Lecture notes - Introduction to Reinforcement learning with David Silver

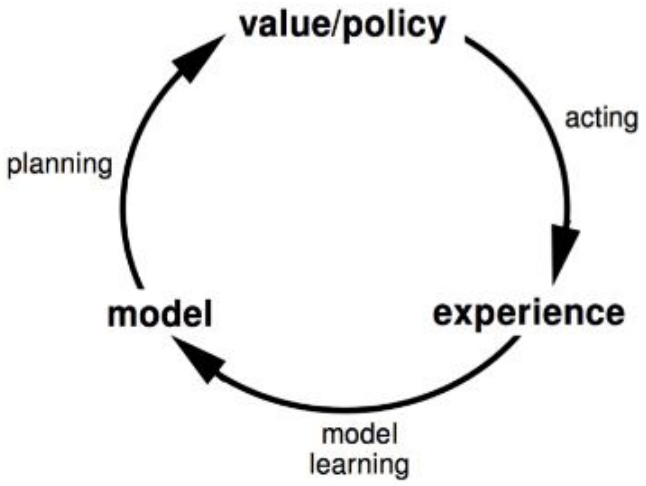
Lecture 8

By - Amal Sunny

Integrating Learning and Planning Methods

Model-Based RL

- So far, we've been learning either the policy or the value function from experience - to obtain the optimal path for maximum reward
- These were all model-free methods, they didn't require knowledge of the environment beyond the samples taken by the agent.
- In this lecture, we learn the model from experience, i.e. the agent learns the dynamics of the environment (we define the agent's model of env as-
  - what reward we get from an action in a state and
  - which state we end up from taking a certain action and probability of this transition.
- And after building the model, we apply **planning** - to construct a value function/policy to follow.
- This diagram can explain how te process looks like:



- Now starting from the experience part, the agent based on real experience in the actual environment - builds/learns the model and creates an internal replication of it.
  - Or alternatively, this is creating an MDP
- Based on the internal replica, it then plans a course of action (i.e a value function/policy).
  - Solving created MDP
- It then implements this course on the real environment, and records the experience gained to improve the model - thus repeating the loop.
  - Updating created MDP with real experience.

Advantages and Disadvantages

- Sometimes a model is more useful information to us then the corresponding value function
- And for games like chess, value functions are very erratic(your value can go up with the most trivial of changes in state space). But the model for chess is just the game rules, very simple to model
- Can efficiently learn model by supervised learning (labels are provided in the experience tuples)
- Can reason about model uncertainity - While learning the model, we can be sure of what we know and reevaluate what we don't. We aren't just stuck with finding the optimal in our current world view.

Disadvantages

- Two sources of approximation error - Constructing the model and subsequently the value function.

Model

- Formal defination: A model is a representation of an MDP $\langle S, A, P, R \rangle$, parametrized by η.
- We assume the state space and action space is known(for simplicity)
- So our model $M = \langle P_\eta, R_\eta \rangle$, represents state transitions $P_\eta \approx P$ and rewards $R_\eta \approx R$

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

- Essentially storing transition probabilities and immediate rewards for each state, action pair.

Model Learning

- Goal: estimate model M_η from experience tuples $\{S_1, A_1, R_2, \cdots, S_T\}$
- This is a supervised learning problem, where we have our experience tuples act as our labeled data.

$$S_1, A_1 \to R_2, S_2$$
$$S_2, A_2 \to R_3, S_3$$
$$\vdots$$
$$S_{T-1}, A_{T-1} \to R_T, S_T$$

- Learning $s, a \to r$ is a regression problem
- Learning $s, a \to s\prime$ is a density estimation problem
- Pick loss function, e.g. mean-squared error, KL divergence, ...
- Find parameters η that minimise empirical loss

Examples of Models:

- Table lookup (only one we cover)
  - Does not scale well for bigger models
- Linear expectation
  - We have a set of features describing each state and action, where we predict the features for next state it'll go to.
- Linear Gaussian
- Gaussian Process
- Deep Belief Network

Table Lookup Model

- Model is an MDP $\hat{P}, \hat{R}$
- We essentially count the number of times a state-action pair occurs, and take the frequency and reward during each occurence, averaged over all occurences of that state-action pair.

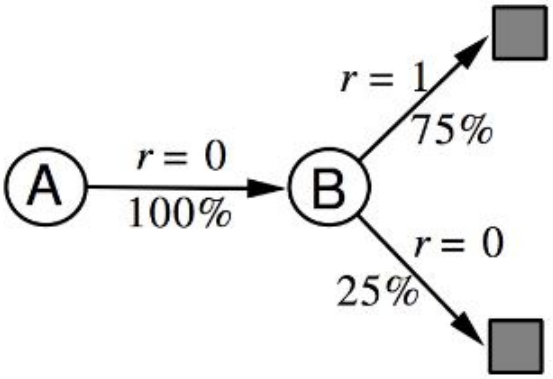
$$\hat{P^a_{s,s\prime}} = \frac{1}{N(s,a)} \sum_{t=1}^{T} 1(S_t, A_t, S_{t+1} = s, a, s\prime)$$

$$\hat{R^a_s} = \frac{1}{N(s,a)} \sum_{t=1}^{T} 1(S_t, A_t = s, a) R_t$$

- An example of the table lookup in action:

Two states A, B; no discounting; 8 episodes of experience

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0



Planning with a model

- Planning is simply solving the MDP we created.
- Given a model $M_\eta = \ \langle P_\eta, R_\eta$
- Solve the MDP $\ \langle S, A, P_\eta, R_\eta \rangle$
- We can solve them using any of the methods like:
  - Value iteration
  - Policy Iteration
  - Tree search

Sample-Based Planning

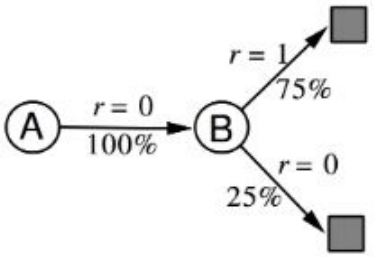
- Is a simple but pretty powerful approach to planning(supported by empirical results)
- We use the model(agent generated) **only** to generate samples(like start from any state, and generate a few state-action-reward samples)
- Then we apply model-free RL methods to those samples. Eg:
  - MC-control
  - SARSA
  - Q-learning
- Essentially, the agent simulates experience and based on the idealised solution to the simulation, it

formulates a course of action.

- Sample-based planning methods are often much more efficient than just applying those model-free ones directly.
- Taking the AB-example again, we first generate a table-lookup model from real experience
- Then sample from our model, and apply model-free RL to it.

Real experience

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0



Sampled experience

B, 1
B, 0
B, 1
A, 0, B, 1
B, 1
A, 0, B, 1
B, 1
B, 0

e.g. Monte-Carlo learning: $V(A) = 1, V(B) = 0.75$

- Under this, we can pull as many samples as our device can handle. So as we keep sampling more and more, we get a more accurate model.

Planning with an inaccurate Model

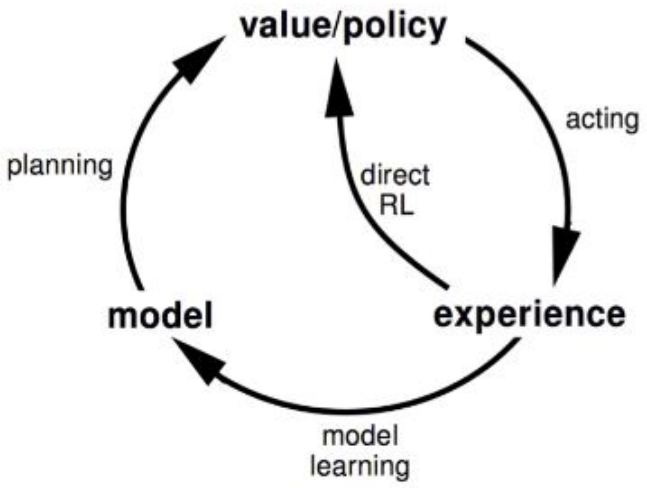
- So, what if our model is imperfect ?
- For starters, the performance of our model-based RL would be limited to optimal policy of that approximated MDP.
  - i.e. Model-based RL is only as good as our estimation of the model.
- Inaccuracy of the model would just lead to a suboptimal policy for the problem. However that shouldn't be a problem always, as it just means we haven't explored enough data to form a good image of the environment(model)
- However, if we know it isn't right there are solutions:
  - 1. When model is wrong, use model-free RL
  - 2. Reason explicitly why model is uncertain(states where we have not explored enough)

Integrated Architecture (Model-free and based) - Dyna

- So far, we have two sources of experience we can consider
  - Real experience - Sampled from the actual environment (True MDP)
    - $$S\prime \sim P^a_{s,s\prime}$$
      $$R = R^a_s$$
  - Simulated experience - Sampled from model (approximate MDP)
    - $$S\prime \sim_P \eta(S\prime|S, A)$$
      $$R = R_\eta(R|S, A)$$
- And we've covered these two paradigms

    - Model-Free RL
        - No model
        - Learn value function (and/or policy) from real experience
    - Model-Based RL (using Sample-Based Planning)
        - Learn a model from real experience
        - Plan value function (and/or policy) from simulated experience
- We introduce the fusion of these two architectures, the Dyna architecture
    - Learns a model from real experience
    - Learn and **plan** value function(/policy) from real and simulated experience.

Architecture



Dyna-Q Algorithm

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy(S,Q)
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

- Dyna-Q+ is a improved version that has a bonus for unexplored states(to encourage exploration)

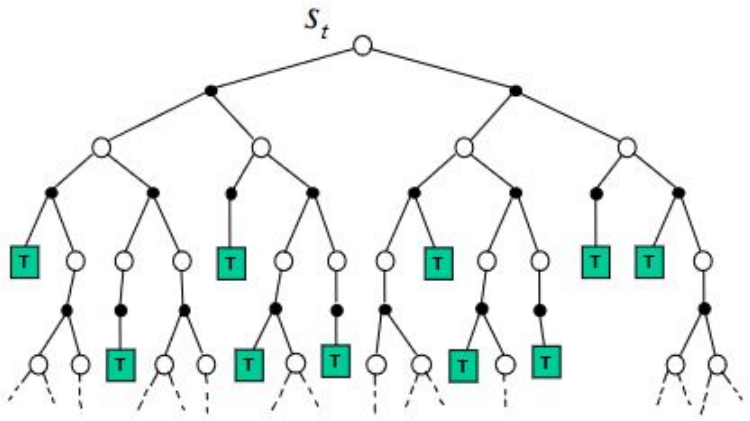
Simulation-Based Search

- Focuses on the effectively planning, based on a given model (SoTA methods for planning)
- Key ideas used -
    - Sampling

- Forward Search

Forward Search

- Forward search algorithms don't search the entire search space (for eg: dyna samples across the entire sample space), they focus on the **current** state we're in.
- We just want to decide on our next state, from the current one.
- Forward search algorithms achieve that by selecting the best action by lookahead.
- A search tree is built with current state at its root, and a model of the MDP is used to look ahead.



- We don't solve the whole MDP, we just solve the sub-MDP from this point onward.

Simulation-Based Search

- Its a forward search algorithm using sample-based planning.
- Based on the current state, it simulates episodes of experience with the current model.
- Then model-free RL is applied to those simulated episodes.

Working:

- Simulate episodes of experience with model:
  - $$\{s_t^k, A_t^k, R_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim M_v$$
    - superscript k, stands for # of the episode
- Apply model-free RL to these simulated episodes
  - MC-control -> MC-search
  - Sarsa -> TD search

Monte-Carlo Search

Simple MC Search

- We're given a model M_ν and a simulation policy π (**static policy**)
- For each action $a \epsilon A$
  - Simulate K episodes from current (real) state s_t, following policy π
    - $$\{s_t, a, R_{t+1}^k, S_{t+1}^k, A_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim M_\nu, \pi$$

- Evaluate actions by mean return (Monte-Carlo evaluation)
  - $$Q(s_t, a) = \frac{1}{K}\sum_{k=1}^{K} G_t \xrightarrow{P} q_\pi(s_t, a)$$
    - Here $\xrightarrow{P}$ refers to by sufficiently large samples, it would converge to
- Select current (real) action with maximum value
  - $$a_t = argmax_{a\epsilon A} Q(s_t, a)$$

MC Tree Search

- How MC tree differs from simple, is that we retain the information during the tree like sampling for each step- to calculate the Q-value for each of the nodes under the root node. This information is then used to improve the simulation policy π.
- Given a model M_ν
- Simulate K episodes from current (real) state s_t, using current simulation policy π
  - $$\{s_t, A_t^k, R_{t+1}^k, S_{t+1}^k, ..., S_T^k\}_{k=1}^K \sim M_\nu, \pi$$
- Build a search tree containing visited states and actions
- Evaluate states Q (s ,a) by mean return of episodes from s ,a
  - $$Q(s,a) = \frac{1}{N(s,a)}\sum_{k=1}^{K}\sum_{u=t}^{T} 1(S_u, A_u = s, a) G_u P \rightarrow q\pi(s,a) \xrightarrow{P} q_\pi(s_t, a)$$
- After search is finished, select current (real) action with maximum value in search tree
- $$a_t = argmax_{a\epsilon A} Q(s_t, a)$$
- Here, the simulation policy improves.
- Each simulation consists of two phases (already explored-in tree, out-of-tree)
  - Tree policy(improves): Pick to maximize Q value
  - Default(fixed): Pick randomly(when unknown)
- Repeat(every simulation)
  - Evaluate states under current state by MC evaluation
  - Improve tree policy, eg: ϵ-greedy(Q)
- This is essentially MC-control applied to simulated exp
- Therefore guarenteed to converge to optimal search tree, Q(S,A) $\rightarrow q_*(S, A)$

Advantages of MC Tree Search

- Highly selective best-first search
- Evaluates states dynamically, i.e. based on our current state onwards (unlike eg: DP, which learns entire state space)
- Uses sampling to break curse of dimensionality (all possible combinations aren't factored in together)
- Works for black-box models (we only need samples)
- Computationally efficient, anytime, parallelisable

TD Search

- So, obviously MC is not the only approach we can take to planning
- We apply TD instead of MC here (bootstrapping)
- So like how MC tree search applied, TD search applies SARSA to sub-MDPs from now.

MC vs TD

- As we already covered for model-free RL, bootstrapping is helpful
  - TD learning reduces variance but increases bias
  - TD learning is usually more efficient than MC
  - TD(λ) can be much more efficient than MC
- For simulation-based search, bootstrapping is still helpful
  - TD search reduces variance but increases bias
  - TD search is usually more efficient than MC search
  - TD(λ) search can be much more efficient than MC search

TD search

- Simulate episodes from the current (real) state s_t
- Estimate action-value function Q (s ,a)
- For each step of simulation, update action-values by Sarsa
  - $$\Delta Q(S, A) = \alpha(R + \gamma Q(S\prime, A\prime) - Q(S, A))$$
- Select actions based on action-values Q (s ,a)
  - e.g. ϵ-greedy
- TD search is effective in problems, where there are multiple ways to end up in one state - because its updating online, if we end up in already discovered state again we'll already have good info about it.
- For both MC and TD, there's no reason to be limited to search trees. We can use function approximators (and they can be helpful too)

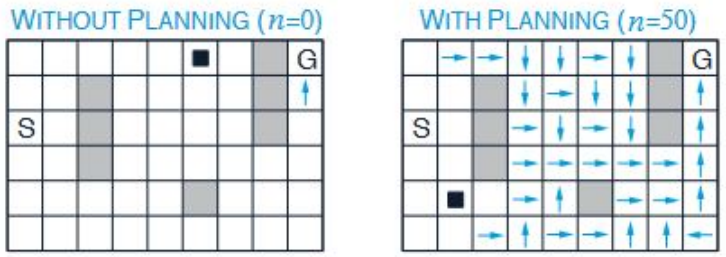
Dyna-2

- In Dyna-2, the agent stores two set of feature weights
  - Long-term memory
  - Short-term memory
- Long-term memory is updated from real experience using TD learning
  - General domain knowledge that applies to any episode, eg: a rock climber must his hands going up step by step, alternatively.
- Short-term memory is updated from simulated experience (to deal with the more immediate problem that arise), using TD search.
  - Specific local knowledge about the current situation.
- The overall value function is a sum of both, so it learns from both to deal with all possible situations.

Problem of the week

Problem 1: Exercise 8.1 - Sutton and Barto

The nonplanning method looks particularly poor in Figure 8.3 because it is a one-step method; a method using multi-step bootstrapping would do better. Do you think one of the multi-step bootstrapping methods from Chapter 7 could do as well as the Dyna method? Explain why or why not. (Here ch7 refers to n-bootstrapping methods like n-step sarsa, TD prediction, etc)



Ans: Yes, methods like n-step sarsa would definitely do better than the nonplanning version. As it only considers the immediate next step, instead of n-step sarsa which would end up with multiple states having a non-zero Q-value - as they would depend on more than just their immediate neighbour (which subsequently would lead to an optimal policy starting from state S) unlike nonplanning dyna here, after sufficient iterations.

Problem 2: Exercise 8.3 - Sutton and Barto

Careful inspection of Figure 8.5 reveals that the di↵erence between Dyna-Q+ and Dyna-Q narrowed slightly over the first part of the experiment. What is the reason for this?

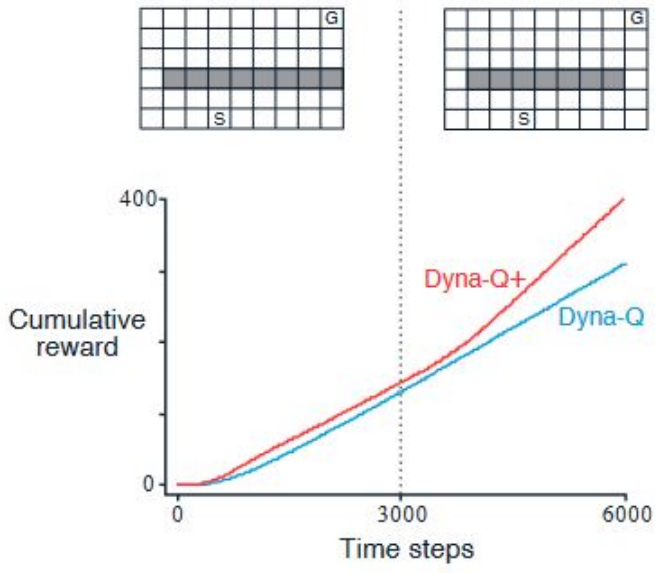


Figure 8.5: Average performance of Dyna agents on a shortcut task. The left environment was used for the first 3000 steps, the right environment for the rest.

Ans: Dyna Q+ is considered an improvement over Dyna Q due to the slight bonus it has for exploring unknown states. Due to this, Dyna Q+ is able to find the optimal policy quicker and has a higher reward gain than Dyna Q initially. But eventually Dyna Q catches up(and possible not all states are explored so Dyna Q+

might be incurring slight loss by exploring) and the gap becomes constant.

Program of the week

Beat the google dino game.