Planning Algorithms: An Overview

Zamboni Riccardo

e-Novia, Italy, ricc.zamboni@gmail.com

Keywords: Planning, MCTS, UCT, OLOP, UGapE

Abstract

Planning algorithms are gaining increasing interest, as they are a powerfull framework to deal with decision processes, and consist of a fruitfull intersection between games and more general policy based scenarios.

1 Introduction

Starting from a simple RL framework to make things more clear, a planning is conceived as when an agent uses some generative model of the environment to acquire more knowledge about the best policy to be followed, and then act accordingly. Planning algorithms are based on many different settings, and they can provide open-loop policies (independent from the state in which the agent is), and closed-loop policies, in which the actions are based on the stases they lead to. We will take into account various settings for the planning framework, starting from the most common formulation of the problem.

2 Multi-Armed Bandits

2.1 Overview

Imagine you are in a casino facing multiple slot machines and each is configured with an unknown probability of how likely you can get a reward at one play. The question is: What is the best strategy to achieve highest long-term rewards?

In this post, we will only discuss the setting of having an infinite number of trials. The restriction on a finite number of trials introduces a new type of exploration problem. For instance, if the number of trials is smaller than the number of slot machines, we cannot even try every machine to estimate the reward probability (!) and hence we have to behave smartly w.r.t. a limited set of knowledge and resources (i.e. time).

A naive approach can be that you continue to playing with one machine for many many rounds so as to eventually estimate the "true" reward probability according to the law of large numbers. However, this is quite wasteful and surely does not guarantee the best long-term reward.

Now let's give it a scientific definition.

A Bernoulli multi-armed bandit can be described as a tuple of $\langle \mathcal{A}, \mathcal{R} \rangle$, where:

- We have K machines with reward probabilities, $\theta_1, \cdots, \theta_K$
- At each time step t, we take an action a on one slot machine and receive a reward r.
- A is a set of actions, each referring to the interaction with one slot machine. The value of action a is the expected reward, $Q(a) = E[r|a] = \theta$. If action at at the time step t is on the i-th machine, then $Q(a_t) = \theta_i$
- R is a **reward function**. In the case of Bernoulli bandit, we observe a reward r in a stochastic fashion. At the time step t, $r_t = R(a_t)$ may return reward 1 with a probability $Q(a_t)$ or 0 otherwise.

It is a simplified version of Markov decision process, as there is no state S, and it well describes many decision processes, as it is in the planning framework. The goal is to maximize the cumulative reward $\sum_{t=1}^{T} r_t$. If we know the optimal action with the best reward, then the goal is same as to minimize the potential regret or loss by not picking the optimal action. The optimal reward probability θ^* of the optimal action a^* is:

$$\theta^* = Q(a^*) = \max_{a \in A} Q(a) = \max_{1 \le i \le K} \theta_i$$

Our loss function is the total regret we might have by not selecting the optimal action up to the time step T:

$$\mathcal{L}_T = E[\sum_{t=1}^{T} (\theta^* - Q(a_t))]$$

2.2 Upper Confidence Bounds and UCB1

An usefull concept which is often take into account when dealing with planning and Bandits is the one of the confidence bounds, and in particular the upper one. Random exploration gives us an opportunity to try out options that we have not known much about. However, due to the randomness, it is possible we end up exploring a bad action which we have confirmed in the past (bad luck!). To avoid such inefficient exploration, one approach is to be optimistic about options with high uncertainty and thus to prefer actions for which we haven't had a confident value estimation yet. Or in other words, we favor exploration of actions with a strong potential to have a optimal value.

The **Upper Confidence Bounds** (UCB) algorithm measures this potential by an upper confidence bound of the reward value,$U^t(a)$, so that the true value is below with bound $Q(a) \leq Q^t(a)+U^t(a)$ with high probability. The upper bound $U^t(a)$ is a function of $N_t(a)$; a larger number of trials $N_t(a)$ should give us a smaller bound $U^t(a)$.

In UCB algorithm, we always select the greediest action to maximize the upper confidence bound:

$$a_t^{UCB} = argmax_{a\in A} Q^t(a) + U^t(a)$$

Now, the question is how to estimate the upper confidence bound. If we do not want to assign any prior knowledge on how the distribution looks like, we can get help from **Hoeffding's Inequality**, a theorem applicable to any bounded distribution.

Let $X_1, \cdots, X_t$ be i.i.d. (independent and identically distributed) random variables and they are all bounded by the interval $[0, 1]$. The sample mean is $\bar{X}_t = \frac{1}{t}\sum_{\tau=1}^{t} X_\tau$. Then for $u > 0$, we have:

$$\mathcal{P}[E[X] > \bar{X}_t + u] \leq \exp -2tu^2$$

Given one target action a, let us consider:

- $r_t(a)$ as the random variables,
- $Q(a)$ as the true mean,
- $Q^t(a)$ as the sample mean,
- u as the upper confidence bound, $u = U_t(a)$

Then we have,

$$\mathcal{P}[Q(a) > Q^t(a) + U_t(a)] \leq \exp -2tU_t(a)^2$$

We want to pick a bound so that with high chances the true mean is blow the sample mean + the upper confidence bound. Thus $\exp -2tU_t(a)^2$ should be a small probability. Let's say we are ok with a tiny threshold p:

$$e^{-2tU_t(a)^2} = p \text{ Thus, } U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

One heuristic is to reduce the threshold p in time, as we want to make more confident bound estimation with more rewards observed. Set $p = t^{-4}$ and we get **UCB1 Algorithm**:

$$U_t(a) = \sqrt{\frac{2\log t}{N_t(a)}} \text{ and } a_t^{UCB1} = \arg\max_{a\in\mathcal{A}} Q(a) + \sqrt{\frac{2\log t}{N_t(a)}}$$

3 Monte-Carlo Tree Search and UCT

In the development of planning algorithms, the approach based on the Monte-Carlo exploration has utterly gained attention. In general, Monte Carlo tree search (MCTS) algorithms employ strong (heuristic) search methods to identify the best available action in a given situation. They gain knowledge by simulating the given problem and thus require at least its generative model (also known as a forward, simulation, or sample model), which may be simpler than the complete transition model of a given task. The MCTS planning process incrementally builds an asymmetric search tree that is guided in the most promising direction by an exploratory action-selection policy, which is computed iteratively. It improves when increasing the number of iterations. An MCTS iteration usually consists of four consecutive phases: (1) selection of actions already memorized in the tree (descent from the root to a leaf node), (2) expansion of the tree with new nodes, (3) playout, i.e., selection of actions until a terminal state is reached, and (4) backpropagation of the feedback up the tree. A tree policy guides the selection phase and a default policy guides the playout phase.

Subsequently to the planning, the starting move leading most frequently to a win is played. The efficiency of Monte-Carlo search heavily depends on the way moves are sampled during the episodes. A pseudo-code for Monte-Carlo Search is showed in 1.

```
procedure MonteCarloSearch(position)
repeat
    search(position, rootply)
until Timeout
return bestMove(position);

function search(position, depth)
if GameOver then
    return GameResult
end if
m := selectMove(position, depth);
v := −search(position after move m, depth + 1);
Add entry (position, m, depth, v, ...) to the TT
return v;
```

Figure 1. MCTS Pseudo-Code

UCT is a Monte-Carlo search algorithm with a specific randomized move se- lection mechanism based on the UCB1 algorithm, which can be exploited to find a meaningfull way to select which move/action to perform, modelling the move selection problem as a separate multi-armed bandit for every (explored) internal node. In order to find the best move in the root, one has to determine the best moves in the internal nodes as well (at least along the candidate principal variations). Since the estimates of the values of the alternative moves rely on the estimates of the values of the (best) successor nodes, we must have small estimation errors for the latter ones. Hence the problem reduces to getting the estimation error decay quickly. In order to achieve this, the algorithm must balance between testing an alternative that looks currently the best (to obtain a precise estimate) and the exploration of other alternatives (to ensure that some good alternative is not missed), as guaranteed from the UCB1 framework. The pseudo-algorithm of the UCT is showed in 2.

4 Temporal Differences Extensions and SARSA-UCT(λ)

It is possible to include temporal differences formalisms in the MCTS framework. In particular, it is possilbe to make assumptions on the missing values of playout states (and actions) in a similar way as assumptions can be made on the initial values $V_{init}(s)$ and $Q_{init}(s; a)$. We introduce the notion of a playout value function, which replaces the missing value estimates in the non-memorized part of the state space with play-

```
function selectMove($position$, $depth$)
$nMoves$ := # available moves in $position$
$nsum$ := 0 {$nsum$ will contain # times the descendants of $position$ are considered}
for $i$ := 1 to $nMoves$ do
    Let $tte[i]$ be the TT entry matching the $i$th descendant of $position$
    if the entry $tte[i]$ is invalid then
        return random move in $position$
    end if
    $nsum := nsum + tte[i].n$ {$tte[i].n$ = # times the $i$th descendant is considered}
end for
$maxv := -\infty$;
for $i$ := 1 to $nMoves$ do
    if $tte[i].n = 0$ then
        $v := +\infty$ {Give high preference to an unvisited descendant}
    else
        $v := tte[i].value + \sqrt{2 * \ln(nsum)/tte[i].n}$
    end if
    if $v > maxv$ then
        $maxv := v$
        Let $m$ be the $i$th move
    end if
end for
return $m$
```

Figure 2. UCT Pseudo-Code

out values $V_{playout}(s)$ and $Q_{playout}(s; a)$. The focus on the non-memorizing part as well comes from the fact that this is a peculiarity of the MCTS framework, not represented in the RL one. Then, a TD back-up can be introduced in the usual MCTS case as shown in the pseudo-code 3.

```
procedure BackupTDerrors($episode$)
    $\delta_{\text{sum}} \leftarrow 0$                                   ▷ cumulative decayed TD error
    $V_{\text{next}} \leftarrow 0$
    for $i = $ Length($episode$) down to 1
        $(s, R) \leftarrow episode(i)$
        if $s$ is in $tree$
            $V_{\text{current}} \leftarrow tree(s).V$
        else                                                      ▷ assumed playout value
            $V_{\text{current}} \leftarrow V_{\text{playout}}(s)$
        $\delta \leftarrow R + \gamma V_{\text{next}} - V_{\text{current}}$                     ▷ single TD error
        $\delta_{\text{sum}} \leftarrow \lambda\gamma\delta_{\text{sum}} + \delta$                          ▷ decay and accumulate
        if $s$ is in $tree$                                             ▷ update value
            $tree(s).n \leftarrow tree(s).n + 1$
            $\alpha \leftarrow \frac{1}{tree(s).n}$                              ▷ example of MC-like step-size
            $tree(s).V \leftarrow tree(s).V + \alpha\delta_{sum}$
        $V_{\text{next}} \leftarrow V_{\text{current}}$                  ▷ remember the value from before the update
```

Figure 3. TD Backup Pseudo-Code

5 The other side of the coin: Best-Arm Identification and UGapE

5.1 Extension to MDPs: MDP-UGapE

6 Open Loop Planning: OLOP