Reinforcement Learning David Silver: Lecture 3 Notes

Name: Eli Andrew

- Dynamic programming
  - Dynamic sequential or temporal component to the problem
  - Program as in the policy we are trying to optimize
  - Method for solving complex problems by breaking into subproblems and then solving the subproblems and putting them together to arrive at a solution
  - Two necessary properties: (1) optimal substructure (problem can be broken into subproblems that can then be combined again) (2) overlapping subproblems (the subproblems occur many times)
  - MDPs have these two properties
    - Recursive decomposition is given by the Bellman Equation
    - Subproblem value caching can be acheived with the value function (stores all useful computed information about the MDP)
  - Assumes full knowledge of MDP and is used for planning
  - Can be used for prediction:
    - Input: MDP $< S, A, P, R, \gamma >$ and a policy π
    - Output: value function v_π
  - Can also be used for control:
    - Input: MDP $< S, A, P, R, \gamma >$
    - Output: optimal policy π_* and optimal value function v_*
- Policy evaluation
  - This is when you are told the MDP and the policy and you want to calculate the value of the policy v_π
  - General strategy: perform an iterative application of the Bellman expectation backup
  - Start with inital state values in v_1 and then apply Bellman expectation to get v_2 and continue to converge to v_pi
  - Using synchronous backups:
    - At each iteration $k + 1$
    - For all states $s \in S$:
    - Update $v_{k+1}(s)$ from $v_k(s')$ where s' is a successor state of s
    - Where $v_{k+1}(s) = \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s'))$
- Policy iteration

- Goal is to find the best possible policy in the MDP vs. evaluating a fixed policy like we did in policy evaluation
- One way to look at this is given a policy π how can we return a polciy π' that is better than π
- So, given a policy π
  - **Evaluate** the policy π: $v_\pi(s) = E[R_{t+1} + \gamma R_{t+2} + \ldots | S_t = s]$
  - **Improve** the policy by acting greedily with respect to v_π: $\pi' = greedy(v_\pi)$
- Policy iteration always converges to the optimal policy π_*
- On each iteration of this we generate a value function v_π which we act greedily on to get some new policy π', which on the next iteration gives a new value function $v_{\pi'}$ and then a new policy π'' and so on and so on until we obtain v_* and π_*
- Put another way:
  - Consider a deterministic policy: $a = \pi(s)$
  - We can improve the policy by acting greedily: $\pi'(s) = argmax_{a \in A} q_\pi(s, a)$
  - Acting greedily at least improves value for a single step: $q_\pi(s, \pi'(s)) = max_{a \in A} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$
  - Follow this logic to the rest of the steps and you can assume its better for the whole trajectory
  - When improvement stops we have the condition: $q_\pi(s, \pi'(s)) = max_{a \in A} q_\pi(a, s) = q_\pi(s, \pi(s)) = v_\pi(s)$
  - This then satisfies the Bellman Optimality Equation

- Any optimal policy can be broken into two pieces: (1) an optimal first action A_*, and (2) an optimal policy from the successor state S'
- Therefore a policy is optimal if the policy from all states we can reach from the current state is optimal.
- If we assume we know $v_*(s')$ then we can find $v_*(s') = max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')$
- The idea of value iteration is to start at the leaves with this assumption and then iteratively work your way back to arrive at the root
- Value iteration
  - Intuition: start with final rewards and work backwards
  - Still works with cyclic, stochastic MDPs
  - Important to note that this final rewards intuition is just intution. In practice all states are being updated on all iterations.
  - Problem: find optimal policy π

  - Solution: iterative application of Bellman optimality backup
  - $v_1 \dots v_2 \dots v_*$
  - Using synchronous backups:
    * At each iteration $k+1$
    * For all states $s \in S$
    * Update $v_{k+1}(s)$ from $v_k(s')$
    * $v_{k+1}(s) = max_{a \in A}(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s'))$
  - Unlike in policy iteration there is no explicit policy here
  - Intermediate value functions may in fact not be achieveable by any particular policy

- Overview:
  - In all these problems were given the MDP and its dynamics and we are trying to plan
  - **Problem:** Prediction, **Bellman Equation:** Bellman expectation, **Algorithm:** Iterative policy evaluation
  - **Problem:** Control, **Bellman Equation:** Bellman expectation + greedy policy improvement, **Algorithm:** Policy improvement
  - **Problem:** Control, **Bellman Equation:** Bellman optimality, **Algorithm:** Value iteration

- All algorithms are based on state-value functions $v_\pi(s)$ or $v_*(s)$
- Complexity $O(mn^2)$ per iteration, for m actions and n states
- Could also apply action-value function $q_\pi(s, a)$ or $q_*(s, a)$
- Complexity $O(m^2n^2)$ per iteration
- Extensions to DP:
  - DP methods so far used *synchronous* backups
  - i.e. all states are backed up in parallel
  - Asynchronous DP backs up states individually, in any order
  - For each selected state, apply the appropriate backup
  - Can significantly reduce computation
  - Guaranteed to converge if all states continue to be selected
- Asynchronous DP methods

  - In-place DP
  - Prioritized sweeping
  - Real-time dynamic programming

- In-place DP just stores a single state value for each state rather than storing the old value and the updated value. It just overwrites the old value with the new one. This causes subsequent updates that rely on the state value of that state to use the updated value rather than the old value but that is ok and can even help as it is essentially propagating information faster to other states. This brings into question how to pick the order to update states.

- Prioritized sweeping is born from the value gained by selecting the order of states updated in in-place DP.

  - Use the magnitude of Bellman error to guide selection:
    $|max_{a \in A}(R_s^a + \gamma \sum_{s' \in s} P_{ss'}^a v(s')) - v(s)|$
  - Backup the state with the largest remaining Bellman error
  - Update Bellman error of affected states after each backup
  - Requires knowledge of reverse dynamics (predecessor states)
  - Can be implemented efficiently by maintaining a priority queue

- Real-time dynamic programming

  - Idea: only states that are relevant to agent
  - Uses agent's experience to guide selection of states
  - After each time step S_t, A_t, R_{t+1}
  - Backup the state S_t
  - You are updating the states that the agent is actually visiting in the real world
  - $v_{S_t} = max_{a \in A}(R_{S_t}^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s'))$

- DP uses full-width backups

- For each backup (sync or async):
  (1) every successor state and action is considered
  (2) Using knowledge of MDP transitions and rewards

- DP is effective for medium-sized problems (millions of states)

- For large problems DP suffers from curse of dimensionality due to its states

- Number of states $n = |S|$ grows exponentially with number of state variables

- Even a single backup is too expensive in large problems

- Solution is to sample the MDP and perform updates just for that single sample