Reinforcement Learning David Silver - Lecture 8 Notes: Integrating Learning and Planning

Name: Eli Andrew

- **Advantages of Model-based RL**
  - Can efficiently learn model by supervised learning methods
  - Model is like the teacher that provides the supervised learning
  - Example:
    * Domain where learning policy or value function is hard (i.e. chess)
    * Many different states
    * Has sharp value function (single move of a piece can change from won position to lost position)
    * Hard to learn this type of value function directly
    * Model is straight forward - essentially just rules of the game
    * If you can use model to "look ahead" you can estimate the value function by planning (by tree search)
    * This is easy compared to learning the value function because you are just learning that you have 0 reward for all positions except check mates and draws
    * As compared to learning the value function where you are evaluating how likely you are to win from all the many configurations of the pieces
  - Model can be a more useful (and compact) representation of the information than a value function
  - Can reason about model uncertainty
    * Helps you see what you know and don't know about the world
    * This way you can strengthen your true understanding of the world and not just your current understanding
  - Disadvantage: learn model and then construct value function (2 sources of error)
- **What is a model**
  - Model M is a representation of an MDP $< S, A, P, R >$ parameterized by η
  - Assume state space and action space are known
  - Model $M =< P_\eta, R_\eta >$ represents state transitions $P_\eta \approx P$ and rewards $R_\eta \approx R$

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

- **Model learning**

- Goal: estimate model M_η from experience $\{S_1, A_1, R_2, \ldots, S_T\}$
- Supervised learning problem

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

- Learning $s, a \to r$ is a regression problem
- Learning $s, a \to s'$ is a density estimation problem (since it is likely stochastic we are learning the distribution)
- Pick loss function (MSE, KL divergence, ...)
- Find parameters η that minimize empirical loss

- **Examples of Models**
  - Table lookup model
  - Linear expectation model
  - Linear Gaussian model
  - Gaussian process model
  - Deep belief network model
  - ...

- **Sample-based Planning**
  - Use the model *only* to generate samples
  - Unlike DP where you look at probabilities of transitions and integrate over the probabilities
  - You sample experience from the model (rather than knowing all the transition probabilities)

$$
\begin{gathered}
S_{t+1} \sim P_\eta(S_{t+1} | S_t, A_t) \\
R_{t+1} = R_\eta(R_{t+1} | S_t, A_t)
\end{gathered}
$$

  - Apply *model-free* RL to samples: Monte-Carlo control, Sarsa, Q-learning, etc.
  - Sample based planning methods are often more efficient
  - Planning is essentially done by solving for the simulated experience drawn from the agents imagined world (its model)
  - Sampling is more efficient, even in the case when you know the entire model, because you are essentially focusing on the things that are most likely to happen

- **Dyna-Q**
  - Use a $Q(s, a)$ and a $Model(s, a)$ to make your decisions
  - Get your current $S_t = s$
  - Choose $A_t = a$ using your $\max_a Q(s, a)$
  - Observe your next state $S_{t+1} = s'$ and reward $R_{t+1} = r$
  - Update $Q(s, a)$ with standard Q-learning update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R_{t+1} + \gamma \max_a Q(s', a) - Q(s, a))$$

  - Add this example to your $Model(s, a)$ (assuming deterministic environment)

$$Model(s, a) \leftarrow s', r$$

  - $Model(s, a)$ is updated using supervised learning
  - Then use your $Model(s, a)$ for n iterations:

$$s = \text{random state you've seen before}$$
$$a = \text{random action you've taken from s before}$$
$$s', r = Model(s, a)$$
$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_a Q(s', a) - Q(s, a))$$

- **Foward Search**
  - Select best action by **lookahead**
  - Build **search tree** with current state s_t at root
  - Use **model** of MDP to look ahead
  - No need to solve for whole MDP, just sub-MDP starting from now
- **Simulation-based search**
  - **Forward search** paradigm using sample-based planning
  - **Simulate** episodes of experience from **now** with the model
  - Apply **model-free** RL to simulated episodes
    - Monte-Carlo control on simulated episodes is called **Monte-Carlo Search**
    - Sarsa control on simulated episodes is called **TD Search**
- **Simple Monte-Carlo Search**
  - Given a model M_v and a **simulation policy** π

- For each action $a \in A$:
  - Simulate K episodes from current (real) state s_t

$$\{s_t, a, R_{t+1}^k, S_{t+1}^k, A_{t+1}^k, \ldots, S_T^k\}_{k=1}^K \sim M_{v,\pi}$$

  - Evaluate actions by mean return (Monte-Carlo evaluation)

$$Q(s_t, a) = \frac{1}{K} \sum_{k=1}^{K} G_t \to q_\pi(s_t, a)$$

- Select current (real) action with maximum value

$$a_t = \arg\max_{a \in A} Q(s_t, a)$$

- In other words, look at the what you can do from current state (actions you can take)
- Then for each thing you can do from where you are - imagine what happens next (sample trajectories)
- Say that the average reward you get on all trajectories following what you do now is your estimate for how valuable it is to do that thing now
- Pick the action for what to actually do now by selecting the action that had the highest average return

• **Monte-Carlo Tree Search (Evaluation)**
  - Given a model M_v
  - Simulate K episodes from current state s_t using current simulation policy π

$$\{s_t, A_t^k, R_{t+1}^k, S_{t+1}^k, \ldots, S_T^k\}_{k=1}^K \sim M_{v,\pi}$$

  - Build a search tree containing visited states and actions
  - **Evaluate** states $Q(s, a)$ by mean return of episodes s, a

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

  - After search is finished, select current (real) action with maximum value in search tree

$$a_t = \arg\max_{a \in A} Q(s_t, a)$$

  - Leaves you with a rich tree history that can be used later (as compared to Simple Monte-Carlo search)

- **Monte-Carlo Tree Search (Simulation)**
  - In Monte-Carlo Tree Search, the simulation policy π improves
  - Each simulation consists of two-phases (in-tree, out-of-tree)
    - **Tree Policy** (improves): pick actions to maximize $Q(S, A)$
    - **Default Policy** (fixed): pick actions randomly
  - Repeat (each simulation):
    - **Evaluate** states $Q(S, A)$ by Monte-Carlo evaluation
    - **Improve** tree policy, e.g. by ϵ-greedy(Q)
  - **Monte-Carlo control** applied to **simulated experience**
  - Converges on the optimal search tree, $Q(S, A) \rightarrow q_*(S, A)$
  - The tree policy is always sending you in the current best (ϵ-greedy) trajectory, according to your current estimates. Those estimates are updated on each trajectory that runs through the state-action pair
- **Advantages of Monte-Carlo Tree Search**
  - Highly selective best-first search
    - Searches the current best path first - rather than trying to search all paths
  - Evaluates states *dynamically* (unlike in DP)
    - Dynamic programming evaluates whole state-space
    - Here we are focusing on where we are right now
  - Uses sampling to break curse of dimensionality
  - Works for "black-box" models (only requires samples)
  - Computationally efficient, anytime, parallelisible
- **Temporal-Difference Search**
  - Simulation based search
  - Using TD instead of MC (bootstrapping)
  - MC tree search applies MC control to sub-MDP from now
  - TD search applies SARSA to sub-MDP from now
  - Can be very effective in search spaces that are cyclic
  - Process
    - 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

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

    - Select actions based on action values $Q(S, A)$ (ϵ-greedy)
    - Can also use function-approximation for Q

- **Dyna-2**
  - Agent stores two sets of feature weights
    - **Long-term** memory
    - **Short-term** (working) memory
  - Long-term memory is updated from **real experience** using TD learning
    - General domain knowledge that applies to any episode
  - Short-term memory is updated from **simulated experience** using TD search
    - Specific local knowledge about the current situation
  - Overall value function is sum of long and short-term memories