Tabular Methods

Friday, June 19, 2020 7:57 PM

Model based methods rely on planning, while model free methods rely on learning
Planning uses simulated experience (experience generated from a model) while learning uses real experience (experience generated by the environment)
Model:
    Anything that an agent can use to predict how its environment will respond to its actions
Distribution Model:
    A model that produces a description of all possibilities and their probabilities
Sample Model:
    A model that produces one possible outcome, sampled according to the probability distribution of outcomes
Planning:
    Any computational process that takes a model as input and produces or improves a policy for interacting with the environment
State-Space Planning:
    A search through the state space for an optimal policy or path to a goal
        Note that this being a search through state space means that value functions are computed over states
Plan-Space Planning:
    A search through the space of plans
        Note that here value functions (if they are used) are computed over the space of plans
General Structure of State-Space Planning Methods:
1. They involve computing value functions as a key intermediate toward improving the policy
2. They compute value functions by updates or backup operations applied to simulated experience
        This structure is also present in the learning methods presented

Random-Sample One-Step Tabular Q-Planning:
1. Select a state and action at random
2. Send this state and action to a sample model and get a sample reward and next state
3. Apply one-step tabular Q-Learning

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_{a} Q(S',a) - Q(S,A)\right]$$

    Note that this converges under the same conditions as one-step tabular Q-Learning (each state action pair must be selected an infinite number of times as the number of trials goes to infinity and α must decrease appropriately over time)

Model Learning:
    Using real experience to improve the model (make it more accurately represent the real environment)
        This is sometimes called indirect reinforcement learning since improving the model will lead to more accurate value functions and policies
Direct Reinforcement Learning (Direct RL):
    Using real experience to improve the value function and policy
Comparison of Direct and Indirect Reinforcement Learning:
    Indirect RL methods often make better use of limited experience (achieve a better policy with fewer environmental interactions)
    Direct methods are simpler and not affected by biases in the design of the model
    Some people think that indirect models are always superior
    Some people think that direct methods are responsible for most human and animal learning
        A related debate concerns the relative importance of cognition and trial-and-error learning and deliberate planning vs reactive decision making

Dyna agents combine planning, acting, model-learning, and direct RL

Essentially they use both real and simulated experience (from an internal model) to update their policy/value functions

Dyna-Q:

n represents the number of planning steps taken per real step (the bottom loop is the planning phase)

1. Take an action determined by an ε-greedy method and record the reward and next state
2. Apply one-step tabular Q-learning
   $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_{a} Q(S',a) - Q(S,A)\right]$
3. Assign the observed reward and state to be the result of taking the action taken in the previous state in the model
   $Model(S,A) \leftarrow R, S'$
4. Repeat this loop n times:
   a. Randomly select a previously observed state and action taken in that state
   b. Use the internal model to simulate taking this action in this state
   c. Apply one-step tabular Q-learning with the result of the simulated experience

Note that if the model is wrong, planning will only lead to a suboptimal policy

If the model is wrong in such a way that something the model thinks has high reward actually has low reward, this will be discovered quickly but if something the model thinks has low reward actually has high reward this will be slow to be discovered, especially if many exploratory actions in a row are required to discover the error in the model

For planning methods, exploration refers to actions that improve the current model while exploitation refers to acting optimally according to the current model

Heuristics are often used to help balance exploration and exploitation

Dyna-Q+:

This is the Dyna-Q algorithm with the modification that if a state-action pairs has a modelled reward of r and the transition hasn't been tried in τ time steps, then planning is done as if that transition produced a reward of $r + \kappa\sqrt{\tau}$ for some small κ

Note that this is using a heuristic

Backward Focusing:

The idea of doing model updates by working backward from states that have changed in value (updating the actions that lead to those states then the states that those actions can be performed from, and so on)

Prioritized Sweeping:

The idea of keeping a queue of state-action pairs whose value would change nontrivially if updated, ordered by the size of the change

Note that as updates are computed and applied for the pair at the top of the queue, the effect of the applied change must be computed for the rest of the states and the queue must be updated

Prioritized Sweeping in a Deterministic Environment:

1. Take an action in accordance with the current policy and observe the resultant reward and new state
2. Assign the observed reward and state to be the result of taking the action taken in the previous state in the model
   $Model(S,A) \leftarrow R, S'$
3. Compute the magnitude of the update resulting from that action taken
   $P \leftarrow \left|R + \gamma \max_{a} Q(S',a) - Q(S,A)\right|$
4. If the magnitude of the update is greater than some threshold θ, then insert the state-action pair taken into the queue ordered by P
5. Repeat this loop n times:
   a. Compute the reward and next state for state-action pair at the top of the queue according to the current model

b. Apply one-step tabular Q-learning to this state-action pair
$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_{a} Q(S',a) - Q(S,A)\right]$$
c. Repeat this loop for all states and actions that lead to the updated state:
   i. Get the predicted reward from the model for state and action that lead to the updated state
   ii. Compute the magnitude of the update resulting from taking the action
   $$P \leftarrow \left|R + \gamma \max_{a} Q(S',a) - Q\left(\bar{S},\bar{A}\right)\right|$$
   Where $\bar{S}$ is the state that leads to the updated state and $\bar{A}$ is the action taken from that state that leads to the updated state
   iii. If the magnitude of this update is greater than some threshold θ then insert the state-action pair into the queue according to the magnitude of the update

To extend the above algorithm to stochastic environments, use expected updates

Forward Focusing:

The idea of doing model updates based on states according to how easily they can be reached from states that are visited frequently under the current policy

Comparison of Expected Updates and Sample Updates:

Expected updates are more accurate then sample updates but require most computational resources, which is often the limiting factor in problems

An expected update requires roughly a factor of b more computation than a sample update, where b is the branching factor (number of possible successor states with nonzero probabilities of being reached)

Note than an expected update is better than b sample updates due to the lack of sampling error

Sample updates are generally more computationally efficient than expected updates, since the majority of the benefit of an expected update can be achieved with a few sample updates

This is especially true on problems with large branching factors and a large state space

This is analogous to batched training for neural networks

Trajectory Sampling:

Simulating explicit individual trajectories and performing updates at the states or state-action pairs encountered

Irrelevant States:

States that cannot be reached from any start state under any optimal policy

Relevant States:

States that can be reached from some start state under an optimal policy

Optimal Partial Policy:

A policy that is optimal for relevant states but may be suboptimal or even undefined for irrelevant states

Real-Time Dynamic Programming (RTDP):

An algorithm that updates the values of states visited in actual or simulated trajectories by means of expected tabular value-iteration updates

RTDP converges to an optimal partial policy for Stochastic Optimal Path Problems (problems that meet the following conditions):

1. The initial value of every state is 0
2. There exists at least one policy that guarantees that a goal state will be reached with probability 1 from any start state
3. All rewards for transitions from non-goal states are strictly negative
4. All initial values are equal to or greater than their optimal values (note that these can be 0)

Background Planning:

Planning that improves the function approximation parameters needed to select actions from many states

Decision-Time Planning:

Planning that focuses on the current state by looking one or more states ahead and evaluating action choices leading to many different predicted state and reward trajectories

Note that background planning will lead to faster decisions than decision-time planning

Heuristic Search:

Classical decision-time state-space planning methods

In these methods, for each state encountered a large tree of possible continuations is considered with the approximate value function applied to leaf nodes then backed up towards the current state (the root). The best of these nodes is then chosen as the action and all backup values are discarded

In the conventional versions of these old methods, no effort is made to update the value functions, as it was usually designed by people

These can be viewed as an extension of greedy policies beyond a single step

If the search is of sufficient depth k such that γ^k is very small then the actions taken will be very near optimal

Rollout Algorithms:

Decision-time planning algorithms based on MC control applied to simulated trajectories beginning at the current state

That is to say they simulated a bunch of trajectories from the current state and estimate action values by averaging the returns of the simulated trajectories then take the action with the highest estimated value

The term rollout comes from playing out (i.e. "rolling out") the interaction between agent and environment

Rollout Policy:

The policy used in a rollout algorithm for which the rollout algorithm generates MC action-value estimates

The goal of rollout algorithms is to improve the rollout algorithm

Note that decision time constraints can hinder a rollout algorithm as they need to simulate enough trajectories to get a good estimate of the action values (although these can be run in parallel)

Monte Carlo Tree Search (MCTS):

A rollout algorithm enhanced by the addition of a means of accumulating values estimates from MC simulations in order to successively direct simulations towards more highly-rewarding trajectories

It does this by extending from the initial portions of trajectories that have received high rewards in earlier simulations using a tree policy

Steps:

1. Selection: A tree policy starts at the root node (current state) and traverses the tree to select a leaf node
2. Expansion: The tree is expanded from the selected leaf node by adding one or more child nodes reached by unexplored actions
3. Simulation: From the selected node or one of the child nodes, simulation is run of a complete episode with actions selected by the rollout policy

    Note that this results in a MC trial with actions selected first by the tree policy then beyond the tree by the rollout policy
4. Backup: The return generated by the simulated episode is backed up to update the action values attached to the edges of the tree traversed by the tree policy (the leaf node selected earlier)