Final Project of SI252 --AlphaGo

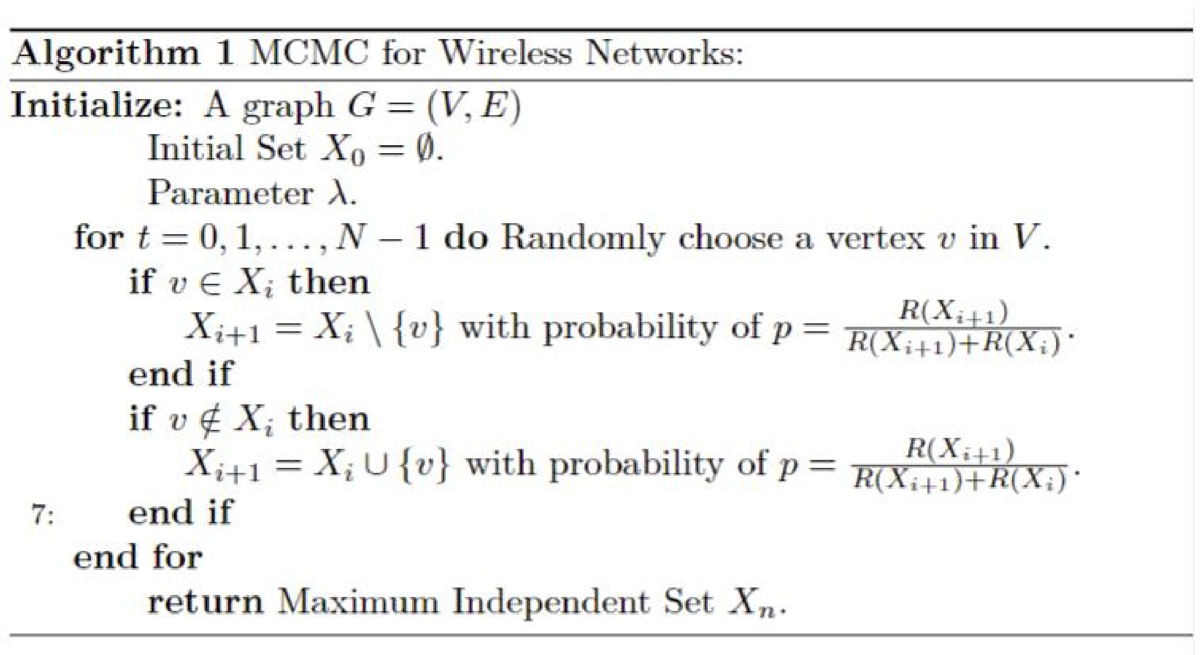
Author (One Person Group): 徐博文

Student ID: 2020533049

Email: xubw1@shanghaitech.edu.cn

1. Maximal Cuts

(a) MCMC Method



```
Algorithm 1 MCMC for Wireless Networks:
Initialize: A graph $G = (V, E)$
        Initial Set $X_0 = \emptyset$.
        Parameter $\lambda$.
    for $t = 0, 1, \dots, N-1$ do Randomly choose a vertex $v$ in $V$.
        if $v \in X_i$ then
            $X_{i+1} = X_i \setminus \{v\}$ with probability of $p = \frac{R(X_{i+1})}{R(X_{i+1})+R(X_i)}$.
        end if
        if $v \notin X_i$ then
            $X_{i+1} = X_i \cup \{v\}$ with probability of $p = \frac{R(X_{i+1})}{R(X_{i+1})+R(X_i)}$.
7:      end if
    end for
        return Maximum Independent Set $X_n$.
```

Discrete Markov Chain

Theory Proof:

Firstly, the chain is ergodic with stationary distribution π. Because the chain has self-loop, thus it is aperiodic. Also, each state can return to initial state, and the initial state can transit to any state, thus the chain is irreducible.

Suppose a state of valid independent set X, after the calculation, we can find that the stationary distribution of this chain satisfies: $\pi(X) \propto e^{R(X)}$, where $R(X)$ means the reward of state X. Therefore, the most frequent state of the chain is the maximum independent set.

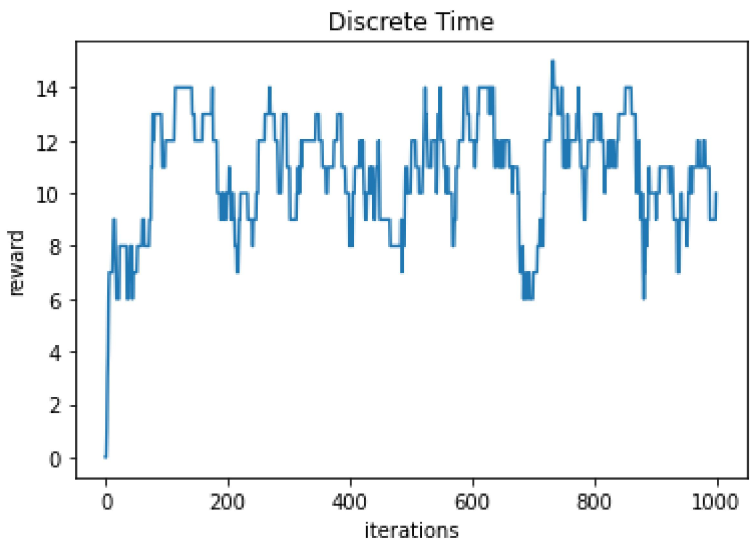
Therefore, we can simulate the chain to find the maximum independent set.

In [ ]:
```
import numpy as np
import pandas as pd
import collections
from matplotlib import pyplot as plt

# check wheter the state is a valid independent set
def return_reward(state, Edge):
    reward = 0

    for edge in Edge:
        if (state[edge[0]] == 0 and state[edge[1]] == 1) \
            or (state[edge[1]] == 1 and state[edge[1]] == 0):
            reward += 1

    return reward

def DTMC_transition(state, edge):

    v = np.random.randint(1,len(state))
    next_state = state.copy()

    if state[v] == 1:
        next_state[v] = 0
    else:
        next_state[v] = 1

    reward = return_reward(state,edge)
    next_reward = return_reward(next_state,edge)
    p = np.exp(next_reward)/(np.exp(reward)+np.exp(next_reward))

    if np.random.binomial(1,p,1) == 1:
        return next_state
    else:
        return state

def DTMC_networks(num_exper, num_ver, edge):

    # initial state
    state = np.zeros(num_ver+1)
    max_state = np.zeros(num_ver+1)
    Reward = np.array([])
    max_reward = 0

    # neighbour
    neigh = []
    for i in range(num_ver+1):
        neigh.append([])

    for i in range(len(edge)):

        # store the neighbour of vertex
        if edge[i][1] not in neigh[edge[i][0]]:
            neigh[edge[i][0]].append(edge[i][1])
            neigh[edge[i][1]].append(edge[i][0])

    for i in range(num_exper):

        # transit
        state = DTMC_transition(state,edge)
        reward = return_reward(state,edge)

        Reward = np.append(Reward,reward)

        if reward > max_reward:

            max_reward = reward
            max_state = state.copy()

    return max_state, Reward, max_reward

# parameter lambda
lam = 20

# number of vertex
num_ver = 20
```

```
# number of experiment
num_exper = 1000

# edge
edge = [[1,2],[2,3],[3,4],[4,5],[5,1],
        [1,16],[2,15],[3,13],[4,8],[5,6],
        [6,7],[7,8],[8,11],[11,13],[13,14],[14,15],[15,17],[17,16],[16,18],[18,6],
        [7,9],[10,11],[12,14],[17,20],[18,19],
        [9,10],[10,12],[12,20],[20,19],[19,9]]

max_state,Reward,max_reward = DTMC_networks(num_exper,num_ver,edge)

max_set = np.array([])
for i in range(num_ver+1):
    if max_state[i] == 1:
        max_set = np.append(max_set,i)

print("maximum of reward is = ", max_reward)
print("maximum number of vertex in independent set:")
print(max_set)

plt.plot(Reward)
plt.title("Discrete Time")
plt.xlabel("iterations")
plt.ylabel("reward")
```

```
maximum of reward is =  15
maximum number of vertex in independent set:
[ 3.  5.  6.  8.  9. 11. 14. 15. 16. 19. 20.]
```

```
Out[ ]: Text(0, 0.5, 'reward')
```



Continuous Markov Chain

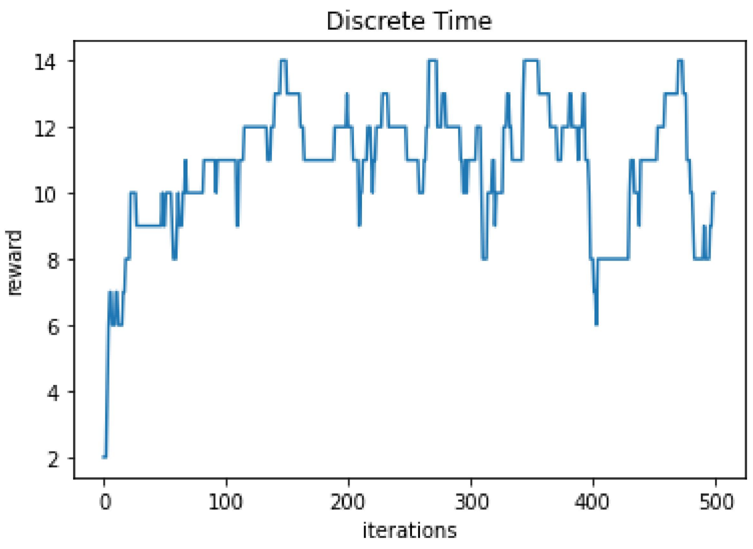
We choose possion process to simulate the CTMC.

Similar to the DTMC above, the CTMC chain is also ergodic and the transition probability of embedded chain $p = \frac{\lambda 1}{\lambda 1+\lambda 2}$, which is the same as the DTMC. Thus, the CTMC chain also, converge to a stationary distribution $\pi(X) \propto e^{R(X)}$, and the most frequent state is the maximum independent set. Therefore, we can simulate the chain to find the maximum independent set.

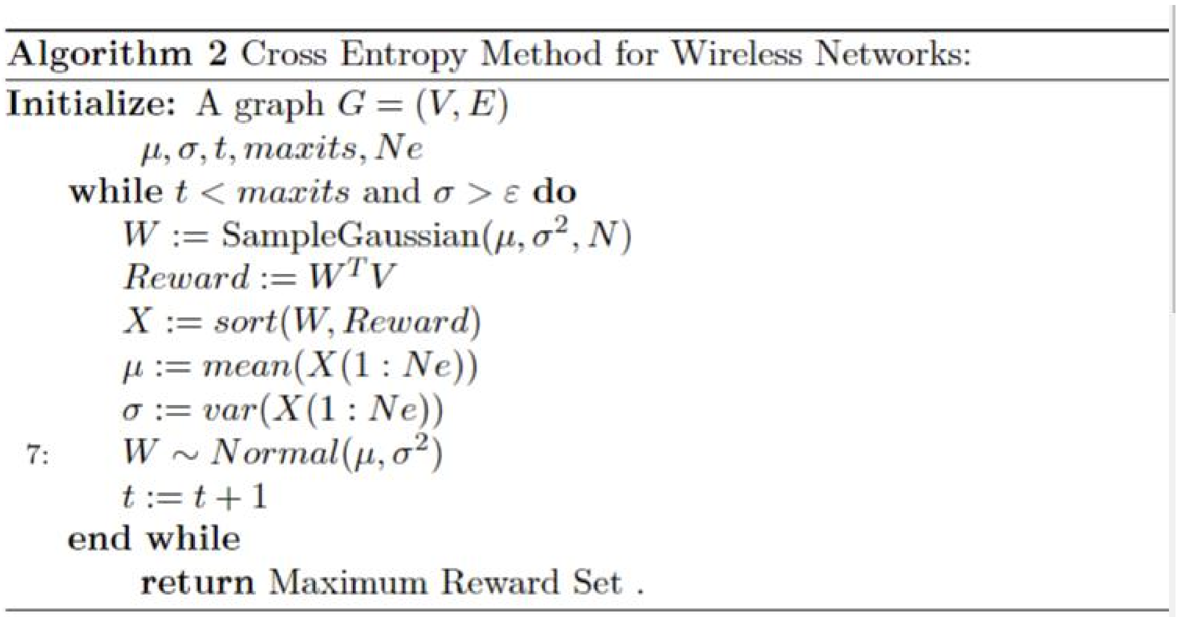
```
In [ ]: import numpy as np
        import pandas as pd
import collections
from matplotlib import pyplot as plt

# check wheter the state is a valid independent set
def return_reward(state, Edge):
    reward = 0

    for edge in Edge:
        if (state[edge[0]] == 0 and state[edge[1]] == 1) \
            or (state[edge[1]] == 1 and state[edge[1]] == 0):
            reward += 1

    return reward

def CTMC_transition(state, edge):

    v = np.random.randint(1,len(state))
    next_state = state.copy()

    if state[v] == 1:
        next_state[v] = 0
    else:
        next_state[v] = 1

    reward = return_reward(state,edge)
    next_reward = return_reward(next_state,edge)
    p = np.exp(next_reward)/(np.exp(reward)+np.exp(next_reward))

    # continuous time: perspective2
    if np.random.exponential(1/p) < np.random.exponential(1/(1-p)):
        return next_state
    else:
        return state

def CTMC_networks(num_exper, num_ver, edge):

    # initial state
    state = np.zeros(num_ver+1)
    max_state = np.zeros(num_ver+1)
    Reward = np.array([])
    max_reward = 0

    # neighbour
    neigh = []
    for i in range(num_ver+1):
        neigh.append([])

    for i in range(len(edge)):

        # store the neighbour of vertex
        if edge[i][1] not in neigh[edge[i][0]]:
            neigh[edge[i][0]].append(edge[i][1])
            neigh[edge[i][1]].append(edge[i][0])

    for i in range(num_exper):

        # transit
        state = CTMC_transition(state,edge)
        reward = return_reward(state,edge)

        Reward = np.append(Reward,reward)

        if reward > max_reward:
                max_reward = reward
                max_state = state.copy()

        return max_state, Reward, max_reward

# parameter lambda
lam = 20

# number of vertex
num_ver = 20

# number of experiment
num_exper = 500

# edge
edge = [[1,2],[2,3],[3,4],[4,5],[5,1],
        [1,16],[2,15],[3,13],[4,8],[5,6],
        [6,7],[7,8],[8,11],[11,13],[13,14],[14,15],[15,17],[17,16],[16,18],[18,6],
        [7,9],[10,11],[12,14],[17,20],[18,19],
        [9,10],[10,12],[12,20],[20,19],[19,9]]

max_state,Reward,max_reward = CTMC_networks(num_exper,num_ver,edge)

max_set = np.array([])
for i in range(num_ver+1):
    if max_state[i] == 1:
        max_set = np.append(max_set,i)

print("maximum of reward is = ", max_reward)
print("maximum number of vertex in independent set:")
print(max_set)

plt.plot(Reward)
plt.title("Discrete Time")
plt.xlabel("iterations")
plt.ylabel("reward")
```

```
maximum of reward is =  14
maximum number of vertex in independent set:
[ 2.  5.  6.  8.  9. 13. 14. 16. 19. 20.]
```

Out[ ]:
```
Text(0, 0.5, 'reward')
```



(b) Cross-Entropy Method



```
Algorithm 2 Cross Entropy Method for Wireless Networks:
Initialize: A graph $G = (V, E)$
        $\mu, \sigma, t, maxits, Ne$
    while $t < maxits$ and $\sigma > \varepsilon$ do
        $W :=$ SampleGaussian$(\mu, \sigma^2, N)$
        $Reward := W^T V$
        $X := sort(W, Reward)$
        $\mu := mean(X(1 : Ne))$
        $\sigma := var(X(1 : Ne))$
7:      $W \sim Normal(\mu, \sigma^2)$
        $t := t + 1$
    end while
        return Maximum Reward Set .
```

In [ ]:

```python
import math
import numpy as np
from matplotlib import pyplot as plt

PERCENTILE = 70
NUM_VER = 20
SAMPLE_TOTAL = 16
# EDGE
EDGE = [[1,2],[2,3],[3,4],[4,5],[5,1],
        [1,16],[2,15],[3,13],[4,8],[5,6],
        [6,7],[7,8],[8,11],[11,13],[13,14],[14,15],[15,17],[17,16],[16,18],[18,6],
        [7,9],[10,11],[12,14],[17,20],[18,19],
        [9,10],[10,12],[12,20],[20,19],[19,9]]

def return_reward(weight):
    index = []
    reward = 0

    for i in range(1, NUM_VER+1):
        # print("weight",weight[i])
        if np.random.binomial(1, weight[i], 1) == 1:
            index.append(i)

    for edge in EDGE:
        if (edge[0] in index and edge[1] not in index) \
            or (edge[0] not in index and edge[1] in index):
            reward += 1

    return reward

def CEM(error):
    weight = np.zeros([SAMPLE_TOTAL, NUM_VER+1])
    mu = np.zeros(NUM_VER+1)
    sigma = np.zeros(NUM_VER+1)
    mu.fill(0.5)
    sigma.fill(1)
    # mu = [0.5 for i in range(NUM_VER+1)]
    # sigma = [1 for i in range(NUM_VER+1)]

    while True:
        old_mu = mu
        old_sigma = sigma
        # sample
        for i in range(SAMPLE_TOTAL):
            for j in range(1, NUM_VER+1):
                weight[i][j] = np.random.normal(mu[j],sigma[j])
                while weight[i][j] > 1 or weight[i][j] < 0:
                    weight[i][j] = np.random.normal(mu[j],sigma[j])

        reward_dict = {}
        # calculate reward
        for i in range(SAMPLE_TOTAL):
            # reward_dict: (i: [reward,weight])
            reward_dict[i] = [return_reward(weight[i]),weight[i]]

        sample_fliter = int(SAMPLE_TOTAL*PERCENTILE/100)
        reward_list = reward_dict.values()
        reward_list = sorted(reward_list, key=lambda x:x[0])

        # fliter_reward: [(reward,weight)]
        fliter_reward = reward_list[:sample_fliter]
        # fliter_weight: {i: weight}
        fliter_weight = {}
        for i in range(sample_fliter):
            fliter_weight[i] = fliter_reward[i][1]

        for dim in range(1,NUM_VER+1):
            mu[dim] = np.mean(list(fliter_weight.values()))
            sigma[dim] = math.sqrt(np.var(list(fliter_weight.values())))

        error_mu = np.abs(np.sqrt(np.sum(np.square(old_mu-mu))))
        error_sigma = np.abs(np.sqrt(np.sum(np.square(old_sigma-sigma))))

        if error_mu < error and error_sigma < error:
            break

    return fliter_reward[0][1]

error = 0.001
weight = CEM(error)
opt_index = []
max_reward = 0
num_eposide = 50

for eposide in range(num_eposide):
    reward = 0
    index = []
    for i in range(1, NUM_VER+1):
        if np.random.binomial(1, weight[i], 1) == 1:
            index.append(i)

    for edge in EDGE:
        if (edge[0] in index and edge[1] not in index) \
            or (edge[0] not in index and edge[1] in index):
            reward += 1

    if reward > max_reward:
        max_reward = reward
        opt_index = index

print("maximum of reward is = ", max_reward)
print("maximum number of vertex in independent set:")
print(opt_index)
```

```
maximum of reward is =  20
maximum number of vertex in independent set:
```

```
[1, 3, 6, 7, 10, 11, 15, 20]
```

(c) MCTS Method

The code is in the python file "**mcts.py**".

The procedure of MCTS:

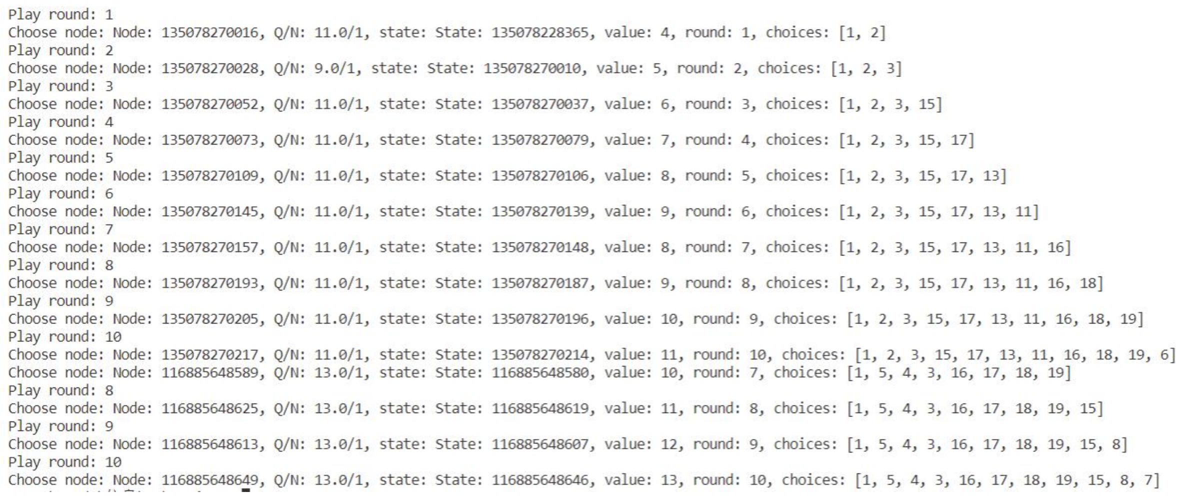
1. Selection: Starting from the root node R, recursively select the optimal child node (explained later) until the leaf node L is reached.
2. Expansion: If L is not a termination node (i.e., does not cause the game to terminate) then create one or more subnodes of the word, choosing one of C.
3. Simulation: The output of running a simulation from C until the end of the game.
4. Backpropagation: Update the current action sequence with the resulting output of the simulation

To choose the best child node each step, we use the Upper confidence bound:

$$C = \underset{v' \in children\ of\ v}{argmax} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2logN(v)}{N(v')}}$$

To ensure both exploration ($c\sqrt{\frac{2logN(v)}{N(v')}}$) and exploitation ($\frac{Q(v')}{N(v')}$).

The code is in the attachment. The result is as below:



```
Play round: 1
Choose node: Node: 135078270016, Q/N: 11.0/1, state: State: 135078228365, value: 4, round: 1, choices: [1, 2]
Play round: 2
Choose node: Node: 135078270028, Q/N: 9.0/1, state: State: 135078270010, value: 5, round: 2, choices: [1, 2, 3]
Play round: 3
Choose node: Node: 135078270052, Q/N: 11.0/1, state: State: 135078270037, value: 6, round: 3, choices: [1, 2, 3, 15]
Play round: 4
Choose node: Node: 135078270073, Q/N: 11.0/1, state: State: 135078270079, value: 7, round: 4, choices: [1, 2, 3, 15, 17]
Play round: 5
Choose node: Node: 135078270109, Q/N: 11.0/1, state: State: 135078270106, value: 8, round: 5, choices: [1, 2, 3, 15, 17, 13]
Play round: 6
Choose node: Node: 135078270145, Q/N: 11.0/1, state: State: 135078270139, value: 9, round: 6, choices: [1, 2, 3, 15, 17, 13, 11]
Play round: 7
Choose node: Node: 135078270157, Q/N: 11.0/1, state: State: 135078270148, value: 8, round: 7, choices: [1, 2, 3, 15, 17, 13, 11, 16]
Play round: 8
Choose node: Node: 135078270193, Q/N: 11.0/1, state: State: 135078270187, value: 9, round: 8, choices: [1, 2, 3, 15, 17, 13, 11, 16, 18]
Play round: 9
Choose node: Node: 135078270205, Q/N: 11.0/1, state: State: 135078270196, value: 10, round: 9, choices: [1, 2, 3, 15, 17, 13, 11, 16, 18, 19]
Play round: 10
Choose node: Node: 135078270217, Q/N: 11.0/1, state: State: 135078270214, value: 11, round: 10, choices: [1, 2, 3, 15, 17, 13, 11, 16, 18, 19, 6]
Choose node: Node: 116885648589, Q/N: 13.0/1, state: State: 116885648580, value: 10, round: 7, choices: [1, 5, 4, 3, 16, 17, 18, 19]
Play round: 8
Choose node: Node: 116885648625, Q/N: 13.0/1, state: State: 116885648619, value: 11, round: 8, choices: [1, 5, 4, 3, 16, 17, 18, 19, 15]
Play round: 9
Choose node: Node: 116885648613, Q/N: 13.0/1, state: State: 116885648607, value: 12, round: 9, choices: [1, 5, 4, 3, 16, 17, 18, 19, 15, 8]
Play round: 10
Choose node: Node: 116885648649, Q/N: 13.0/1, state: State: 116885648646, value: 13, round: 10, choices: [1, 5, 4, 3, 16, 17, 18, 19, 15, 8, 7]
```

We can find that there exists some bias between the MCTS result and the optimal solution.

Pros and Cons

(a) MCMC

1. pros: The speed to converge to the optimal is fast.

   According to the distribution of each chains, we can find that the most frequent state of each chain is exactly the set with max reward, and the gap between the optimal and the suboptimal is large, because the distribution $\pi(X) \propto \lambda^{R(X)}$ is an exponential distribution. That means most of the time the chain will stay at the set with the max reward, which is faster to converge to the optimality.

2. cons: Easy to fall into the trap of local optimum.

If the λ is not proper. For example, if λ is large, then the chain may fall into the local optimum. If λ is small, the frequency of the maximum independent set may be small, which will not satisfy our expectation.

(b) Cross-Entropy

1. pros: Low bias. Has the best accuracy among the three algorithms
2. cons: High variance. The value of reward and the selection of the set fluctuate relatively greatly.

(c) MCTS

1. pros:

   (1) Not likely to converge to the locality optimality.

   (2) Due to pruning algorithm, a lot of useless exploration can be avoided.

2. cons:

   (1) Limited by the depth and width of exploration, it is likely for MCTS to have some bias.

2. Paper Summary

(a) Human-level control through deep reinforcement learning

$\textbf{Core idea}$: Deep Q-network

$\textbf{Contributions}$:

1. Derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations.
2. Learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning
3. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks

$\textbf{Surprising}$:

1. High-dimension: combining the traditional linear reinforcement with the neural network, DQN makes the reinforcement able to solve the high-action-dimensional problem.
2. Best performing and Stable: DQN agent performed at a level that was comparable to that of a professional human games tester across the set of 49 games, achieving more than 75% of the human score on more than half of the games
3. Minimal prior knowledge: DQN can successfully learn control policies in a range of different environments with only very minimal prior knowledge, receiving only the pixels and the game score as inputs, and using the same algorithm, network architecture and hyperparameters on each game, privy only to the inputs a human player.
4. Experience replay: DQN stores the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$, in a data set $D_t = e_1, \ldots, e_t$, pooled over many episodes (where the

end of an episode occurs when a terminal state is reached) into a replay memory.

5. Frame-skipping technique: In DQN, the agent sees and selects actions on every k_{th} frame instead of every frame, and its last action is repeated on skipped frames. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network.
6. Separate network: Use a separate network for generating the targets y_j in the Q-learning update. This modification makes the algorithm more stable compared to standard online Q-learning, where an update that increases $Q(s_t, a_t)$ often also increases $Q(s_{t+1}, a)$for all a and hence also increases the target y_j, possibly leading to oscillations or divergence of the policy.
7. Clipping the squared error to be between -1 and 1 to improve the stability.

Difficult:

1. In this paper, DQN uses a function approximator to estimate the action-value function, $Q(s,a;\theta) \approx Q(s,a)$, where the approximator is not the traditional linear function approximator, but a neural network.
2. However, DQN cannot be used for continuous control problems, because in the loss function between the optimal target values $r + \gamma \underset{a'}{max}\, Q^*(s', a')$ and the approximate target values $r + \gamma \underset{a'}{max}\, Q(s', a'; \theta_i^-)$ is

$$L_i(\theta_i) = E_{s,a,r,s'}(y - Q(s,a;\theta_i))^2 + E_{s,a,r}(V_{s'}(y))$$

Differentiating the loss function with respect to the weights we arrive at the following gradient to update the approximating parameters:

$$\Delta_{\theta_i} L(\theta_i) = E_{s,a,r,s'}((r + \gamma \underset{a'}{max}\, Q(s', a'; \theta_i^-) - Q(s,a;\theta_i \Delta_{theta_i} Q(s,a;\theta_i))))$$

In the gradient the $\underset{a'}{max}\, Q(s', a')$ function can only handle discrete situation. When in continuous situation, DQN may not converge, which is a drawback of DQN.

Whether convincing:

In fact, for games with high-dimensional action spaces, DQN has incomparable advantages over other traditional algorithms, which has been recognized. Therefore, from the test result, it is convincing.

Applied in other paper:

1.The DDPG algorithm inherits the DQN algorithm's idea of combining reinforcement learning with neural networks, using neural networks to estimate action-value functions to solve high-dimensional action space problems. But DDPG introduces the idea of Actor-Critic algorithm to solve the problem. 2.The Actor-Critic algorithm uses the idea of separating the network in the DQN algorithm, which means Actor-Critic algorithm separates the policy network(Actor) and value network (Critic) to enhance the stability of algorithm.

(b) Mastering the game of Go with deep neural networks and tree search

Core idea: AlphaGo

Contributions: the first time that a computer program has defeated a human professional player in the full-sized game of Go.

Combine Monte Carlo simulation(Monte Carlo Tree search) with value and policy networks

Procedure of AlphaGo:

1. Choose First, traverse down from the root of the tree, choosing the move with the highest confidence each time until the leaf node. The confidence is composed of the Q-value stored in each node and the prior probability P given by the policy network.

2. Extend and evaluate After reaching the leaf node, the tree needs to be expanded, the current situation is evaluated by the strategy network and the value network, and the node with the highest probability is added to the search tree.

3. Backtracking Backtrack the value v of the newly added node to the Q-value of each node on the path

Aim:

Minimize the loss function l that sums over mean-squared error and cross-entropy losses:

$$(\mathbf{p}, v) = f_\theta(s), l = (z-v)^2 - \pi^T log\mathbf{p} + c||\theta||^2$$

Surprising:

1. Monte Carlo Tree Search: using Monte Carlo rollouts to estimate the value of each state in a search tree

2. Combines policy and value network in MCTS:

At each time step t of each simulation, an action at is selected from state s_t:

$$a_t = \underset{a}{argmax}(Q(s_t, a) + u(s_t, a))$$

So as to maximize action value plus a bonus

$$u(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$$

Policy network is used to reduce the search width and only consider the method recommended by the network,

value network is used to reduce the depth of the search tree, you can replace the search subtree with a value to indicate the probability of winning the position

1. Evaluate the leaf node in two different ways: first, by the value network $v_\theta(s_L)$; and second, by the outcome z_L of a random rollout played out until terminal step T using the fast rollout policy p_π; these evaluations are combined, using a mixing parameter λ, into a leaf evaluation $V(s_L)$:

After test, we find that mixed evaluation $(\lambda = 0.5)$ performed best The two position-evaluation mechanisms are complementary: the value network approximates the outcome of games played by the strong but impractically slow p_ρ, while the rollouts can precisely score and evaluate the outcome of games played by the weaker but faster rollout policy p_π.

1. Combine the SL policy network and the MCTS: begin by training a supervised learning (SL) policy network p_ρ directly from expert human moves. This provides fast, efficient learning updates with immediate feedback and high-quality gradients. Next, we train a reinforcement learning (RL) policy network p_ρ that improves the SL policy network by optimizing the final outcome of games of self-play.

Difficult:

1. Larger networks achieve better accuracy but are slower to evaluate during search.
2. Evaluating policy and value networks requires several orders of magnitude more computation than traditional search heuristics.

Sol:

AlphaGo uses an asynchronous multi-threaded search that executes simulations on CPUs, and computes policy and value networks in parallel on GPUs.

Confusing:

1. Why mixed evaluation $(\lambda = 0.5)$ performed best, is there any theoretical proof?
2. Why alphago collapsed when played against Li Sedol?

Whether convincing:

It is certain that alphago with its MCTS theory is a huge feat, and its theory is successful.

However, when alphago played against Li Sedol, there was a situation where the alphago collapsed, indicating that alphago still has certain problems, and it is not as perfect as the paper says.

Applied in other paper:

1.Derived from AlphaGo, AlphaGo Zero still utilizes the MCTS. However, AlphaGo Zero abandoned the supervised learning part, which means it can master the go without human's knowledge.

(c) Mastering the game of Go without human knowledge

Core idea: AlphaGo Zero

Contributions:

Search-based Policy Iteration, Combine policy network and value network, residual network

Surprising:

1. AlphaGo Zero is based solely on reinforcement learning, without human data, guidance or domain knowledge beyond game
2. It uses only the black and white stones from the board as input features

3. It uses a single neural network, this neural network combines the roles of both policy network and value network into a single architecture rather than separate them in AlphaGo .
4. It uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte Carlo rollouts.
5. Residual network architecture rather than convolutional network architecture in AlphaGo

Search-based Policy Iteration:

1. Search-based Policy Improvement:

   (1) Run MCTS search using current network (to get the result of one game)

   (2) Actions selected by MCTS > actions selected by raw network

2. Search-based Policy Evaluation

   (1) Play self-play games with AlphaGo search

   (2) Evaluate improved policy by the average outcome (to train neural network)

Aim:

Minimize the loss function l that sums over mean-squared error and cross-entropy losses:

$$(\mathbf{p}, v) = f_\theta(s), l = (z-v)^2 - \pi^T log\mathbf{p} + c||\theta||^2$$

Difficult:

Computation power: AlphaGo Zero used a single machine with 4 tensor processing units (TPUs)29, whereas AlphaGo Lee was distributed over many machines and used 48 TPUs

Whether convincing:

The new MCTS utilized a network combined with policy and value network, which makes the self-play much more efficient, which improves the performance of AlphaGo Zero. What’s more, the residual network architecture makes iteration take full advantage of prior results. All above makes the AlphaGo Zero performs better. And the competition with Kejie proved its effectiveness.

Applied in other paper:

1. AlphaZero inherits the self-learning ability of alphago zero and combination of deep neural network and MCTS.

(d) A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play

Core idea: AlphaZero

Contributions:

1. Accommodates, without special casing, a broader class of game rules;
2. Uses a general-purpose Monte Carlo tree search (MCTS) algorithm instead of Alpha-beta search.

Surprising:

1. Accommodates a broader class of game rules: In AlphaZero, we reuse the same hyper - parameters, algorithm settings, and network architecture for all games without game-specific tuning. The only exceptions are the exploration noise and the learning rate schedule
2. Doubts about Alpha-beta search: Before the advent of alphazero, the alphabeta algorithm was considered a very good pruning algorithm. But alphazero uses the MCTS algorithm to replace the alpha-beta algorithm and achieves better results
3. Fewer search than other programme: AlphaZero may compensate for the lower number of evaluations by using its deep neural network to focus much more selectively on the most promising variations.
4. Bayesian Optimization: In AlphaZero, we reuse the same hyperparameters, algorithm settings, and network architecture for all games without game-specific tuning. The only exceptions are the exploration noise and the learning rate schedule

Similarity with AlphaGo zero:

Their core algorithm is similar (deep neural network and MCTS): AlphaZero uses a deep neural network $(p, v) = f_q(s)$ with parameters q. This neural network $f_q(s)$ takes the board position s as an input and outputs a vector of move probabilities p with components $p_a = Pr(a|s)$ for each action a and a scalar value v estimating the expected outcome z of the game from position s, $v \approx E(z|s)$. AlphaZero learns these move probabilities and value estimates entirely from self-play; these are then used to guide its search in future games.

Differences: AlphaZero uses a general purpose Monte Carlo tree search (MCTS) algorithm

Difficult:

1. Computation Power: During training only, 5000 first generation tensor processing units (TPUs) were used to generate self-play games, and 16 second-generation TPUs were used to train the neural networks.

Confusion:

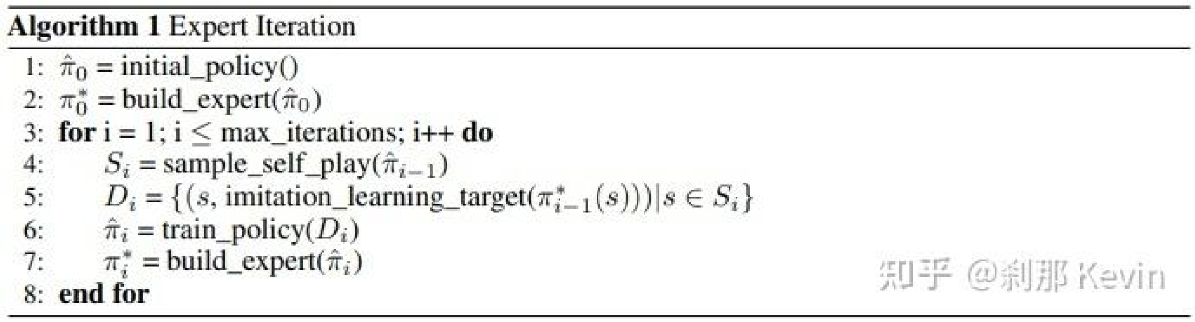
1. Why we need to maximize the similarity of the policy vector pt to the search probabilities pt
2. Why AlphaZero performs much better than AlphaGo Zero, assuming their core algorithm is similar (deep neural network and MCTS)

Whether convincing:

In experiment and competition with other programmes such as AlphaGo Zero, Crazy Stone, AlphaZero exhibits extremely high robustness and excellent performance. Therefore, its algorithm implementation is credible.

Applied in other paper:

1. In fact, AlphaZero draws on the “expert iteration” algorithm, where MCTS is expert and neural network is apprentice.



Algorithm 1 Expert Iteration

```
1: $\hat{\pi}_0$ = initial_policy()
2: $\pi_0^*$ = build_expert($\hat{\pi}_0$)
3: for i = 1; i $\leq$ max_iterations; i++ do
4:     $S_i$ = sample_self_play($\hat{\pi}_{i-1}$)
5:     $D_i = \{(s,$ imitation_learning_target$(\pi_{i-1}^*(s)))|s \in S_i\}$
6:     $\hat{\pi}_i$ = train_policy($D_i$)
7:     $\pi_i^*$ = build_expert($\hat{\pi}_i$)
8: end for
```

3. Gomoku

(a) Gomoku with forbidden rules

The code of AlphaZero with forbidden rule is in the folder "**AlphaZero_Gomoku_with_forbidden_rule**". The relative instruction is in the "README.md" file in the folder.

The main change is to add the forbidden rules (mainly in python file "gomoku.py")

Possible improvement:

In the step of MCTS backtracking, we can use Actor-Critic Algorithm to optimize the estimation of prior probability $P(s,a)$.

(b) AlphaZero without neural network

The code of AlphaZero without neural network is in the python file "mcts_pure.py" in the folder "AlphaZero_Gomoku_with_forbidden_rule". The relative instruction is in the "README.md" file in the folder.

In MCTS without neural network: we estimate the winning rate (also represents the value of (s,a)) $Q(s,a)$ by simulation to the end of game and based on the winning rate, we choose the next action. However, the method of simulation is to choose the next action randomly until the end of one game. This randomness introduces a large bias to the value of each action.

While in MCTS with neural network, we simulate the final result based on both the value and its upper confidence bound. And in the process of simulation, we will use the prior probability of (s,a): $P(s,a)$, which is generated by neural network, to estimate the upper confidence bound:

$$U(s,a) = c_{puct}P(s,a)\frac{\sqrt{N(s,b)}}{1+N(s,a)}$$

Based on the value with upper confidence bound, we choose the optimal approximate action a: $a = \underset{a}{argmax}(Q(s,a)+U(s,a))$, where $Q(s,a)$ is the value of (s,a) (or winning rate).

In these process, the Neural network is used to estimate the prior probability $P(s,a)$, and further estimate the choice of action.

After the end of simulation of one game and get the final result of one game, we can backtrack the neural network to update the probability $P(s,a)$ to be more accurate.

Through the iteration of neural network, we can estimate a relatively accurate win probability for each move.

While in the MCTS without neural network, we can only estimate the prior probability $P(s, a)$ with a random value or completely finish the game. The former alternative has a large error, and the latter alternative has much more time cost. In order to reduce the time cost, we choose to take a random value for the prior probability, as shown in the following figure (the function "rollout_policy_fn" in "mcts_pure.py"):

In [ ]:
```
def rollout_policy_fn(board):
    """a coarse, fast version of policy_fn used in the rollout phase."""
    # rollout randomly
    action_probs = np.random.rand(len(board.availables))
    return zip(board.availables, action_probs)
```

In this case, the accuracy of prior probability $P(s, a)$ with neural network is certainly better than those without neural network. Therefore, in each simulation, AlphaZero with neural network can always choose the better action with more value and performs better. We can also verify that through simulation. We let the AlphaZero with the neural network play against the one without the neural network :

```
Start game 0
The winner is player 1
Start game 1
The winner is player 1
Start game 2
The winner is player 1
Start game 3
The winner is player 1
Start game 4
The winner is player 1
Start game 5
The winner is player 1
Start game 6
The winner is player 1
Start game 7
The winner is player 1
Start game 8
The winner is player 1
Start game 9
The winner is player 1
Start game 10
The winner is player 1
Start game 11
The winner is player 1
Start game 12
The winner is player 1
Start game 13
The winner is player 1
Start game 14
The winner is player 1
Start game 15
The winner is player 1
Start game 16
The winner is player 1
Start game 17
The winner is player 1
Start game 18
The winner is player 1
Start game 19
The winner is player 1
Start game 20
The winner is player 1
```

Through the result, we can find that in 60 games, the AlphaZero with the neural network won 60 games, while the AlphaZero without the neural network only won 0 games. The result proves that the AlphaZero with neural network performs much better than the one without neural network.

(c) Experience share

1. A core idea of RL reflected in MCTS is "exploration and exploitation trade-off".

   In MCTS, we choose the next action based on the value with upper confidence bound:

   \begin{align*}

   ```
    a = \mathop{argmax}\limits_{a}(Q(s,a)+U(s,a))
   ```
   \end{align*}

And the upper confidence bound of (s,a) is:

\begin{align*}

```
 U(s,a) = c_{puct}P(s,a)\frac{\sqrt{N(s,b)}}{1+N(s,a)}
```
\end{align*}

The bound reflects the "exploration and exploitation trade-off": the winning rate $Q(s,a)$ represents the value of action a, while the upper confidence bound $U(s,a)$ represents the degree of our exploration of action a. For example, for the action a which we have not explore yet, the $N(s,a) = 0$ and the upper confidence bound will be large, which promote the exploration of action a.

2. In the process, the understanding of MCTS has been further deepened:

   The procedure of MCTS:

   (1) Selection: Starting from the root node R, recursively select the optimal child node (explained later) until the leaf node L is reached.

   (2) Expansion: If L is not a termination node (i.e., does not cause the game to terminate) then create one or more subnodes of the word, choosing one of C.

   (3) Simulation: The output of running a simulation from C until the end of the game.

   (4) Backpropagation: Update the current action sequence with the resulting output of the simulation

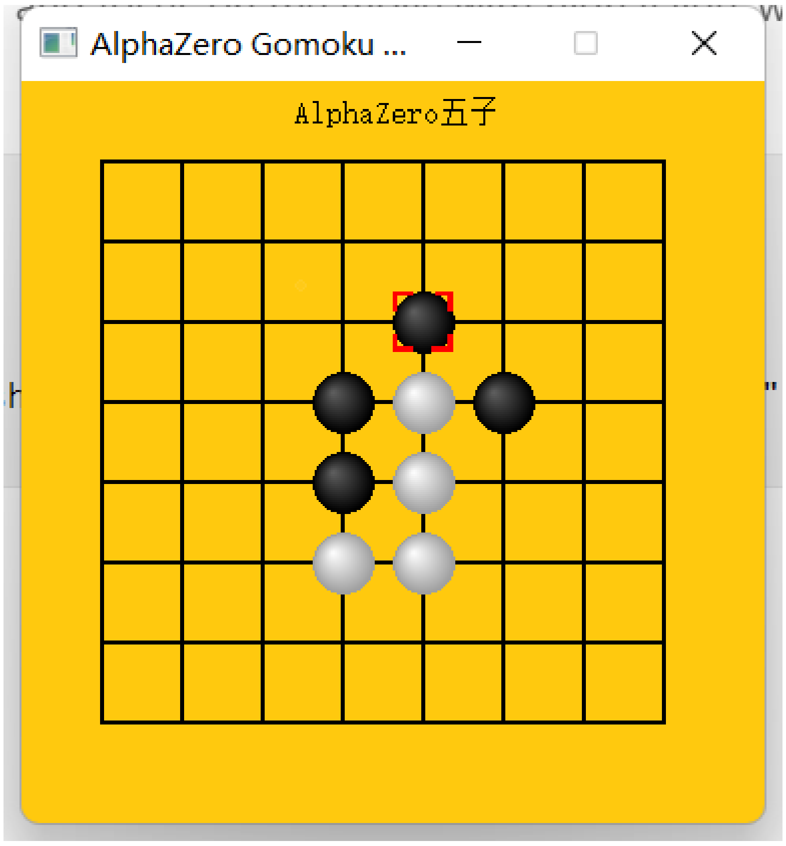
   In AlphaZero, the neural network has been added to estimate the value of each action, which improve the performance of MCTS.

3. One of the main idea of MCTS reflected in AlphaZero is pruning, which means after much exploration, the value with upper confidence bound of bad move will be small, to decrease the probability of choosing them to a small value. In this case, under a certain degree of exploration, the agent can avoid much unnecessary exploration and focus on the move with high value, which construct the process of pruning.

(d) Innovative part

1. User-interface

   A user-interface will be shown after running python file "game.py" in folder "AlphaZero_Gomoku_with_forbidden_rule". The user can choose to play against either an AI or another human player.



2. MuZero In a paper published by deepmind: "Mastering Atari, Go, chess and shogi by planning with a learned model, the algorithm of MuZero is introduced in detail, and it is confirmed that muzero has a better performance than alphazero in learning Go.

   Different from AlphaZero with knowledge of rules of game, MuZero has no knowledge with these rules. Therefore, MuZero takes a different approach, using a "representation network" h to convert the observed chessboard state O into its own hidden state S_0.

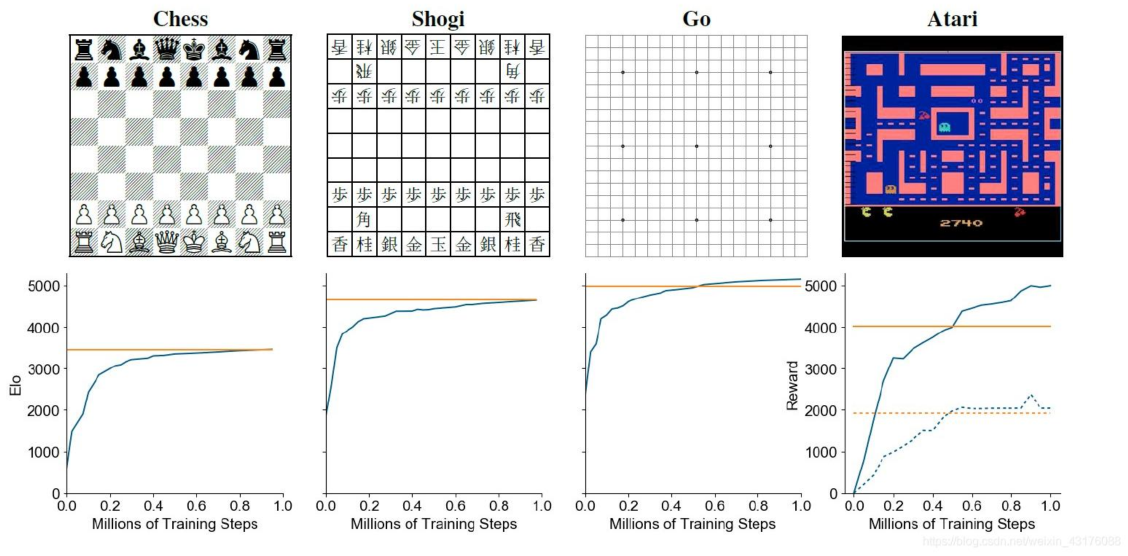
   With the hidden state S_0, MuZero can calculate the various actions that can be made in the S_0 state (after learning, the possible actions obtained by MuZero will become more and more in line with the rules), and can pass the "dynamic network" (dynamic network) g, Using MuZero's chosen action and the current latent state S0, deduce S1. At each real-world time point, MuZero can use its internal model and MCTS to select the best behavior at that time point and apply it to the real world. After actually making an optimal behavior, this behavior can be "recycled" and used to train the dynamic network g (thus improving MuZero's internal model).

   Another optimization used in MuZero is "Reanalyse":

   Re-running MCTS, keeping the trajectory (observations, actions, and rewards) unchanged, generates new search statistics that provide new goals for policy and value prediction.

   Because during direct interaction with the environment, searching with an improved network yields better statistics. Similarly, re-searching with the improved network on existing trajectories also yields better statistics, allowing repeated improvements using the same trajectory data.

   Therefore, without knowing the rules of the game, MuZero can simulate the rules through the hidden state, and through Monte Carlo tree search, make accurate choices in the real environment. The comparison of performance between MuZero and AlphaZero is as below:



We can find that after a certain amount of training, the performance of MuZero (Blue line) will exceed the AlphaZero (Orange line), which means the MuZero without the knowledge of game rules has better performance than AlphaZero.