A Dive to MCTS

liuhanzuo

December 15, 2024

Abstract

TODO

1 BackGround

1.1 MDP

We start the background part by introducing Markov Decision Process(MDP). The basic elements of MDP are as follows:

- State Space $\mathcal{S}$: The set of all possible states.
- Action Space $\mathcal{A}$: The set of all possible actions.
- Transition Probability $T(s', s, a)$: The probability of transitioning to state s' given that the current state is s and the action taken is a.
- Reward Function $R(s, a, s')$: The reward received after transitioning from state s to state s' by taking action a.

The goal of MDP is to find a policy π that maximizes the expected cumulative reward. The policy π is a mapping from states to actions. Which indicates our strategy or policy for a given state. Here, we need to note that we usually choose the π to mamximize the reward.

1.2 Monte Carlo Methods

From the MDP we can see that: when we are making a decision, what we really care about the the "expected reward" or "potential reward" for taking an action a in state s. The Monte Carlo Methods is a class of algorithms that estimate the expected reward by sampling the environment. The basic idea is to simulate the environment and collect the reward. The expected reward is then estimated by the average of the collected rewards. The Monte Carlo Methods are widely used in reinforcement learning, game theory, and other fields.
To be more secific, we define the Q-value function as:

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{N(s)} \mathbb{I}_{s,a} R_i$$

Where the definition is:

- $N(s, a)$ is the number of times action a has been taken in state s.
- $N(s)$ is the number of times state s has been visited. i.e. $N(s) = \sum_a N(s, a)$
- $\mathbb{I}_{s,a}$ is the indicator function that is 1 if the action a is taken in state s and 0 otherwise.
- R_i is the reward received after taking action a in state s.
- $Q(s, a)$ is the estimated Q-value of taking action a in state s (expected reward).

Until here, we have already reached the inner core of the whole system – exploration. It means that how can we make a exploration strategy to adequately explore the state and get the corresponding Q-value(and the optimal policy). The Monte Carlo Methods is a good start for this problem.

2 Monte Carlo Tree Search

2.1 Introduction

This section introduces the family of algorithms known as Monte Carlo Tree Search (MCTS). MCTS rests on two fundamental concepts: that the true value of an action may be approximated using random simulation; and that these values may be used efficiently to adjust the policy towards a best-first strategy. The algorithm progressively builds a partial game tree, guided by the results of previous exploration of that tree. The tree is used to estimate the values of moves, with these estimates (particularly those for the most promising moves) becoming more accurate as the tree is built.
To be more specific, we will introduce some typical algorithm and list the corresponding result, also explain the intuition behind the algorithm.

2.2 General MCTS

The general MCTS algorithm is as follows:

Algorithm 1 General MCTS

```
1: function MCTS(s_0)
2:     create a node v_0 with state s_0
3:     while within computational budget do
4:         v_l ←TreePolicy(v_0), Δ ←DefaultPolicy(s(v_l))
5:         Backup(v_l, Δ)
6:     end while
7:     return BestChild(v_0)
8: end function
```

Basically, we have four steps:

1. `Selection`: Start from the root point, we use a policy – child selecting policy to select the next node to expand. (run the `TreePolicy` function)
2. `Expansion`: Expand the selected node and get the next node.

3. `Simulation`: A simulation about what would happen to run on the new point. (run the `DefaultPolicy` function). Typically, we would use a simple(default) strategy to find out the result.

4. `BackPropagation`: Update the information of the nodes on the path from the root to the leaf node.

This four steps can be also expained as a psuedo code at 1.
After the training steps (what we mentioned above), we get a strategy that label the reward of each action in the current state. For each state, we should run a `BestChild` function to get the best action to take.
The choice of `BestChild` function is mainly listed below:

1. `reward-based`: choose the child with highest-reward.

2. `visited-based`: choose the child with highest-visited times.

3. `hybrid-based`: mix the above two policy with a factor, balance the reward and visited times.

Afterwards, we will enumerate some specific algorithms based on the general MCTS algorithm – mainly focus on how to choose `TreePolicy` and `DefaultPolicy`.

2.3 Reward-based MCTS

Reward-based MCTS is the basic version of MCTS. It uses the reward to guide the exploration. The `TreePolicy` function selects the next node to expand based on the estimated Q-value of the nodes. The `DefaultPolicy` function uses a random policy to simulate the environment. The `Backup` function updates the Q-values of the nodes on the path from the root to the leaf node.
Each step, it chooses the `BestChild` with the highest Q-value to expand. The intuition behind this is that the children with higher Q-values are more likely to lead to a higher reward.
Since it is straight to understand this algorithm, we will not show the psuedo code here.
There is also some variant of this kind of MCTS:

- visited-based MCTS: simply change the choice of `BestChild` function to choose the child with highest visited times. The intuition for this algorithm is that during the exploration, the high-visited node is likely to be the right choice (and the reward is also high).

- randomized MCTS: with probability of p we randomly explore a point in the tree, with probability of $1 - p$ we choose the best child.

- hybrid MCTS: mix the above two policy with a factor, balance the reward and visited times. Use a factor α to control the balance.

2.4 UCT

The Upper Confidence bounds for Trees (UCT) algorithm [6] is a popular variant of MCTS. It uses the UCB1 formula to balance exploration and exploitation. The UCB1 formula is given by:

$$UCB1 = \frac{Q(s,a)}{N(s,a)} + C\sqrt{\frac{\ln N(s)}{N(s,a)}}$$

where C is a constant that controls the balance between exploration and exploitation. $N(s,a)$ is the number of times action a has been taken in state s. $N(s)$ is the number of times state s has been visited. $Q(s,a)$ is the estimated Q-value of taking action a in state s. The UCT algorithm uses the UCB1 formula to select the next node to expand.
Here let's consider the intuition for the UCT algorithm.

- For a point that is never visited before, the $N(s,a)$ term is zero, thus its UCB1 value is infinite large, will be more likely to expolore.
- For a point that is visited many times, the $Q(s,a)$ term will be more important, which means that the point with higher reward will be more likely to be selected.
- If all points in the graph is visited, the $N(s,a)$ term will be non-zero. Thus a larger C means that we focus more on exploration. A smaller C value, means that we focus more on exploitation.
- As a comparison to reward-based MCTS, the UCT algorithm solve the problem of insufficient exploration of node, which often cause reward-based MCTS to get stuck in a local minimum. Moreover, it consider the Q value and the visited times of the node, which is more reasonable.

Now we show the psuedo code for the UCT algorithm: ([1] is our reference).

Algorithm 2 UCT

```
1: function UCT(s_0)
2:     create a node v_0 with state s_0
3:     while within computational budget do
4:         v_l ←TreePolicy(v_0), Δ ←DefaultPolicy(s(v_l))
5:         Backup(v_l, Δ)
6:     end while
7:     return BestChild(v_0)
8: end function
9: function TreePolicy(v)
10:    while v is not a terminal node do
11:        if v is not fully expanded then
12:            return Expand(v)
13:        else
14:            v ←BestChild(v)
15:        end if
16:    end while
17:    return v
18: end function
19: function Expand(v)
20:    choose a ∈ A(s(v))
21:    add a new node v' as a child of v
22:    s(v') = f(s(v), a)
23:    a(v') = v
24:    return v'
```

end function
function BestChild(v)
    return

$$\arg \max_{v' \in \text{children}(v)} \left(\frac{Q(v')}{N(v')} + C\sqrt{\frac{\ln N(v)}{N(v')}} \right)$$

end function
function DefaultPolicy(s)
    while s is not a terminal state **do**
        choose $a \in A(s)$
        $s \leftarrow f(s, a)$
    end while
    return $R(s)$
end function
function Backup(v, Δ)
    while v is not null **do**
        $N(v) \leftarrow N(v) + 1$
        $Q(v) \leftarrow Q(v) + \Delta$
        $v \leftarrow a(v)$
    end while
end function

3 Experiment

In this part, to test the performance of different algorithms, I implement some games, train the model with certain epochs and let different algorithm to play against in the game.
The definitions are as follows:

- selection: the algorithm selected to train the model.
- game: the game that the algorithm will play.
- max-iter: the maximum computational budget for the algorithm (Can explore how many nodes in each training epoch)
- epochs: a certain value that the algorithm will train the model.

You can see the codes in `https://github.com/liuhanzuo/algorithm-design-survey`

3.1 Game: Tic-Tac-Toe

The Tic-Tac-Toe is a simple game that two players take turns to place their mark on a 3x3 board. The player who first places three marks in a row, column, or diagonal wins the game. The game ends in a draw if the board is full and no player has won.
Note that reward means the reward based model, visited means the visited based model, UCT means the UCT model, and Beta means the randomized model, with $p = 0.1$, while the test part is implemented by the reward-based model. The baseline is the reward-based model, with `iter-max=80` and `epochs=1000`. The pictures is ploted from `iter-max=10` to `iter-max=120` with

step 10.
The conclusion is that UCT has the best performance, while the visited-based model has the worst performance. The reward-based model and the randomized model has a similar performance. Also, when two players have a similar power, they are more likely to reach a draw during the game.

4 Algorithm Variations

In this part, we will introduce more algorithms based on the UCB/UCT algorithm.

4.1 Bandit Algorithm

The definitions we used are as follows:

- $X_{i,n_i} = \frac{1}{n_i}\sum_{t=1}^{n_i} x_t$
- $X_{d',n_{d'}}$: The reward received at node d' after $n_{d'}$ visits.
- n_d: The number of times the node of depth d in the optimal branch is reached.
- n: The first instant when the optimal leaf is reached.
- D: The depth of the tree.
- s: A constant used in the confidence sequence.
- $$B_{i,p,n_i} =^{\text{def}} X_{i,n_i} + \sqrt{\frac{2\log(p)}{n_i}}$$

The pseudeo Code for bandit algorithm is as follows:

Algorithm 3 Bandit Algorithm

for $n \geq 1$ **do**
    Run $n-$th trajectory from the root to leaf
    set current node i_0 to root
    for $d = 1 \cdots D$ **do**
        Select node i_d as the children j of node i_{d-1} that maximizes $B_{j,n_{i_{d-1}},n_j}$
    end for
end for
receive reward $x_n \sim^{\text{iid}} X_{i_D}$
for $d = D \cdots 0$ **do**
    Update the number of visits $n_{i_d} = n_{i_d} + 1$
    Update the bound $B_{i_d,n_{i_{d-1}},n_{i_d}}$
end for

Coquelin and Munos propose flat UCB which effectively treats the leaves of the search tree as a single multiarmed bandit problem.[2] This is distinct from flat Monte Carlo search, in which the actions for a given state are uniformly sampled and no tree is built. Coquelin and Munos demonstrate that flat UCB retains the adaptivity of standard UCT while improving its regret bounds in certain worst cases where UCT is overly optimistic.

Intuitionlly, UCT may spend many time exploring an area that there is no best reward. Let's provide an example: the Monte Carlo tree has m stages, while the i-th stage has a decision to get a reward of $\frac{m-(i+1)}{m}$ directly, while in the mth stage, you could get a reward of 1. However, since the UCT algorithm will tend to visit the first stage with a reward of $\frac{m-1}{m}$, it will cost the UCT algorithm many times to explore the last stage whose reward is 1. (It will take UCT $O(\exp(\exp m))$ time).

We now establish a lower bound on the number of times suboptimal rewards are received before getting the optimal 1 reward for the first time. Write n the first instant when the optimal leaf is reached. Write n_d the number of times the node (also written d making a slight abuse of notation) of depth d in the optimal branch is reached. Thus $n = n_0$ and $n_D = 1$. At depth $D-1$, we have $n_{D-1} = 2$ (since action 2 has been chosen once in node $D-1$).

We consider both the logarithmic confidence sequence used in (1) and the square root sequence in (2). Let us start with the square root confidence sequence (2). At depth $d-1$, since the optimal branch is followed by the n-th trajectory, we have (writing d' the node resulting from action 2 in the node $d-1$):

$$X_{d',n_{d'}} + \frac{s\sqrt{n_{d-1}}}{n_{d'}} \leq X_{d,n_d} + \frac{s\sqrt{n_{d-1}}}{n_d}.$$

But $X_{d',n_{d'}} = \frac{D-d}{D}$ and $X_{d,n_d} \leq \frac{D-(d+1)}{D}$ since the 1 reward has not been received before. We deduce that

$$\frac{1}{D} \leq \frac{s\sqrt{n_{d-1}}}{n_d}.$$

Thus for the square root confidence sequence, we have $n_{d-1} \geq \frac{n_d^2}{D^4}$. Now, by induction,

$$n \geq \frac{n_1^2}{D^4} \geq \frac{n_2^2}{2D^4(1+2)} \geq \frac{n_3^2}{3D^4(1+2+3)} \geq \cdots \geq \frac{n_{D-1}^2}{(D-1)D^{2D(D-1)}}.$$

Since $n_{D-1} = 2$, we obtain $n \geq \frac{2^{2D-1}}{D^{2D(D-1)}}$. This is a double exponential dependency with respect to D. For example, for $D = 20$, we have $n \geq 10^{156837}$. Consequently, the regret is also $\Omega(\exp(\exp(D)))$.

Now, the usual logarithmic confidence sequence defined by (1) yields an even worse lower bound on the regret since we may show similarly that $n_{d-1} \geq \exp(n_d/(2D^2))$ thus

$$n \geq \exp(\exp(\cdots \exp(2) \cdots)) \quad \text{(composition of } D-1 \text{ exponential functions)}.$$

Thus, although UCT algorithm has asymptotically regret $O(\log(n))$ in n, (or $O(\sqrt{n})$ for the square root sequence), the transitory regret is $\Omega(\exp(\exp(\cdots \exp(2) \cdots)))$ (or $\Omega(\exp(\exp(D)))$ in the square root sequence).

The reason for this bad behavior is that the algorithm is too optimistic (it does not explore enough and may take a very long time to discover good branches that looked initially bad) since the bounds (1) and (2) are not true upper bounds.

5 Learning in MCTS

5.1 Temporal Difference Learning

The Temporal Difference (TD) learning algorithm [5] is a model-free reinforcement learning algorithm that learns the value function by bootstrapping. Based on the Bellman equation, which states that the value of a state is equal to the immediate reward plus the discounted value of the next state. Also, it uses the difference between the estimated value and the target value to update the value function.
Here we show the psuedo code for the TD learning algorithm:

Algorithm 4 Temporal Difference Learning

```
1: function TD(s_0)
2:     initialize the value function V(s)
3:     s ← s_0
4:     while s is not a terminal state do
5:         choose a from A(s)
6:         s' ← f(s, a)
7:         r ← R(s, a, s')
8:         V(s) ← V(s) + α(r + γV(s') − V(s))
9:         s ← s'
10:    end while
11: end function
```

the algorithm 4 shows a typical kind of TD learning algorithm. The α is the learning rate, γ is the discount factor. r refers to the immediate reward at current state and s' is the next state.
Intuition The main difference between the TD learning algorithm and the MCTS algorithm is that the TD learning algorithm learns the value function by bootstrapping, which means that TD does not need a model of the environment. It will obtain the reward by the environment and update the value function.
Problems Gerald Tesauro proposes that [4] the TD learning algorithm has a problem of high variance. The reason is that the TD learning algorithm uses the reward received from the environment to update the value function. The reward is often noisy and may not be accurate. This leads to a high variance in the value function. Additionally. In fact, TD methods can suffer from instability and divergence, especially when combined with function approximation and off-policy learning.
As a result, when Gerald Tesauro trains the TD learning algorithm to play backgammon, his network (a feedforward fully-connected architecture with either no hidden units, or a single hidden layer with between 10 and 40 hidden units) had only 40% to 60% win rate against the best human players.
The TDMC algorithm is given by Yasuhiro OSAKI [3].
Intuition The main idea of TDMC is to use the Monte Carlo method to estimate the value function and approximate the winning rate for each state without knowing the actual action. The main difference between TDMC and TD is that TDMC uses Monte Carlo simulations to estimate the value function. This approach reduces the variance in the value function estimation by averaging the results of multiple simulations. By running multiple simulations from a given state, TDMC can obtain a more accurate estimate of the expected reward, leading to better policy evaluation and improvement. This method combines the strengths of both Temporal Difference learning and

Monte Carlo methods, providing a more stable and reliable learning process.
Now we show the psuedo code for the TDMC algorithm:

Algorithm 5 TDMC Algorithm

function TDMC($s_0, \alpha, \gamma, \lambda, T$)
  initialize the feature weights w
  for each episode **do**
    initialize the state $s \leftarrow s_0$
    initialize the eligibility traces $e \leftarrow 0$
    for $t = 1, 2, \ldots, T-1$ **do**
      observe features x_t of state s
      compute the value $V(x_t, w) \leftarrow x_t \cdot w$
      perform action a and observe reward r_t and next state s'
      simulate the environment to get r_i for $i = t+1, \ldots, T$
      compute the return $R_t \leftarrow \sum_{i=t}^{T-1} \gamma^{i-t} r_i$
      compute the n-step return $(n)R_t \leftarrow r_t + \gamma r_{t+1} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(x_{t+n}, w)$
      compute the λ-return $\lambda R_t \leftarrow (1-\lambda) \sum_{n=1}^{T-t} \lambda^{n-1} (n)R_t + \lambda^{T-t} R_t$
      compute the gradient $\nabla V(x_t, w)$
      update the eligibility traces $e \leftarrow \gamma \lambda e + \nabla V(x_t, w)$
      update the weights $w \leftarrow w + \alpha(\lambda R_t - V(x_t, w))e$
      $s \leftarrow s'$
    end for
  end for
end function

6 UCT-based algorithm and Tree MCTS

6.1 UCB1-tuned

The UCB1-Tuned algorithm is an enhancement suggested by Auer et al. [**?**] to tune the bounds of UCB1 more finely. It replaces the upper confidence bound $\sqrt{\frac{2 \ln n}{n_j}}$ with:

$$\sqrt{\frac{\ln n}{n_j} \min\left(\frac{1}{4}, V_j(n_j)\right)}, V_j(s) = \left(\frac{1}{2} \sum_{\tau=1}^{s} X_{j,\tau}^2 - \overline{X}_{j,s}^2 + \sqrt{\frac{2 \log t}{s}}\right)$$

where $V_j(n_j)$ is the empirical variance of the rewards obtained from action j after n_j plays. The UCB1-Tuned algorithm aims to reduce the exploration of suboptimal actions by incorporating the variance of the rewards into the confidence bounds.
The pseudo code for the UCB1-Tuned algorithm is as follows:

Algorithm 6 UCB1-Tuned

function UCB1-TUNED(s_0)
  create a node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$TREEPOLICY(v_0), $\Delta \leftarrow$DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)

end while
return BestChild(v_0)
end function
function TreePolicy(v)
while v is not a terminal node **do**
if v is not fully expanded **then**
return Expand(v)
else
$v \leftarrow$BestChild(v)
end if
end while
return v
end function
function Expand(v)
choose $a \in A(s(v))$
add a new node v' as a child of v
$s(v') = f(s(v), a)$
$a(v') = v$
return v'
end function
function BestChild(v)
return

$$\arg \max_{v' \in \text{children}(v)} \left(\frac{Q(v')}{N(v')} + \sqrt{\frac{\ln N(v)}{N(v')} \min \left(\frac{1}{4}, \frac{V(v')}{N(v')} + \sqrt{\frac{2 \ln N(v)}{N(v')}} \right)} \right)$$

end function
function DefaultPolicy(s)
while s is not a terminal state **do**
choose $a \in A(s)$
$s \leftarrow f(s, a)$
end while
return $R(s)$
end function
function Backup(v, Δ)
while v is not null **do**
$N(v) \leftarrow N(v) + 1$
$Q(v) \leftarrow Q(v) + \Delta$
$v \leftarrow a(v)$
end while
end function

References

[1] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[2] Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. *arXiv preprint cs/0703062*, 2007.

[3] Yasuhiro Osaki, Kazutomo Shibahara, Yasuhiro Tajima, and Yoshiyuki Kotani. An othello evaluation function based on temporal difference learning using probability of winning. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 205–211, 2008.

[4] Gerald Tesauro. Practical issues in temporal difference learning. *Advances in neural information processing systems*, 4, 1991.

[5] Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[6] David Tolpin and Solomon Shimony. Mcts based on simple regret. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pages 570–576, 2012.