Partially Observable Monte Carlo Planning

Dhruv Malik, Andy Palan

April 17th 2017

We wrote these notes based on what we learned from the 2010 paper *Monte Carlo Planning in Large POMDPs* by Silver et al., the 2006 paper *Bandit based Monte Carlo Planning* by Kocsis et al., the 2002 paper *A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes* by Kearns et al., and the 2002 paper *Finite-time Analysis of the Multiarmed Bandit Problem* by Auer et al. We hoped to take what we learned from each resource and write a condensed and easy to follow version.

Sampling for MDPs

Sampling can provide a far more efficient method to find near-optimal actions in MDPs, as opposed to using optimal but computationally intensive methods like Value Iteration. This is useful for MDPs with very large or infinite state spaces. Sampling thus approximately simulates the behavior of an MDP, instead of explicitly computing it. Kearns et al. define a generative model G for an MDP M to be a randomized algorithm, that on input of a state action pair (s, a), outputs next state s' and corresponding reward $R(s, a, s')$, where s' is sampled according to the transition probabilities given by $T(s, a, s')$. Based on this, they come up with the algorithm presented in Figure 1.

The idea is simply to maintain two different types of nodes in the search tree. The first type is for actual states s, the other is for Q states or state-action pairs (s, a). We explain as follows. From our starting state s_0, we call the **EstimateQ** method, which iterates over the list of actions available from s_0, and for each action a, computes the value of Q state (s_0, a). It does so by sampling C next states s' from the transition distribution $T(s_0, a)$. It then takes the values of these future states, and averages them to determine the value of the Q state. In turn, the values of the future states s' are calculated using **EstimateV**, because this method once again recursively calls **EstimateQ** to get values of each future Q state for an s' and takes the maximum over those. This algorithm thus simulates the Bellman Update Equation used during Value Iteration, by using sampling.

Function: **EstimateQ**(h, C, γ, G, s)
Input: depth h, width C, discount γ, generative model G, state s.
Output: A list $(\hat{Q}_h^*(s,a_1), \hat{Q}_h^*(s,a_2), \ldots, \hat{Q}_h^*(s,a_k))$, of estimates of the $Q^*(s,a_i)$.

1. If $h = 0$, return $(0, \ldots, 0)$.
2. For each $a \in A$, use G to generate C samples from the next-state distribution $P_{sa}(\cdot)$. Let S_a be a set containing these C next-states.
3. For each $a \in A$ and let our estimate of $Q^*(s,a)$ be

$$\hat{Q}_h^*(s,a) = R(s,a) + \gamma \frac{1}{C} \sum_{s' \in S_a} \textbf{EstimateV}(h-1, C, \gamma, G, s'). \tag{5}$$

4. Return $(\hat{Q}_h^*(s,a_1), \hat{Q}_h^*(s,a_2), \ldots, \hat{Q}_h^*(s,a_k))$.

Function: **EstimateV**(h, C, γ, G, s)
Input: depth h, width C, discount γ, generative model G, state s.
Output: A number $\hat{V}_h^*(s)$ that is an estimate of $V_h^*(s)$.

1. Let $(\hat{Q}_h^*(s,a_1), \hat{Q}_h^*(s,a_2), \ldots, \hat{Q}_h^*(s,a_k)) :=$ **EstimateQ**(h, C, γ, G, s).
2. Return $\max_{a \in \{a_1, \ldots, a_k\}} \{\hat{Q}_h^*(s,a)\}$.

Function: **Algorithm** $\mathcal{A}(\epsilon, \gamma, R_{max}, G, s_0)$
Input: tolerance ϵ, discount γ, max reward R_{max}, generative model G, state s_0.
Output: An action a.

1. Let the required horizon H and width C parameters be calculated as given as functions of ϵ, γ and R_{max} in Theorem1.
2. Let $(\hat{Q}_H^*(s,a_1), \hat{Q}_H^*(s,a_2), \ldots, \hat{Q}_H^*(s,a_k)) :=$ **EstimateQ**(H, C, γ, G, s_0).
3. Return $\arg\max_{a \in \{a_1, \ldots, a_k\}} \{\hat{Q}_H^*(s,a)\}$.

Figure 1: Sampling For MDPs by Kearns et al.

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an improvement to the above algorithm, and is based on the idea of rollouts (or playouts). Rollout based algorithms build the search tree by repeatedly sampling episodes from the initial state. This allows us to keep track of estimates of the action values at the sampled states encountered in earlier episodes. There is a node for each state s, and each node keeps track of value $Q(s,a)$ and visitation count $N(s,a)$ for each action a. There are four stages in MCTS algorithm:

1. Selection - Start at the root node, and recursively select child nodes according to a selection strategy, until a leaf node is reached.
2. Expansion - If the leaf node is a terminal node (or has reached the depth we want), then the episode is over. Otherwise, create a child node according to a strategy, and select it.
3. Simulation - Run a simulated rollout (using successive random actions) from the node you just selected until a result is achieved. Note that only this child node is added to the search tree (and nothing from the simulation we just ran).

```
1: function MonteCarloPlanning(state)
2: repeat
3:     search(state, 0)
4: until Timeout
5: return bestAction(state,0)

6: function search(state, depth)
7: if Terminal(state) then return 0
8: if Leaf(state, d) then return Evaluate(state)
9: action := selectAction(state, depth)
10: (nextstate, reward) := simulateAction(state, action)
11: q := reward + γ search(nextstate, depth + 1)
12: UpdateValue(state, action, q, depth)
13: return q
```

Figure 2: MCTS Pseudocode by Kocsis et al.

4. Backpropagation - Update the move sequence you just made with the appropriate values.

We provide the following pseudocode for MCTS from the paper by Kocsis et al. above. The key idea is that as we repeat episodes, we have access to the values of state-action pairs that have already been visited, as well as the number of times they have been visited. This allows us the flexibility to make more educated decisions about which actions to choose from a particular state, instead of say choosing them randomly.

UCT and UCB1

Naive MCTS may use a greedy strategy while within the tree. This means that from a particular state in the tree, it will always pick the action which returns the greatest value. However, it may be a better choice to encourage exploration instead of exploitation, since there may be an optimal action which at that current moment does not have a good estimate of it's true value. This dilemma of exploration versus exploitation shows up in its simplest form in the k-armed bandit problem. Solutions to this problem can be used to determine the appropriate action to pick from a state to maintain the correct balance of exploration and exploitation, as we can think of each action from a state to be an arm of the bandit.

UCB1 is an algorithm which chooses the best arm $a_{correct}$ out of k total arms to play according to the following criterion:

$$a_{correct} = \underset{a_i \in A}{\text{argmax}} \left\{ x_i + c\sqrt{\frac{ln(n)}{n_i}} \right\} \tag{1}$$

In the above equation, A is the set of all the k arms, x_i is the average reward returned by playing arm i so far, n is the number of total plays, and n_i is the number of times arm i has been played thus far. c is often set to $\sqrt{2}$, but can be adjusted appropriately. Increasing c increases exploration, decreasing s increases exploitation. We can adjust this equation to model which action to select when we are in state s while performing MCTS:

$$a_{correct} = \underset{a \in A}{\text{argmax}} \left\{ Q(s,a) + c\sqrt{\frac{ln(N(s))}{N(s,a)}} \right\} \tag{2}$$

In the above equation, A is the set of all actions available from state s, $Q(s,a)$ are the current values of state action pairs, $N(s)$ gives the number of times that state has been visited, and $N(s,a)$ gives the number of times action a has been performed from state s. Thus, we have a method to balance exploitation and exploration while performing MCTS, which greatly improves our complexity and prevents us from exploring useless parts of the search tree while still choosing actions which we believe have high value. This modification to the MCTS algorithm is known as UCT (UCB1 applied to trees).

Monte Carlo Planning In POMDPs

Partially Observable UCT (PO-UCT)

One can extend the UCT algorithm to partially observable environments by using a search tree of histories instead of physical states. For a history h, we have a node which stores $V(h)$ and $N(h)$. $V(h)$ is the current value of history h, estimated by the mean return of all simulations starting with h. $N(h)$ counts the number of times that history h has been visited. For this algorithm, one assumes that the belief state $B(s,h)$, which describes the probability of being in state s given history h, is known exactly. Each simulation starts with an initial state that is sampled from the belief distribution. As in UCT, while still within the scope of the tree, the UCB1 algorithm is used for action selection. Once we reach a leaf node, we perform a rollout, usually using successive random action selection until we have reached our horizon. After each simulation, there is exactly one node added to the tree, corresponding to the first new history encountered during that simulation.

Monte Carlo Belief State Updates

The belief state can be exactly updated based on histories, but in large state spaces, such extensive computation can become infeasible. To solve this, one can use a particle filter, which uses sampling to maintain an approximation of the belief state. At the start of the algorithm, K particles are sampled from the initial state distribution, and each particle corresponds to a particular state. The idea is that K is much smaller than the size of the actual state space, but large enough to provide a reasonable approximation. The belief state B is then approximated by $\hat{B}$, where $\hat{B}(s) = \frac{1}{K}\Sigma_{i=1}^{K}\delta_{sK_i}$, where K_i is the i^{th} particle (representing a state), and δ_{ij} is the kronecker delta function. After a real action a is executed and real observation o is observed, the particles are updated by Monte Carlo simulation as follows. A state s is sampled from the current approximate belief state by sampling one of the K particles. We use our generative model G for the POMDP to sample a successor state s' and new observation o' based on the input (s,a). If the sample observation equals the real observation, meaning $o = o'$, then we have a new particle s' and this is used in the representation of the new belief state. We repeat this until we have K total particles and can completely construct the new approximate belief state. Note that in the limit as $K \to \infty$, the approximate belief state equals the true belief state.

Algorithm 1 Partially Observable Monte-Carlo Planning

procedure SEARCH(h)
  repeat
    if $h = empty$ **then**
      $s \sim \mathcal{I}$
    else
      $s \sim B(h)$
    end if
    SIMULATE($s, h, 0$)
  until TIMEOUT()
  return $\underset{b}{\text{argmax}}\ V(hb)$
end procedure

procedure ROLLOUT($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  $a \sim \pi_{rollout}(h, \cdot)$
  $(s', o, r) \sim \mathcal{G}(s, a)$
  return r $+ \gamma$.ROLLOUT($s', hao, depth{+}1$)
end procedure

procedure SIMULATE($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  if $h \notin T$ **then**
    for all $a \in \mathcal{A}$ **do**
      $T(ha) \leftarrow (N_{init}(ha), V_{init}(ha), \emptyset)$
    end for
    return ROLLOUT($s, h, depth$)
  end if
  $a \leftarrow \underset{b}{\text{argmax}}\ V(hb) + c\sqrt{\frac{\log N(h)}{N(hb)}}$
  $(s', o, r) \sim \mathcal{G}(s, a)$
  $R \leftarrow r + \gamma$.SIMULATE($s', hao, depth + 1$)
  $B(h) \leftarrow B(h) \cup \{s\}$
  $N(h) \leftarrow N(h) + 1$
  $N(ha) \leftarrow N(ha) + 1$
  $V(ha) \leftarrow V(ha) + \frac{R - V(ha)}{N(ha)}$
  return R
end procedure

Figure 3: POMCP Pseudocode by Silver et al.

Partially Observable Monte Carlo Planning (POMCP)

POMCP combines both Monte Carlo Belief State Updates with the PO-UCT algorithm. In addition to $V(h)$ and $N(h)$, now each node also maintains $B(h)$, which uses particle filtering to describe the approximate belief state given history h. Each simulation begins with a particle sampled from $B(h)$ where h is the root history. This particle represents the state we think we are in, and we use this to sample new particles and observations from the generative model. This is necessary because without such a particle we cannot sample a future observation and future particle. For every new history we encounter, we update the belief set of that history by including the particle representing the new state we think we are in. This is the exact procedure that we are performing above. Thus, we are modeling the belief states of histories we encounter during simulations. Eventually, when we have gathered enough samples, we select the action at the root which maximizes expected utility and execute it. We then get a real time observation, corresponding to one of the children of the root. We select this child, this is our new history. We update it's belief using the procedure we described above in the section about Monte Carlo Belief State Updates. We can prune the rest of the search tree, since no other histories are now possible. The algorithm by Silver et al. is shown above.