CS234 Notes - Lecture 14
Model Based RL, Monte-Carlo Tree Search

Anchit Gupta, Emma Brunskill

June 14, 2018

1 Introduction

In this lecture we will learn about model based RL and simulation based tree search methods. So far we have seen methods which attempt to learn either a value function or a policy from experience. In contrast model based approaches first learn a model of the world from experience and then use this for planning and acting. Model-based approaches have been shown to have better sample efficiency and faster convergence in certain settings. We will also have a look at MCTS and its variants which can be used for planning given a model. MCTS was one of the main ideas behind the success of AlphaGo.

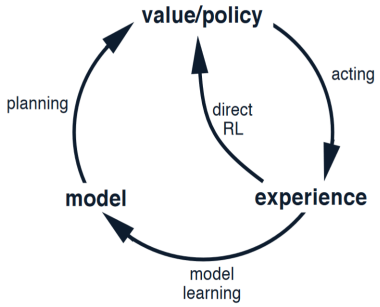


Figure 1: Relationships among learning, planning, and acting

2 Model Learning

By model we mean a representation of an MDP $< S, A, R, T, \gamma >$ parametrized by η. In the model learning regime we assume that the state and action space S, A are known and typically we also assume conditional independence between state transitions and rewards i.e

$$P[s_{t+1}, r_{t+1}|s_t, a_t] = P[s_{t+1}|s_t, a_t]P[r_{t+1}|s_t, a_t]$$

Hence learning a model consists of two main parts the reward function $R(.|s, a)$ and the transition distribution $P(.|s, a)$.

Given a set of real trajectories $\{S_t^k, A_t^k R_t^k, ..., S_T^k\}_{k=1}^K$ model learning can be posed as a supervised learning problem. Learning the reward function $R(s, a)$ is a regression problem whereas learning the transition function $P(s'|s, a)$ is a density estimation problem. First we pick a suitable family of parametrized models, these may include Table lookup models, linear expectation, linear gaussian,

gaussian process, deep belief network etc. Subsequently choose an appropriate loss function eg. mean squared error, KL divergence etc. to optimize the choice of parameters that minimize this loss.

3 Planning

Given a learned model of the environment planning can be accomplished by any of the methods we have studied so far like value based methods, policy search or tree search which we describe soon.

A contrasting approach to planning uses the model to only generate sample trajectories and methods like Q learning , Monte-Carlo control, Sarsa etc. for control. These sample based planning methods are often more data efficient.

The model we learn can be inaccurate and as a result the policy learned by planning for it would also be suboptimal i.e Model based RL is dependent on the quality of the learned model. Techniques based on the exploration/exploitation section can be used to explicitly reason about this uncertainty in the model while planning. Alternatively if we ascertain the model to be wrong in certain situations model free RL methods can be used as a fall back.

4 Simulation based search

Given access to a model of the world either an approximate learned model or an accurate simulation in the case of games like Go, these methods seek to identify the best action to take based on forward search or simulations. A search tree is built with the current state as the root and the other nodes generated using the model. Such methods can give big savings as we do not need to solve the whole MDP but just the sub MDP starting from the current state. In general once we have gathered this set of simulated experience $\{S_t^k, A_t^k R_t^k, ..., S_T^k\}_{k=1}^K$ we can apply model free methods for control like Monte-Carlo giving us an Monte-Carlo search algorithm or Sarsa giving us a TD search algorithm.

More concretely in a simple MC search algorithm given a model M and a simulation policy π, for each action $a \in A$ we simulate K episodes of the form $\{S_t^k, a, R_t^k, ..., S_T^k\}_{k=1}^K$ (following π after the first action onwards). The $Q(s_t, a)$ value is evaluated as the average reward of the above trajectories and we subsequently pick the action which maximizes this estimated $Q(s_t, a)$ value.

4.1 Monte Carlo Tree Search

This family of algorithms is based on two principles. 1) The true value of a state can be estimated using average returns of random simulations and 2) These values can be used to iteratively adjust the policy in a best first nature allowing us to focus on high value regions of the search space.

We progressively construct a partial search tree starting out with the current node set as the root. The tree consists of nodes corresponding to states s. Additionally each node stores statistics such as the total visitation count $N(s)$, a count for each pair $N(s, a)$ and the monte-carlo $Q(s, a)$ value estimates. A typical implementation builds this search tree until some preset computational budget is exhausted with the value estimates (particularly for the promising moves) becoming more and more accurate as the tree grows larger. Each iteration can be roughly divided into four phases.

1. Selection: Starting at the root node, we select child nodes recursively in the tree till a non-terminal leaf node is reached.
2. Expansion: The chosen leaf node is added to the search tree
3. Simulation: Simulations are run from this node to produce an estimate of the outcomes

4. Backprop: The values obtained in the simulations are back-propagated through the tree by following the path from the root to the chosen leaf in reverse and updating the statistics of the encountered nodes.

Variants of MCTS generally contain modifications to the two main policies involved -

- **Tree policy** to chose actions for nodes in the tree based on the stored statistics. Variants include greedy, UCB
- **Roll out policy** for simulations from leaf nodes in the tree. Variants include random simulation, default policy network in the case of AlphaGo

Algorithm 1 General MCTS algorithm

function MCTS(s_0)
  Create root node v_0 corresponding to s_o
  while within computational budget **do**
    $v_k \leftarrow TreePolicy(v_o)$
    $\Delta \leftarrow Simuation(v_k)$
    $Backprop(v_k, \Delta)$
  return $arg\max_a Q(s_o, a)$

These steps are summarized in 1 we start out from the current state and iteratively grow the search tree, using the tree policy at each iteration to chose a leaf node to simulate from. Then we backpropagate the result of the simulation and finally output the action which has the maximum value estimate form the root node.

A simple variant of this algorithm choses actions greedily amongst the tree nodes in the first stage as implemented in 2, and generates roll outs using a random policy in the simulation stage.

Algorithm 2 Greedy Tree policy

function TREEPOLICY(v)
  $v_{next} \leftarrow v$
  if $|Children(v_{next})| \neq 0$ **then**
    $a \leftarrow arg\max_{a \in A} Q(v, a)$
    $v_{next} \leftarrow nextState(v, a)$
    $v_{next} \leftarrow TreePolicy(v_{next})$
  return v_{next}

Various modifications of the above scheme exist to improve memory usage by adding a limited set of nodes to the tree, smart pruning and tree policies based on using more complicate statistics stored in the nodes.

A run through of this is visualized in 2. We start out from the root node and simulate a trajectory which gives us a reward of 1 in this case. We then use the tree policy which greedily selects the node to add to the tree and subsequently simulate an episode from it using the simulation(default) policy. This episode gives us a reward of 0 and we update the statistics in the tree nodes. This process is then repeated, to check your understanding you should verify in the below example that the statistics have been updated correctly and that the tree policy chooses the correct node to expand.

The main advantages of MCTS include

- Its tree structure makes it massively parallelisable.
- Dynamic state evaluation i.e solve MDP from current state onwards unlike DP.

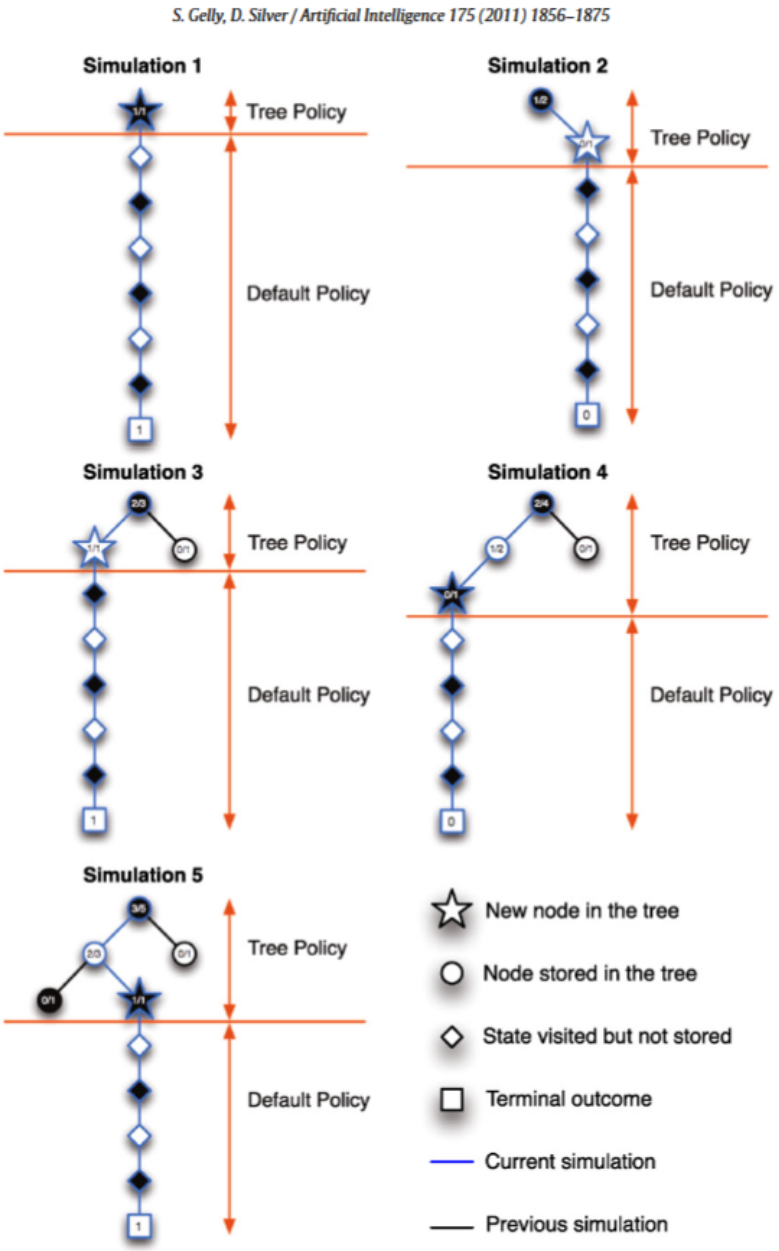


Figure 2: Demonstration of a simple MCTS. Each state has two possible actions (left/right) and each simulation has a reward of 1 or 0. At each iteration a new node (star) is added into the search tree. The value of each node in the search tree (circles and star) and the total number of visits is then updated

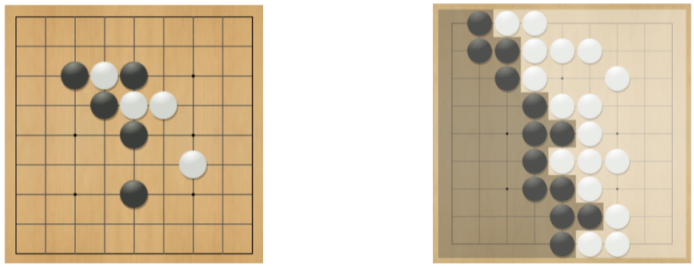


Figure 3: Go

- No need for domain specific engineering i.e it works for black box models and needs only samples
- Efficiently combines planning and sampling to break the curse of dimensionality in games like Go.

4.2 Upper Confidence Tree Search

Similar to the multi armed bandit case using a greedy policy as the tree policy can often be suboptimal, making us avoid actions after even one bad outcome even though there is significant uncertainty about its true value. As an example consider the rightmost node in 2 from which we do a single rollout, receive a 0 reward and never visit it again even though this reward might have just been due to bad luck. To fix this we can apply the principle of optimism under uncertainty to MCTS using the UCB algorithm. More specifically the tree policy instead of picking the action greedily picks the action which maximizes the upper confidence bound on the value of the action i.e $Q(s,a) + \sqrt{\frac{2 \log N(s)}{N(s,a)}}$

Refer 3 for pseudo-code of the tree policy used in UCT which can be plugged into the general MCTS algorithm described earlier. The $nextState(s,a)$ function uses the model of the MDP to sample a next state when picking action a from state s.

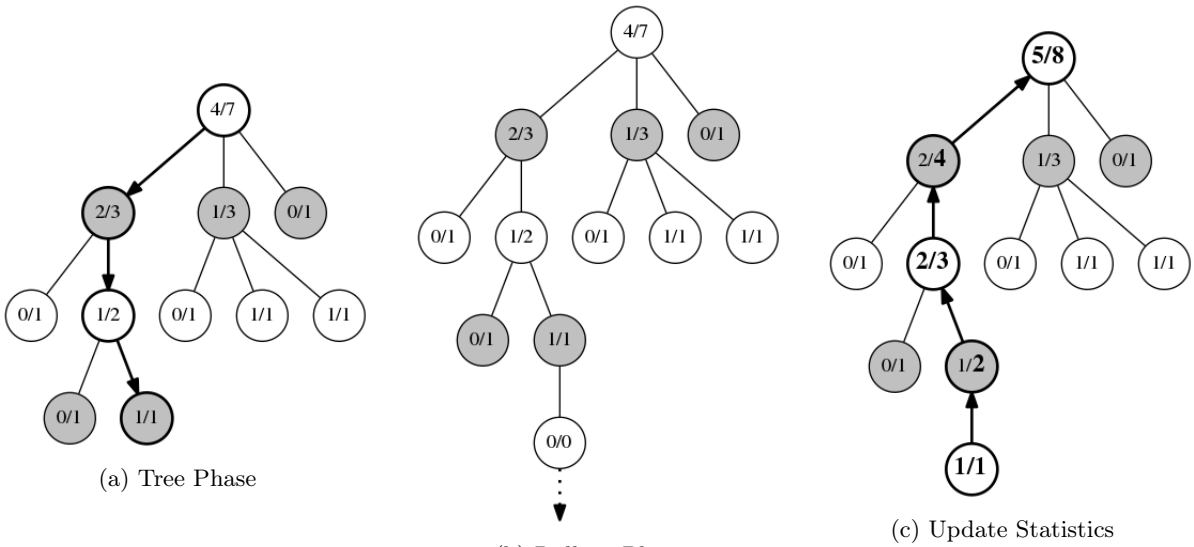
Algorithm 3 Upper Confidence Tree policy

function TREEPOLICY(v)
  $v_{next} \leftarrow v$
  if $|Children(v_{next})| \neq 0$ **then**
    $a \leftarrow arg\max_{a \in A} Q(v,a) + \sqrt{\frac{2 \log N(v)}{N(v,a)}}$
    $v_{next} \leftarrow nextState(v,a)$
    $v_{next} \leftarrow TreePolicy(v_{next})$
  return v_{next}

5 Case Study: Go

Go is the oldest continuously played board game in the world. Solving it had been a long standing challenge for AI and before AlphaGO traditional game tree search algorithms had failed to achieve professional human level performance on it. Go is a two player game(B/W), it is played on a 19x19 board (smaller variants exist). Black, White alternatingly place down stones on the board. The main goal of the game is to surround and capture territory. Additionally stones surrounded by the opponent are removed.

The simplest reward function gives a reward of +1 if Black wins and 0 if White wins on terminal



(a) Tree Phase

(b) Rollout Phase

(c) Update Statistics

Figure 4: UCT on Go, colored nodes represent the players and each node contains the probability of that player winning. (Image credits: [3])

states and 0 elsewhere. As a result the goal of the Black player is to maximize the reward and white tries to minimize it. Given a policy $\pi =< \pi_B, \pi_W >$ for both players the value function $V_\pi(s) = E_\pi[R_T|s] = P[\text{Black wins}|s]$ and the optimal value function is $V^*(s) = \max_{\pi_B} \min_{\pi_W} V_\pi(s)$.

5.1 MCTS for Go

Go being a two player game warrants some fairly natural extensions to the previosuly discussed MCTS algorithm. We now build a minimax tree with nodes alternating players across levels. The white nodes seek to minimize while the black ones maximize. We use UCB as described above at the black nodes and LCB (Lower Confidence Bound) i.e $\min_a Q(s,a) - \sqrt{\frac{2 \log N(s)}{N(s,a)}}$ at the white nodes as they seek to minimize the reward.

Consider the following state of the tree in 4a with the statistics (win/total games) recorded in the nodes and the colors representing the players. The first phase of the algorithm or the Tree policy would use these statistics in the nodes, treating each of them as an independent MAB instances and would sequentially choose actions using UCB(LCB) from the root onwards. These are highlighted with bold arrows (taking $c = \sqrt{2}$).

Once we reach a leaf node in this tree 4b we use our rollout policy to simulate a game. The outcome of this game is then back propagated through the tree 4c and the statistics updated.

This process is continued until desired and the best action to take from the root returned subsequently. For detailed pseudocode you can refer to [4] and [3] for a python implementation.

AlphaGo [1] used a deep policy network for the rollout phase, this allows the simulations to be much more realistic than just using random rollouts. Also in a complex game like Go it is not feasible to simulate till completion, early stopping is used along with a Value network to get the required win probabilities. Recently AlphaGo Zero [2] has been proposed which uses a single network to output both the policy and the value function and is trained purely using self play with no expert knowledge built in. This gets even more impressive performance than AlphaGo.

References

[1] Silver, D. et al. "Mastering the game of Go with deep neural networks and tree search." Nature 529, 484–489 (2016).

[2] Silver, D. et al. "Mastering the game of Go without human knowledge" Nature doi:10.1038/nature24270 (2017).

[3] Bradberry, J "Introduction to Monte Carlo Tree Search" https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/

[4] Gelly, S. et al. "Monte-Carlo tree search and rapid action value estimation in computer Go" Artificial Intelligence 175 (2011) 1856–1875