8. Planning and Learning with Tabular Methods

Notes

- We look at *model-based* and *model-free* RL methods, where model-based methods rely on *planning*, whilst model-free methods rely on *learning*. This chapter aims to intermix and unify these two types of methods. Note that examples of model-based methods are DP and heuristic search, and examples of model-free methods are MC and TD methods. Both types rely on computation of value functions, and looking ahead to future events to update the value functions.
- Models and Planning:
    - A model is use to *simulate* the environment and produce a *simulated experience*.. Models can be separated into two categories - *distribution models* and *sample models*. Distribution models produce probabilities for all possibilities, while sample models produce just one of the possibilities, sampled according to the probabilities.
    - DP methods, which uses $p(s', r|s, a)$, are considered to use distribution models. MC methods typically use sample models. In general, sample models are much easier to obtain than distribution models, but is less powerful.
    - *Planning* is the idea of a process that takes a model as input and produces or improves a policy based off the interaction with the modelled environment. There are two distinct approaches - *state-space planning* and *plan-space planning*.
    - State-space planning encompasses the approach that we have explored so far, where the state space is searched through for an optimal policy by computing the value functions of states and/or action.
    - Plan-space planning involves a search through plan space, where value functions are defined over the space of plans. Plans-space planning

includes evolutionary methods, and they are generally difficult to apply efficiently in stochastic sequential decision problems, so will not be explored.

- There are two basic ideas to state-space planning: 1) it involves computing value functions as a key intermediate step to improving the policy, and 2) it computes value function by updates or backup operations applied to simulated experience. This can be given as:

  $$\text{model} \rightarrow \text{simulated experience} \rightarrow^{backups} \rightarrow \text{values} \rightarrow \text{policy}$$

  DP methods fit this structure very well. Other state-space planning methods fit this structure as well, but with modifications to the updates they do and their order, as well as how long the backed-up information is retained.
- Planning vs Learning (Model-based vs Model-free):
  - Key difference between planning and learning is that planning uses simulated experience generated by a model whilst learning methods use real experience generated by the environment. This leads to differences in how performance is assessed and how flexibly experience can be generated.
  - The common structure in both means that many ideas/algorithms can be transferred between the two. For example, learning methods can take in simulated experience rather than real experience.
- There are benefits to planning in small incremental steps, where it allows planning to be interrupted or redirected at any time with little wasted computation.

- Dyna: Integrated Planning, Acting and Learning
  - Dyna-Q is a simple online planning agent. Online planning refers to the idea of planning occurring at the same time as when the agent is interacting with the environment, and new information gained from the interaction can change the model and affect planning.
  - There are two roles to real experience: 1) *model-learning* or *indirect RL*, which is to improve the model to more accurately match the real

environment, and 2) *direct RL*, which is to directly improve the value function and policy.

    - Indirect RL methods make fuller use of a limited amount of experience, achieving a better policy with fewer environmental interactions. Direct RL methods are simpler and not affected by biases of the model and its design.
    - Dyna-Q involves a continuous process of planning, acting, model-learning and direct RL. Here, the planning method is the random-sample one-step tabular Q-planning method, where it samples from state-action pairs that have been experienced. The direct RL is on-step Q-learning.The model-learning is table-based and assumes that the environment is deterministic, where it will only return the last-observed next state and reward as its prediction. A good diagram of the process can be found on page 162 and 163.
        - *Search control* refers to the process that selects the starting states and actions for the simulated experiences generated by the model.
        - Typically, the same RL method is used for learning and planning in Dyna-Q. The method uses the final common path for both learning and planning.
        - Implementation can be found in age 164.
    - The example given shows that the larger the number of planning steps n per real steps, the more rapid the performance improves. Without planning, the agent is only able to add one additional step to the policy after one episode, since the reward is only given on the final step, hence only the value of the last non-terminal state is update. With planning, only one step is learned in the first episode, but an extensive policy has already been established by the planning process.
- When the Model is Wrong
    - Models can be incorrect because 1) the environment is stochastic, 2) a limited number of samples have been observed, 3) the model was learning using function approximation that has generalised imperfectly, or 4) the environment has changed and its new behaviour has not been observed.

    - A suboptimal policy is likely to be computed if the model is incorrect. However, this often leads to the discovery and correction of the modelling error.
    - For a Dyna-Q algorithm, problems can occur when the environment changes to become *better* than it was before, where there could be a quicker path through a maze. In such cases, the modelling error may not be detected for a long time, if ever. This brings about the conflict of exploration and exploitation, where we want the agent to explore, but not to have its performance significantly degraded.
    - Dyna-Q+ keeps track for each state-action pair how many time steps have elapsed since the pair was last tried in a real interaction with the environment. The more time that has elapsed, the higher chance that the dynamics of the state-action pair has changed, thus making the model inaccurate.
        - To encourage behaviour that test these state-action pair with a long elapsed time, a "bonus reward" is given on simulated experiences involving these actions. In particular, for a transition with modelled reward r with elapsed time of τ, the planning updates are done such that such transitions produce a reward of $r + \kappa\sqrt{\tau}$, for small κ.
        - This additional reward encourages the agent to test out these state-action pairs in the real environment, and even to find long sequences of cations. These kinds of computational curiosity is well worth the extra exploration.
- Prioritised Sweeping
    - Planning can be much more efficient if simulated transitions and updates are focused on particular state-action pairs. In this example, we want to work backwards from those state-action pairs who value has changed. In a scenario where the estimates are initially correct, a changed value would imply that other values need to be changed, but the only meaningful change would be done to those that lead directly to the changed state-action pair. This creates a knock on effect, where predecessor states may have to change as well. This idea is called *backward focusing* of planning computations.

  - The idea behind *prioritised sweeping* is to rank the urgency of the update by the magnitude of the change in following states, and for a stochastic environment, the variations in estimated transition probabilities. A queue is maintained of each state-action pair whose estimated value would change non-trivially if updated, prioritised by the size of the change. Implementation can be found in page 170.
  - In a stochastic environment, it is natural to update with an expected update rather than a sample update. This is one of the downsides of prioritised sweeping, as low probability transitions makes computation inefficient. Later sections show that sample updates can be better since it breaks up the computation into smaller pieces.
  - Other than backward focusing, *forward focusing* is the idea of focusing on states according to how easily they can be reached from the states that are visited frequently under the current policy.
- Expected vs. Sample Updates
  - Expected updates consider all possible events that might happen. Sample updates consider a single sample of what might happen.
  - Expected updates are not affected by sampling error, but they require more computation, where computation is often a limited resource in planning. However, it is important to remember that the extend to which the difference is significant is the extent to which the environment is stochastic.
  - Recall the equations for the two updates:

$$
\begin{aligned}
\text{Expected:}\quad & Q(s,a) \leftarrow \sum_{s',r} \hat{p}(s',r|s,a)\Big[r + \gamma \max_{a'} Q(s',a')\Big] \\
\text{Sample:}\quad & Q(s,a) \leftarrow Q(s,a)\alpha\Big[R + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big]
\end{aligned}
$$

  - For a particular starting state-action pair, s, a, let b be the branching factor, which is the number of possible next states s'. In a situation where b is large and that there is insufficient time to complete an expected update, sample updates are often preferred. To be more specific, sample updates can result in a dramatic decrease in error within the same amount of time for a single expected update, especially when b is large.

    - Another advantage of sample updates is given successor states are also estimates that are updated, the values backed up from the successor states will also be more accurate in a shorter period of time.

- Trajectory Sampling
    - There are two ways of distributing updates: the classical approach of sweeping through the entire state space (exhaustive sweeps), and sampling from the state space according to some distribution.
    - One can sample from the on-policy distribution, which means following the distribution observed when following the current policy. Note that the sample state transitions and rewards are given by the model. This is called *trajectory sampling*.
    - The advantage of using the on-policy distribution is that it could ignore the vast, uninteresting parts of the space. However, this can also be an disadvantage since it only updates the same areas repetitively. It can be observed that on-policy distribution is most advantageous initially, but the advantage decreases as time goes on. This shows that this distribution helps by focussing on states that are near descendants of the start state. This effect lasts longer on environments with large number of states and a small branching factor.
- Real-time Dynamic Programming (RTDP):
    - RTDP is an on-policy trajectory-sampling version of the value-iteration algorithm of DP. RTDP is an example of an *asynchronous DP algorithm* where the update order is dictated by the order states are visited in real or simulated trajectories.
    - If trajectories can start only from designated set of start states, then on-policy trajectory sampling allows the algorithm to completely skip *irrelevant* states, which are states that cannot be reached by the given policy from the start states. Here, we need an *optimal partial policy*, which only finds a policy optimal to the the relevant states.

    - Finding such an *optimal partial policy* would still require all state-action pairs be visited an infinite number of times, in order to guarantee convergence to optimal policies. This can be done via exploring starts.
    - However, an interesting note is that RTDP can still converge to an optimal partial policy for certain types of problems which satisfy certain conditions, without visiting every state infinitely many times. One of the tasks that can achieve this is an undiscounted episodic tasks for MDPs with absorbing goal states that generate zero reward.
        - Here, the RTDP selects a greedy action and applies the value iteration update to the current state, as well as an arbitrary collection of others states (eg. states visited in a limited horizon look-ahead search).
        - There are certain conditions that need to be fulfilled: 1) the initial value of all goal states is zero, 2) there exist at least one policy that guarantees that a goal state will be reached from any start state, 3) all rewards for transitions from non-goal states are strictly negative, 4) the initial values are greater or equal to their optimal values.
        - Tasks that fulfil these conditions are *stochastic optimal path problems*, which are more of cost minimisation problems, rather than reward maximisation. An example is a minimum time-control task.
    - RTDP methods tend to be much more efficient that conventional DP methods. Another advantage is that the policy used by the agent to generate trajectories approaches an optimal policy as the value function approaches the optimal value function v_*.
- Planning at Decision Time:
    - *Background planning* is what we have been dealing with up till now. Here, planning is used to improve a policy and/or value function on the basis of simulated experience from a model. Before selecting an action, planning has already played a part in improving the table entries of the value functions, or improve the function approximation parameters (discussed later on).
    - *Decision-time planning* is the idea of using planning after encountering a new state, and beginning and completing the planning process to produce

a selection of a single action. Here, each planning process can be seen to be independent of each other. This is often used when only state values are available, and an action is selected by comparing the values of model-predicted next states for each action. Such a method can look much deeper ahead into different trajectories.

    - Background planning is the use of simulated experience to improve a policy or value function, whilst decision-time planning is the use of simulated experience to select an action for the current state.
    - Decision-time planning is more suited for problems where fast responses are not required. In problems with large state spaces, a state is unlikely to be revisited until much later on, so decision-time planning does make sense. Often, a mix of both is good, where the results of planning are stored, but decision-time planning is used.
- Heuristic Search:
    - *Heuristic search* refers to the classical state-space decision-time planning methods. In these methods, a large tree of possible continuations is considered from an encountered state, and the approximate value functions of the leaf nodes are backed up toward the current state of the node.
    - Heuristic search can be seen as an extension of the idea of a greedy policy beyond just a single step. Often, when our search space is large and deep, heuristic search can be very computationally expensive, so searching only a few steps ahead is the only viable option.
    - It is important to note that heuristic search is most effective when it is devoted to imminent events and states, rather than just randomly selecting states.
- Roll Algorithms:
    - Rollout algorithms are decision-time planning algorithm based on MC control applied to simulated trajectories that begin at the current environment state. They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then following the given policy. The action with the

highest estimated value is executed. This is then executed again for the next state.

  - The aim of such an algorithm is not to estimate an optimal value function or a value function based off a given policy. Instead, it produces MC estimates of action values for each current state and for a given policy called the *rollout policy*. It discards the action-value estimates immediate after using it.
  - For two policies where $\pi'(s) = a \neq \pi(s)$ and $q_\pi(s, a) \geq v_\pi(s)$, π' is just as good, or better, than the π. This applies to rollout algorithm as well, where a policy that selects an action at s that maximises the action-value estimates, and thereafter follows π is a better policy than π.
  - The aim of a rollout algorithm is to improve upon the rollout policy, not to find the optimal policy. A rollout algorithm can perform quite well even when the rollout policy completely random. However, do note that the performance of the improved policy depends on properties of the rollout policy and the MC value estimates.
  - It is important to balance the computation time needed by the rollout algorithm to simulate trajectories, and the time constraints of the problem. Methods to overcome this problem is to run the MC trials in parallel since they are independent, or to truncate the simulated trajectories short of complete episodes.
- Monte Carlo Tree Search (MCTS): [Used in AlphaGo]
  - At its core, MCTS is a rollout algorithm, but it is improved upon by using MC value estimates to direct simulations toward more high-yielding trajectories. As in a rollout algorithm, MCTS is executed at each encounter of the next state, and it iteratively simulates many trajectories starting from the current state, running to the terminal state.
  - MCTS extends the initial portions of trajectories that have received high evaluations from earlier simulations, and it often retains action values between action selections, but not necessarily.
  - MC value estimates are maintained only for a subset of state-action pairs that are most likely reached in the next few steps. This forms a tree

    branching out from the current state (node). MCTS will then incrementally extend the tree by adding states (nodes) that look promising based on the results of simulated trajectories.

    - Here is an overview of the process in MCTS:
        - A tree branching out from the current state, which is the root node, is built by MC value estimates for the subset of state-action pairs that are most likely to be reached in the next few steps.
        - MCTS will also incrementally extend the tree by adding states (nodes) that look promising based on the results of simulated trajectories.
        - A *tree policy* is used to traverse the tree branching from the root node. This policy is an informed policy since we know the MC value estimates of the tree, and it is often one that balances exploration and exploitation.
        - Once we reach the leaf node of the tree, the rollout policy will be used from hereafter for the simulation until the terminal state.
        - The action values of the edges traversed in the tree by the tree policy are then backed up by the generated return. Note that the values of the state-action pairs visited by the rollout policy are not backed up.
    - After iterating through these steps continuously, an action is selected. In MCTS, the action with the largest visit count is selected to avoid selecting outliers. In the following state, the MCTS starts with the tree constructed from the previous execution of MCTS.
    - MCTS benefits from online, incremental, sample-based value estimate and policy improvement. It also avoids the problem of having the globally approximate an action-value function while benefiting from past experience to guide exploration.
- Part 1 Overview:
    - Page 190 provides a nice diagram which illustrates the different types of RL methods so far.
    - There are three key dimensions in the methods that we have explored so far: 1. Depth, 2. Width of Update, and 3. On/Off Policy. More aspects and

factors can be found in page 191.

- We have so far only explored tabular methods. It is time to move on to function approximation.

Exercises

8.1 It is unlikely that a multi-step bootstrapping method will be able to do better than a Dyna method. Similar to the one-step case, only n states can be updated every episode, and these updates are often quite slow and only makes a large change at the end of the episode. On the other hand with a Dyna method, the planning process can be considered quite efficient, as it randomly picks a state and action, and exploits it by updating the value estimate with the reward it receives from the episode. The state and action can be anywhere in the state-space, and not just the most recent n steps.

8.2 In the first phase for both experiments, exploration is much more important than exploitation, thus the more exploration-inclined Dyna-Q+ agent will perform better in the first phase, and maintain that good performance.

In the blocking experiment, the second phase requires exploration again to find the optimal policy. Again, the Dyna-Q+ agent will definitely outperform its 'normal' sibling, but note that Dyna-Q is still able to find its way to the optimal policy because it is forced to explore. In the shortcut experiment, the Dyna-Q+ agent is able to find its way through the shortcut because the additional reward bonus encourages the agent to explore for longer sequences. The ϵ-greedy policy in Dyna-Q will not be able to sustain a long enough sequence of exploration to find the new shortcut, thus there is no improvement for Dyna-Q.

8.3 The Dyna-Q+ agent is able to gain an advantage early on because of its exploratory nature. However, as time goes by, the Dyna-Q agent is able to catch up, and come up to parallel with the Dnya-Q+ agent. Moving later on, exploitation becomes more effective in achieving a high reward when compared to exploration. However, the Dyna-Q+ agent will still continue with its exploratory behaviour, which results in a slight decrease in its rewards. This is the reason for the narrowing of the performance gap between the two agents.

8.5 To model a stochastic environment, the model could assign a probability to each reward-next state pair $p(s', r|s, a)$ which is proportional to the number of times it has been observed in the past. However, a changing environment would mean that these true probabilities change as well, and the model can become incorrect quite quickly.

A way to tackle the problem of a changing environment is to give a smaller weight to the experience gained from far in the past, and give more weight to more recent experience. Some sort of exponential decay into the past can be used here.

8.6 With a distribution that is highly skewed, more likely transitions will occur more often in sample updates. This means that the value estimates that matter will be updated more frequently, and reach an estimate that is dramatically lower in error much quicker. On the other hand, the expected update will still have to go through the same amount of computation since the branching factor b does not change. This definitely strengthens the case for sample updates over expected updates.

8.7 I am assuming that scalloped refers to the jagged shape of the curve. One can notice that the graph becomes more jagged as b decreases. For the case of $b = 1$, the changes in a successor state's value estimate is completely transferred to the current state's value estimate, and this can contribute to the jagged shape of the curve. For $b > 1$, the updated value estimate is a weighted average of its successor states, so it is likely that there is a small variance of the difference in value estimates.

8.8 For the experiment with 10000 states and $b = 3$, we can note that the on-policy distribution works remains advantageous throughout, with a relatively significant margin as well. This is in agreement with what was mentioned before, where the on-policy distribution works better with problems with large state-space, and only a small sub-set of the state space is visited in the distribution.