_{\theta_{\ell+1}} \left(\sum_{(c_i,z)} \left(v_{\theta_{\ell+1}}(c_i) - z \right)^2 \right) \quad \text{MSE}$$

$$= \theta_{\ell+1} - \eta \sum_{(c_i,z)} \left(\left(v_{\theta_{\ell+1}}(c_i) - z \right) \frac{\partial v_{\theta_{\ell+1}}(c)}{\partial \theta_{\ell+1}} \right)$$

Monte-Carlo Policy Iteration with Value Network



Game-Tree Search with CNNs Trained by Reinforcement Learning



Outline

Recall: Simplifed Rules of Our 5×5-Go

board.py 참고

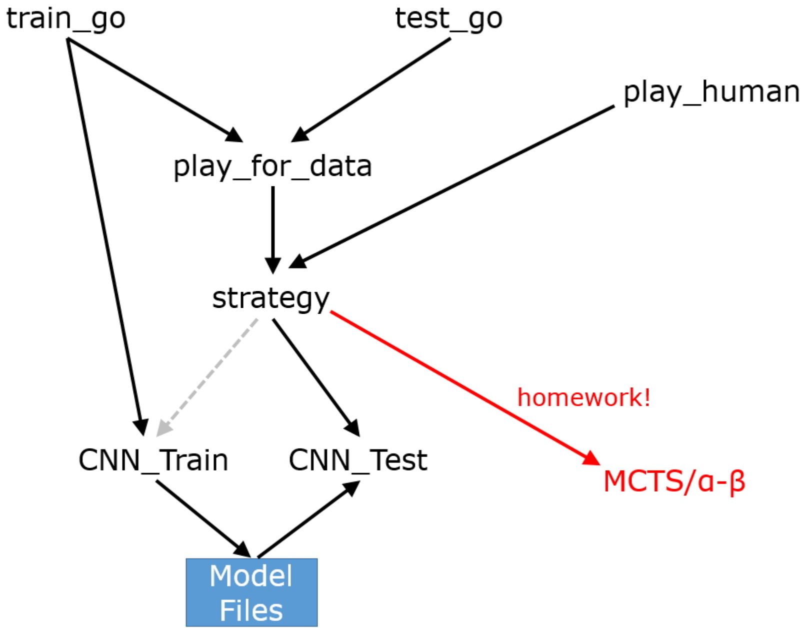
- 5×5 격자 (# states $= 3^{5^2} \approx 8.47 \cdot 10^{11}$)
- 영역 크기 계산시 잡은 돌의 갯수는 무시
- Passing은 valid move가 없을 경우에만 허용됨
- 흑/백 모두 valid move가 없을 때까지 계속 play
  - 사석은 모두 capture되어야 함
  - 공배(neutral space)는 모두 채워져야 함
  - 크기가 1보다 큰 영역은 채워져서 1인 영역들만 남아야 함
- 자신의 크기 1인 영역으로 들어갈 수는 없는데, 예외적으로 그렇게 함으로써 자신의 돌을 보호할 수 있을 때만 허용
  - 즉, 상대가 그 영역으로 들어가면 자신의 돌을 잡을 때
- 백의 덤은 1.5~3.5 (승률이 비슷해지도록 실험으로 결정)

play_human.py로 직접 두면 규칙 파악에 도움이 됨

"Simple-Go" Code for Training/Testing Value Network

- constant.py, graphics/graphics_lib.py, board.py
- play_human.py: 컴퓨터와 사람간 경기
- CNN.py: value/~~policy~~ network를 나타내는 CNN
- strategy.py: value network를 이용하여 최선의 move 선택
  - 이 부분을 MCTS/α-β 등과 결합하여 강화시키는게 숙제
- play_for_data.py: value network를 이용해 random play를 여러번 해서 training/testing data를 모음
  - train_go/test_go.py에 의해 호출됨
  - strategy.py를 호출해서 최선의 move를 선택
- train_go.py: 실험결과를 모아 value network을 training 시킴을 여러 generation에 걸쳐 반복
- test_go.py: 여러 세대의 value network끼리 대결

"Simple-Go" Code: Call Hierarchy



CNN Training Pipeline: 1st Generation Value Network

- 무작위로 두는 게임을 100,000회 수행
- i번째 게임에서 $S_{i,1}, S_{i,2}, \cdots, S_{i,n_i}$ board 상태 순서로 게임이 끝났을 때 승자는 $w_i \in \{흑, 백\} = \{0, 1\}$로 표시하여 모음
  - `train_go.py`에서 `play_for_data.py`를 호출하여 모음
- Training data를 $T = \{(S_{i,j}, w_i) \mid i \in [100000], j \in [n_i]\}$로 하여 CNN(value network)을 training
  - $v(s) =$ "s로 start/resume하면 이길 확률"을 근사화
- $S_{i,j}$는 `S[row][col][stone]` 형태의 3차원 텐서로 표현
  - `board.py`의 `toState()` 참고
  - `(row,col)`위치에 흑이 놓인 경우 `S[row][col][0] = 1`, `S[row][col][1] = S[row][col][2] = 0`
  - 백이 놓인 경우 `S[row][col][1]`를 1로, 비었을 경우 `S[row][col][2]`를 1로 설정하고 나머지는 0으로
- `S[row][col]=stone` 형태의 2차원 텐서로 board 상태를 표현하는 것보다 one-hot vector 표현 방식이 자유도가 높음

CNN Training Pipeline: $g(\geq 2)$-th Generation Value Network

- $g-1$세대 CNN(value network)에 randomness를 추가해서 250,000회 수행하여 g세대 CNN을 training $(g = 2, 3, 4, \cdots)$
  - exploitation: $g-1$세대 value net을 사용하여 move 결정
  - exploration: 두가지 형태의 randomness로 move 결정
- 각 플레이어는 각 게임마다 50%씩의 확률로 두가지 형태의 randomness중 하나를 선택
  1. (각 게임마다 틀리게 선택된) $m \overset{U}{\leftarrow} [1, 12]$에 대해 m 수까지는 random하게, 이후는 $g-1$세대 value net을 사용
     - `train_go.py`에서 `rand[i]`가 음수로 선택된 경우
  2. (각 게임마다 틀리게 선택된) $r \overset{U}{\leftarrow} [0, 1)$에 대해 각 move마다 r의 확률로 random move를 선택
- Randomness 형태와 random value는 `train_go.py`에서 설정되고 `play_for_data.py`에서 exploitation or exploration
- $g = 2, 3$까지는 g세대 value net이 $g-1$세대보다 나은데 그 이후는.. (randomness 형태/정도를 조정해야 함)

CNN.py (1/2)

- Value network(가치망)을 표현하는 CNN 관련 코드
  - Value network은 $v_\theta(s)$ = "s로 start/resume하면 이길 확률"을 근사화하는 (C)NN (with weights θ)
- 현재는 policy network(정책망)을 구성/사용하지는 않음
  - Policy network은 $p_\theta(s,a)$ = "s에서 move a를 선택하는 것이 최적일 확률"을 근사화하는 (C)NN
  - Policy network은 인간 기보가 없다면 효율적이지 않음
- CNN의 입력은 바둑판 상태를 나타내는 `S[i][j][stone]` 형태의 3차원 $N \times N \times 3$ 텐서($N = 5$)
  - `board.py`의 `toState()` 참고
- 주어진 입력 상태에 대한 CNN의 출력은 흑의 승리 확률, 백의 승리 확률, 비길 확률을 나타내는 3개의 softmax unit
  - Policy net을 구성할 경우 25개의 softmax unit를 사용하면 됨
  - 비기는 훈련자료는 없으므로 비길 확률은 거의 0으로 출력

CNN.py (2/2)

- CNN은 convolution layer 3층과 full layer 2층으로 구성됨
  - CNN.py의 NN_state_to_win_prob() 참고
  - 1층은 3×3 filter 30개 사용, 2층은 3×3 filter 50개 사용, 3층은 3×3 filter 70개 사용
  - Pooling layer는 사용하지 않고 padding은 SAME으로 하여 바둑판 크기가 유지됨
  - Convolution 층 간의 연결은 5주차와 마찬가지로 모두 연결
- 흑/백 각각에 대해 value net을 분리하여 구성/이용
  - model 디렉토리에 저장된 파일들 참고
- CNN_Train은 훈련데이터로 CNN을 구성하고 파일에 저장. 바로 이용할 수도 있게 함. train_go.py에서 사용
  - $g-1$세대 CNN을 구성하고 이를 바로 이용하여 g세대 구성
- CNN_Test은 파일에 저장된 CNN을 복원하여 바로 이용 수 있게 함. test_go.py에서 사용

strategy.py

- 입력: 현 보드 상태, 현 상태에서 valid move들의 list, 현재 플레이어(흑/백), value network
- 출력: 현재 플레이어의 "승리 확률"을 최대로 해주는 move
  - 즉, value network를 이용한 minimax search with depth $= 1$
- "승리 확률"은 입력으로 받은 value network으로 추정

- g세대 train_go.py → play_for_data.py로 호출되는 경우
  - random move를 택하지 않고 exploitation을 택할 때, $g-1$ 세대 value net(CNN_Train object)이 입력됨
- test_go.py → play_for_data.py로 호출되는 경우 또는 play_human.py에서 호출되는 경우
  - 항상 exploitation을 택하므로 어떤 세대의 value net (CNN_Test object)이 입력됨

이 부분을 MCTS/α-β 등과 결합하여 강화시키는게 숙제

play_for_data.py

- Training/testing을 위한 데이터를 모으기 위해 strategy.py를 이용하거나 random하게 move를 선택하여 여러 game을 수행
- i번째 게임에서 $S_{i,1}, S_{i,2}, \cdots, S_{i,n_i}$ board 상태 순서로 게임이 끝났을 때 승자는 $w_i \in \{흑, 백\} = \{0, 1\}$로 표시하여 모음
- $\{(S_{i,j}, w_i)\}$와 흑/백의 승률을 리턴

- train_go.py에서 호출되는 경우
  - 입력 rand 배열의 정보에 따라, random move를 택하거나 $g-1$세대 value net으로 move를 결정하면서 게임을 진행
- test_go.py에서 호출되는 경우
  - 어떤 세대의 value net으로 move를 결정하면서 게임을 진행
  - Deterministic하게 게임이 진행되므로 승률이 항상 0 또는 1로만 나오므로 첫 2수를 랜덤하게 두고 게임 진행

train_go.py

"CNN Training Pipeline" 부분 슬라이드 참조

- 1세대의 경우 random move만으로 100,000회 게임 수행
- 2세대 이후의 경우 random 또는 이전 세대의 value net을 이용하여 250,000회 게임을 수행하여 $\{(S_{i,j}, w_i)\}$ 정보 수집
  - play_for_data.py → strategy.py → CNN.py를 거쳐
- 이 정보를 이용하여 CNN.py를 이용하여 value net을 훈련
  - $v(s) =$ "s로 start/resume하면 이길 확률"을 근사화
- 게임 결과는 좌우/상하 대칭 및 회전에 invariant하므로, $\{(S_{i,j}, w_i)\}$에 이 연산들을 적용하여 데이터를 8배로 키움
  - genSymmetry() 참조
- 어떤 보드 상태는 이기기도, 지기도 하는데 이 경우 "중화" 되어 value net의 parameter에 영향을 크게 미치지 않게 됨

test_go.py

- 각 세대의 value net을 이용하여 "승리 확률"을 추정하여 (random move 없이) 1,000회씩 게임을 수행
  - play_for_data.py → strategy.py → CNN.py를 거쳐
  - 0세대는 random move만을 이용
- 백의 덤을 1.5/2.5/3.5로 변경시키며 비교하면 1.5/2.5 일때는 흑이 지나치게 강함. 따라서 3.5로 선택
  - 3.5일 경우에도 여전히 흑이 강함. 초기 상태가 (흑의 관점에서) winning position인 것으로 강력히 추정됨
  - 각 크기의 덤에 대해 15세대씩 train/test
  - 한 세대 데이터 생성/training에 5∼7시간 정도 (TitanX GPU 장착 컴퓨터 3대로 4일 정도)
- 덤 3.5에서 흑은 4세대가 가장 강하고, 백은 5세대가 흑 4 세대에 가장 강해보이므로 이 둘을 숙제용으로 사용

test_go.py: Experimental Results with 백덤= 3.5

	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	Ave.
B1	1.00	0.54	0.39	0.35	0.47	0.41	0.46	0.43	0.45	0.37	0.38	0.476
B2	1.00	0.69	0.58	0.51	0.58	0.58	0.60	0.51	0.54	0.54	0.53	0.606
B3	1.00	0.64	0.59	0.55	0.58	0.54	0.58	0.55	0.55	0.59	0.55	0.611
B4	1.00	0.68	0.63	0.59	0.63	0.57	0.63	0.58	0.60	0.63	0.65	0.654
B5	1.00	0.65	0.51	0.55	0.53	0.56	0.49	0.52	0.57	0.60	0.52	0.592
B6	1.00	0.71	0.62	0.59	0.62	0.61	0.64	0.53	0.52	0.58	0.53	0.632
B7	1.00	0.74	0.59	0.55	0.64	0.54	0.60	0.62	0.58	0.65	0.58	0.644
B8	1.00	0.70	0.58	0.45	0.56	0.53	0.58	0.57	0.55	0.51	0.59	0.601
B9	1.00	0.67	0.53	0.53	0.57	0.56	0.56	0.56	0.53	0.52	0.56	0.598
B10	1.00	0.69	0.62	0.59	0.57	0.52	0.61	0.52	0.50	0.52	0.57	0.610

	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	Ave.
W1	1.00	0.46	0.32	0.36	0.32	0.35	0.29	0.26	0.30	0.33	0.32	0.392
W2	1.00	0.61	0.41	0.41	0.37	0.49	0.38	0.41	0.42	0.47	0.38	0.486
W3	1.00	0.65	0.49	0.45	0.41	0.45	0.41	0.45	0.55	0.47	0.41	0.522
W4	1.00	0.54	0.42	0.42	0.37	0.47	0.38	0.36	0.44	0.43	0.43	0.478
W5	1.00	0.59	0.41	0.46	0.43	0.43	0.39	0.47	0.47	0.44	0.48	0.507
W6	1.00	0.55	0.40	0.41	0.37	0.51	0.36	0.40	0.42	0.44	0.39	0.477
W7	1.00	0.57	0.49	0.45	0.42	0.48	0.47	0.38	0.42	0.44	0.48	0.509
W8	1.00	0.55	0.46	0.45	0.40	0.43	0.48	0.42	0.46	0.47	0.50	0.510
W9	1.00	0.63	0.46	0.41	0.37	0.40	0.42	0.35	0.49	0.48	0.48	0.499
W10	1.00	0.62	0.47	0.45	0.35	0.48	0.47	0.42	0.41	0.44	0.43	0.505

test_go.py: Experimental Results with 백덤= 2.5

	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	Ave.
B1	1.00	0.52	0.31	1.00	0.52	0.32	0.36	0.45	0.36	0.41	0.42	0.514
B2	1.00	0.75	0.68	1.00	0.65	0.61	0.63	0.69	0.62	0.61	0.66	0.719
B3	1.00	0.74	0.69	1.00	0.68	0.55	0.61	0.68	0.63	0.61	0.62	0.710
B4	1.00	0.56	0.44	1.00	0.54	0.35	0.45	0.45	0.52	0.46	0.38	0.560
B5	1.00	0.82	0.74	1.00	0.85	0.74	0.70	0.72	0.71	0.70	0.70	0.790
B6	1.00	0.72	0.64	1.00	0.76	0.67	0.63	0.63	0.58	0.65	0.69	0.725
B7	1.00	0.64	0.55	1.00	0.66	0.54	0.56	0.52	0.47	0.49	0.52	0.630
B8	1.00	0.68	0.62	1.00	0.68	0.53	0.59	0.65	0.59	0.53	0.60	0.681
B9	1.00	0.70	0.61	1.00	0.70	0.59	0.65	0.68	0.66	0.61	0.62	0.711
B10	1.00	0.77	0.71	1.00	0.68	0.56	0.61	0.69	0.58	0.69	0.60	0.719

	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	Ave.
W1	1.00	0.48	0.25	0.26	0.44	0.18	0.28	0.36	0.32	0.30	0.23	0.372
W2	1.00	0.69	0.32	0.31	0.56	0.26	0.36	0.45	0.38	0.39	0.29	0.457
W3	0.53	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.048
W4	1.00	0.48	0.35	0.33	0.46	0.15	0.24	0.34	0.32	0.29	0.32	0.389
W5	1.00	0.68	0.39	0.45	0.65	0.26	0.33	0.47	0.47	0.41	0.44	0.505
W6	1.00	0.64	0.37	0.39	0.55	0.30	0.37	0.44	0.41	0.35	0.39	0.473
W7	1.00	0.55	0.31	0.32	0.55	0.28	0.37	0.48	0.35	0.32	0.30	0.438
W8	1.00	0.64	0.38	0.37	0.48	0.29	0.42	0.53	0.41	0.34	0.42	0.479
W9	1.00	0.59	0.39	0.39	0.54	0.30	0.35	0.51	0.47	0.39	0.30	0.476
W10	1.00	0.58	0.34	0.38	0.62	0.29	0.31	0.48	0.40	0.38	0.40	0.471

test_go.py: Experimental Results with 백덤= 1.5

	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	Ave.
B1	1.00	0.47	0.24	1.00	0.55	0.37	1.00	0.48	0.28	1.00	0.57	0.631
B2	1.00	0.82	0.68	1.00	0.78	0.64	1.00	0.79	0.60	1.00	0.79	0.827
B3	1.00	0.77	0.61	1.00	0.78	0.69	1.00	0.79	0.56	1.00	0.77	0.815
B4	1.00	0.71	0.53	1.00	0.70	0.43	1.00	0.67	0.41	1.00	0.68	0.740
B5	1.00	0.85	0.68	1.00	0.84	0.68	1.00	0.81	0.61	1.00	0.79	0.843
B6	1.00	0.78	0.69	1.00	0.75	0.60	1.00	0.74	0.64	1.00	0.73	0.812
B7	1.00	0.69	0.50	1.00	0.65	0.49	1.00	0.64	0.40	1.00	0.70	0.733
B8	1.00	0.77	0.71	1.00	0.78	0.68	1.00	0.80	0.65	1.00	0.80	0.835
B9	1.00	0.75	0.63	1.00	0.73	0.58	1.00	0.71	0.61	1.00	0.70	0.792
B10	1.00	0.77	0.57	1.00	0.79	0.53	1.00	0.67	0.47	1.00	0.70	0.773

	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	Ave.
W1	1.00	0.53	0.18	0.23	0.29	0.15	0.22	0.31	0.23	0.25	0.23	0.329
W2	1.00	0.76	0.32	0.39	0.47	0.32	0.32	0.50	0.29	0.37	0.43	0.470
W3	0.54	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.050
W4	1.00	0.45	0.22	0.22	0.30	0.16	0.25	0.35	0.23	0.27	0.21	0.331
W5	1.00	0.63	0.36	0.31	0.56	0.32	0.40	0.51	0.32	0.42	0.47	0.481
W6	0.53	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.048
W7	1.00	0.52	0.21	0.21	0.33	0.19	0.26	0.36	0.20	0.29	0.34	0.354
W8	1.00	0.72	0.40	0.44	0.59	0.39	0.36	0.60	0.35	0.39	0.53	0.525
W9	0.51	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.047
W10	1.00	0.43	0.21	0.23	0.32	0.21	0.27	0.30	0.20	0.30	0.30	0.342

play_human.py

- 컴퓨터와 사람간 경기 수행
  - 컴퓨터 플레이어는 과제에서 사용할 흑 ?세대, 백 ?세대 value network를 사용하여 greedy하게
- WingIDE에서는 graphic 문제로 작동안됨. 터미널에서 수행
- 예: `python play_human.py`
- 코드 윗 줄에서 사람/컴퓨터 플레이어의 흑/백 변경
- `selectMove()`의 마지막 parameter인 `show_prob`를 `True`로 설정하여 `CNN.py`에서 각 move에 대한 "승리 확률"을 출력하도록 함

On the "Winning Probability"

- "Combinatorial Game"에서 다루었듯이 각 상태는 각 플레이어에 대해 winning position 또는 losing position 둘 중 하나이므로 승리 확률은 1 또는 0
  - 바둑의 초기 상태도 winning/losing position 둘 중 하나이고 초기 상태는 (흑에게) winning position으로 강력히 추정
  - 사람끼리 두면 흑백의 승률이 0.52/0.48라는데 AlphaGo는 0.8/0.2 정도라고 함 (즉, AlphaGo도 필승전략을 모름)
- 즉, 승리 "확률"이라는 자체가 정의가 잘 안됨
- MCTS에서 다루었듯이, "승리 확률"값을 적절히 정하고 이를 (MCTS/CNN으로) "추정"한 것을 최대화 하면 winning position으로 갈 가능성을 높일 것으로 믿을 뿐
  - PSPACE-hard인 바둑 문제의 경우 winning position을 빠른 시간에 계산하기 매우 힘드므로 "승리 확률"을 도입하여 search할 수 밖에 없는 상황

과제 수행용 코드 (1/2)

- 사용할 value network로 흑 4세대, 백 5세대만 제공
- strategy.py: 이 부분을 value network와 MCTS/α-β 등을 결합하여 강화시키는게 숙제
  - default로 주어진 strategy는 value net를 이용한 minimax search with depth= 1 (training 때도 이 방법을 사용)
  - 학생들간의 경기 자동화를 위해 MCTS/α-β 관련 코드들도 모두 이 파일에 모아둘 것 (KOI에서 처럼)
- test_go.py: default strategy vs. your strategy
- play_human.py: your strategy vs. human
  - 자신의 strategy를 간단히 실험할 때만 사용하면 됨. 정확한 성능 측정은 test_go.py로 하는 것이 도움됨

과제 수행용 코드 (2/2)

- 더욱 강화시키려면 알파고 training pipeline에서와 마찬가지로 CNN + MCTS/α-β으로 훈련데이터 생성해서 CNN 훈련하고 이를 세대마다 반복
  - 현재는 CNN + α-β with depth= 1로 생성한 데이터로 CNN를 훈련을 완료한 후에나 CNN + MCTS/α-β 결합하는 수준
- 단, 위 방식은 시간이 매우 많이 걸림. 위 방식으로 하면 데이터 생성 시간이 몇백배 이상 많이 걸림
  - 현재 250,000회 게임의 데이터 생성시간이 3∼5시간으로 한 게임당 0.05∼0.06 수준
  - MCTS에서 100회 trial할 경우 한 게임당 40초 정도 걸림 (700배 정도). 1000회 trial하면 5∼6분 (6000배 정도)
- 제출 전 다른 학생들과 sparring함을 금지하지는 않으나 과열경쟁을 방지하기 위해 지양함을 부탁함