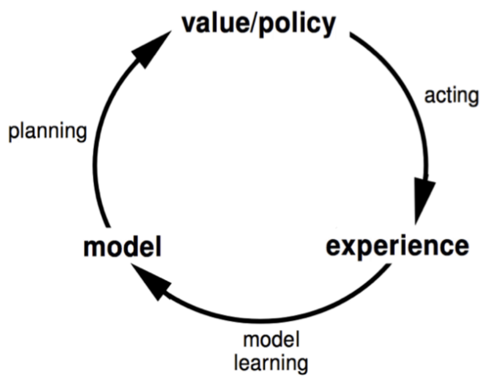
Reinforcement Learning **Lecture–8**

Planning and Learning

Anubhav Gupta

The methods discussed so far are all examples of model-free reinforcement learning methods, where we don't explicitly construct the model of the environment. Monte-Carlo and temporal-difference are examples of model-free methods. In these notes, we will look at model-based methods which first build the model of the environment and then use that model for planning and to perform more effectively in the environment. Dynamic programming methods are example of model-based methods since they require a model of the environment to make effective decisions.

Model-free methods primarily rely on **learning**. That is they learn policy/value-function directly from experience. On the other hand, model-based methods rely on **planning**. They first learn the model of the environment from experience, and then use this model to plan the optimal policy/value-function.



1 Model-Based Learning

Model-based reinforcement learning can efficiently use supervised learning methods to learn the model. Training data can be constructed from agent's experience with current state and action as features/attributes and next state or reward as the labels.

Sometimes a model might be a more useful representation of the information about the environment. Consider for example the game of chess, where the number of possible board positions is around 10^{45}. In this case, learning the policy or value-function directly might not be the best thing to do, since one board position can have value function of +1 (winning) while the next one can have the value function of negative. On the other hand learning a model for chess is quite straightforward. A model for chess is just learning the rules of the game. Then these rules can be used to learn the optimal policy. Model-based methods can also reason about model uncertainty, i.e. which parts of the environment are understood by the model.

1.1 Learning the Model

A model can be thought of as anything that helps agent understand the dynamics of its environment, i.e. how the environment will respond to its actions. More formally, given a state and an action a model predicts the resultant next state and next (immediate) reward. Such deterministic models are called sample models. Stochastic models might produce probability distribution over possible next states and rewards. Such models are called **distribution models**. Dynamic programming methods assume a distribution model of the environment.

Let us formally define a model. A model $\mathcal{M}$ is a representation of an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}$ parameterized by η. We assume that the state space $\mathcal{S}$ and action space $\mathcal{A}$ are known to the agent. So, a model $\mathcal{M} = \langle \mathcal{P}_\eta, \mathcal{R}_\eta$ predicts the state transitions and rewards:

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1}|S_t, A_t)$$
$$R_{t+1} = \mathcal{R}_\eta(R_{t+1}|S_t, A_t)$$

State transitions and rewards are assumed to be conditionally independent given current state and action.

$$\mathbb{P}\left[S_{t+1}, R_{t+1}|S_t, A_t\right] = \mathbb{P}\left[S_{t+1}|S_t, A_t\right]\mathbb{P}\left[R_{t+1}|S_t, A_t\right]$$

The goal is then, to estimate the model $\mathcal{M}_\eta$ from experience $S_1, A_1, R_2, \cdots, S_T$. This is a supervised learning problem. To learn the reward function current state and action are features and next reward is the label we want to predict. i.e. learning the reward function $((S_t, A_t), R_{t+1})$ is a regression problem. Similarly learning the next state $((S_t, A_t), S_{t+1})$ is a density estimation problem. Models such as table lookup model, linear Gaussian model, Gaussian process model or deep belief networks can be used to estimate the parameter η of the model.

Table lookup model explicitly models $\hat{\mathcal{P}}, \hat{\mathcal{R}}$ by counting the occurrences of each next state and reward from current state and action pairs. That is they estimate the parameters using maximum likelihood estimation.

$$\hat{\mathcal{P}}^a_{s,s'} = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbb{1}(S_t, A_t, S_{t+1} = s, a, s')$$
$$\hat{\mathcal{R}}^a_{s} = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbb{1}(S_t, A_t = s, a) R_t$$

1.2 Planning

Once the model of the environment is known/learnt, planning is used to find an (optimal) policy for interacting with the environment. Methods such as value iteration, policy iteration, tree search can be used to plan a policy. Sample-based planning is simple, yet efficient approach to planning. In sample-based planning the model is used only to generate samples

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1}|S_t, A_t)$$
$$R_{t+1} = \mathcal{R}_\eta(R_{t+1}|S_t, A_t)$$

and then model-free methods such as Monte-Carlo, temporal-difference or Q-learning are applied to estimate the policy.

On thing to keep in mind when working with model-based learning methods is that their performance is limited to the optimal policy for the approximated MDP. i.e. if the model of the environment is inaccurate, planning process will only compute a sub-optimal policy.

2 Integrating Learning and Planning

Both learning and planning methods estimate value functions by backing-up update operations. The difference is that learning uses real experience generated by the environment and planning uses simulated experience generated by the (approximate) model of the environment. i.e. for learning

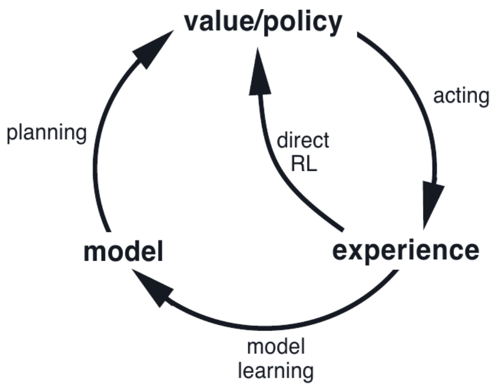
$$S' \sim \mathcal{P}^a_{s,s'}$$
$$R = \mathcal{R}^a_s$$

and for planning

$$S' \sim \mathcal{P}_\eta(S'|S, A)$$
$$R = \mathcal{R}_\eta(R|S, A)$$

2.1 Dyna

Dyna provides a mechanism to combine learning, acting and planning into the same process, such that the agent keeps updating its model of the environment while interacting with it at the same time. Then the agent uses both the real experience and simulated experience to learn the value function/policy. The real experience is used to (1) improve the policy (model learning) and (2) directly improve the value function and policy (direct reinforcement learning). The full pseudo-code of Dyna-Q algorithm is given in algorithm-1.



The direct RL method used in Dyna-Q is one-step tabular Q-learning. The model learning assumes that the environment is deterministic and stores for each

Algorithm 1: Tabular Dyna-Q Algorithm

```
Algorithm Parameters: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialization: Initialize Q(s,a) and Model(s,a) for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
for each time (forever) do
    $S \leftarrow$ current (non-terminal) state
    $A \leftarrow \epsilon$-greedy$(S, Q)$
    Take action $A$, observe $R, S'$
    $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)
    for $n$ times do
        $S \leftarrow$ random previously observed state
        $A \leftarrow$ random action previously taken in $S$
        $R, S' \leftarrow Model(S, A)$
        $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
    end
end
```

state the last transition and reward the agent actually experienced. Then during planning, the agent randomly samples only from state-action pairs that it has previously experienced and updates the value functions and policy. Both learning and planning are deeply integrated in Dyna in the sense that they both share the same machinery. The only difference is in the source of experience for learning and planning.

The agent is always interacting with the environment and gaining real experience, and yet always planning in the background. The amount of planning depends on the amount of simulated experience used by agent at every step. As new information is gained the model is updated to better match reality.

2.2 Dyna-Q+

Dyna-Q algorithm discussed above has one drawback that, after one point it stops exploring preventing it from finding changes in environment. Ideally, we would like the agent to keep exploring a little bit in order to improve its model of the environment.

Dyna-Q+ algorithm solves this problem using a simple heuristic. For each visited state-action pair, it keeps track of how much time has passed since this state-action pair was tried in real interaction with the environment, and gives "bonus reward" on simulated experience involving these actions. The intuition is that if some state-action pair has not been observed for a long time, it's dynamics might have changed. This kind of bonus reward encourages the agent to keep exploring and find changes in the environment. This exploration comes at some computational cost, but often it's worth the effort.

3 Simulation-Based Search

Let us now consider the planning problem in detail. We will see how to generate samples from simulated experiences and use that for effective planning. Most of the search algorithms are based on the idea of forward search. **Forward search** algorithms, unlike Dyna, don't search the entire state space. The goal is to select best action from *current* state. i.e. the focus is on the short term future. Forward search algorithms build a search tree with current state at the root, then do lookahead using the model of the MDP to identify the best next action. These algorithms don't solve the whole MDP, just the sub-MDP starting from *now*.

Simulation-based search is a forward search paradigm that uses sample-based planning. Just like forward search the agent starts from current state S_t, then it simulates the episodes of experience starting from S_t. Then use model-free learning to these simulated episodes. Different model-free models result in different search algorithms. If Monte-Carlo is used for control, the method is called Monte-Carlo search. If Sarsa is used for control, resulting algorithm is called TD search.

3.1 Simple Monte-Carlo Search

Given a model $\mathcal{M}_\eta$ and a simulation policy π, this method simulates K episodes for each possible action a from current state S_t. i.e.

$$\{S_t, a, R^k_{t+1}, A^k_{t+1}, \cdots, S^k_T\}_{k=1}^K \sim \mathcal{M}_\eta, \pi$$

Then actions are evaluated using Monte-Carlo evaluation by their mean return.

$$Q(S_t, a) = \frac{1}{K}\sum_{k=1}^{K} G_t$$

If enough samples are generated from the model $\mathcal{M}_\eta$, this quantity converges to true value function for simulated policy π. After evaluating each action from S_t, the action with maximum value is taken.

$$A_t = \operatorname*{argmax}_{a\in\mathcal{A}} Q(S_t, a)$$

3.2 Monte-Carlo Tree Search

Simple MC search method require a simulated policy along with the model. Monte-Carlo tree search, on the other hand, also improves the simulation policy over time.

Given a model of the environment $\mathcal{M}_\eta$ (exact or learned), Monte-Carlo tree search simulates K episodes from current state S_t using current simulation policy π.

$$\{S_t, A^k_t, R^k_{t+1}, A^k_{t+1}, \cdots, S^k_T\}_{k=1}^K \sim \mathcal{M}_\eta, \pi$$

MC tree search then builds a search tree containing visited states and actions and evaluates them by mean return of episodes from these state-action pairs.

$$Q(s, a) = \frac{1}{N(s, a)}\sum_{k=1}^{K}\sum_{u=t}^{T} \mathbb{1}(S_u, A_u = s, a)G_u$$

Once the search is finished, the action with highest value from current state is chosen.

$$A_t = \operatorname*{argmax}_{a\in\mathcal{A}} Q(S_t, a)$$

One big difference between this and simple Monte-Carlo search is that the search tree constructed in MC tree search method is used to improve the simulation policy as well. This is done by using policy improvement methods. For states present in search tree, simulation policy selects the action that maximizes $Q(s, a)$. For all other states a default policy is used (for example pick random action). So Monte-Carlo tree can be summarized in following two steps:

1. Evaluate states $Q(s, a)$ by Monte-Carlo evaluation
2. Improve tree policy, e.g. by $\epsilon - greedy(Q)$

MC tree search is same as Monte-Carlo control applied to simulated experience. Thus using similar argument, it can be shown that this algorithm converges to optimal search tree. i.e. $Q(s, a) \rightarrow q_*(s, a)$

Several advantages of Monte-Carlo tree search include

- MC tree search is like highly selective best-first search. Every episode (simulation) selects the best action(s) starting from root of the tree.
- It evaluates states dynamically. It focuses more on current states rather than wasting resources on all the states that might never get visited by the agent.
- Uses sampling to break the curse of dimensionality
- Its computationally efficient and parallelisable.

3.3 Temporal-Difference Search

Temporal-difference search is another simulation-based search method, based on bootstrapping. The idea is same as Monte-Carlo tree search, except it applies Sarsa instead of MC control to sub-MDP from now.

Similar to MC tree search TD search also simulates episodes from current state S_t, then estimate action-value function $Q(s, a)$ to evaluate actions. The algorithm used to estimate the action-value function is Sarsa.

$$\Delta Q(S, A) = \alpha(R + \gamma Q(S', A') - Q(S, A))$$

Instead of table-lookup TD, a function approximator can also be used to approximate action values Q. Then, the policy is improved by selecting the best action at this state ($\epsilon - greedy(Q)$).

3.4 Dyna-2

The search methods discusses above, all are based on simulation-search, i.e. they only use the simulated experience to plan. Dyna-2 provides a general architecture to combine simulated experience with real experience to improve the planning process.

Dyna-2 agents stores two sets of feature weights - a long-term memory and a short-term (working) memory. Long-term memory acts as a general domain understanding of the agent, and is only updated from real experience using TD learning. Working memory, on the other hand, represents agents knowledge specific to current situation and is updated from simulated experience using search tree methods such as TD search. Overall value function is represented by the sum of long and short-term memories.

References

[1] Reinforcement Learning: An Introduction (2nd Edition), by Richard S. Sutton, Andrew G. Barto.

[2] David Silver's course "Introduction to Reinforcement Learning" available at
`https://www.davidsilver.uk/teaching/`