Lecture notes - Introduction to Reinforcement learning with David Silver

Lecture 3

By - Amal Sunny

Solving previously created problems

Planning by Dynamic Programming

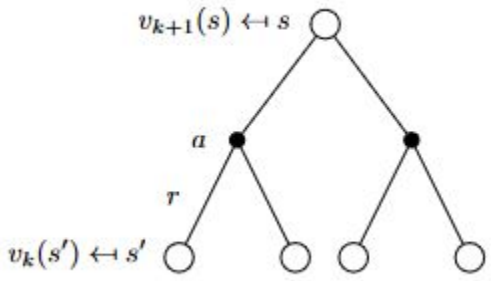
Dynamic Programming (DP)

- Problems that have some sequential strucuture that we solve step-by-step
- Dynamic programming looks to optimize that problem.
- Essentially, it breaks problems down into sub-problems
  - Solves them
  - Then combines them back to solve the bigger problem
- DP needs two properties to be applicable to a problem -
  - Optimal Substructure
    - i.e Dividing the problem into smaller problems, ensures the smaller problems being solved ends up solving the bigger problem
  - Overlapping Subproblems
    - The sub-problems recur repeatedly.
    - So, those solutions can be cached and reused
- MDPs satisfy both these properties
  - The bellman equation gives us a recursive decompositon of the problem
  - Value function stores and reuses solutions to the subproblems here
- DP assumes full knowledge of the MDP - this is only in context of planning for the MDP (other parts may not require or have full knowledge)
- Two cases of planning can be solved:
  - Prediction - predicting rewards attainable from some state
    - We have input: MDP and its policy or an MRP $< S, P^{\pi}, R, \gamma >$
    - Output: value function v_{π} - tells us reward obtained from any state

- Control - full optimization, figuring out the best action in the MDP
  - Input: MDP
  - Output: optimal value function v_* and optimal policy π_*
  - Essentially solving the MDP and get the most reward out of it

Policy Evaluation

- Given policy π , we have to evaluate it
- Solution: iterative application of the bellman expectation backup.
- So we evaluate $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_\pi$, where v_i is the value function for all states at the i^{th} iteration.
- We use synchronus backups to update these states:
  - At every iteration $k+1$
  - For all states $s \epsilon S$
  - Update $v_{k+1}(s)$ from $v_k(s')$
    - where s' is a successor state of s



- From this example, we can clearly see how for $v_{k+1}(s)$ at state s is computed by looking ahead at all possible actions, and using the weighted sum of probabilities of moving into the next state with the value function of those states **from the last iteration**.
- Putting this in equation form

$$v_{k+1}(s,a) = \sum_{a \epsilon A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \epsilon S} P_{ss'}^a v_k(s') \right)$$

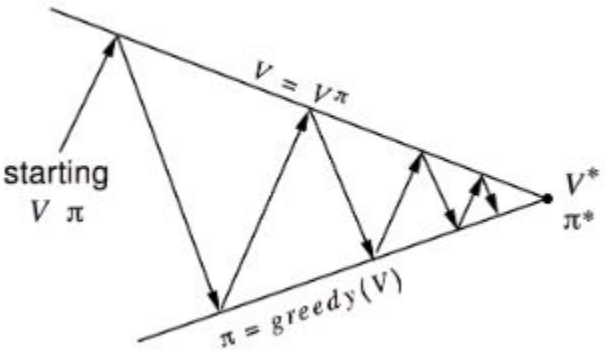
$$v^{k+1} = R^\pi + \gamma P^\pi v^k$$

- We iterate this process and its guarenteed to converge on the true value function.

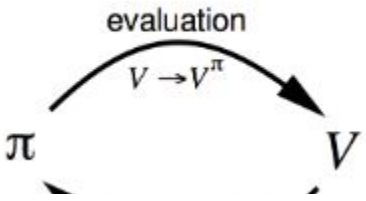
Policy Iteration

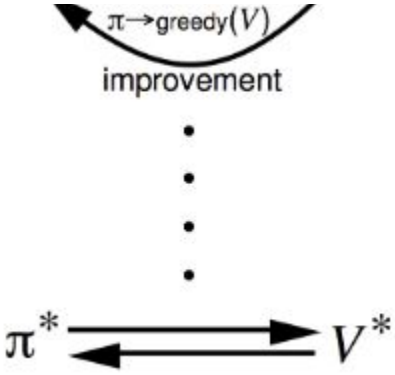
- To find best policy - given a certain policy, how do we improve on it ?
- Given a policy π
  - We first evaluate the policy π - we evaluate expected reward under it.
    - $$v_\pi(s) = E[R_{t+1} + \gamma R_{_{t+2}} + ... | S_t = s]$$
  - Improve the policy by acting greedily w.r.t to the data we just obtained from v_π
    - $$\pi' = greedy(v_\pi)$$
- We iterate over these two processess enough times.
- Eventually, this process **always** converges to optimal policy.

> There is a relationship between policy iteration, and expectation and maximization - a bit subtle and being explored in recent papers.



- How this entire process works:
  - Policy evaluation: We estimate v_π using given policy.
    - Iterative policy evaluation
  - Policy improvment: After estimating v_π, a greedy policy is discovered such that $\pi' \geq \pi$.
  - This policy is passed back to be evaluated again, and the cycle continues until we converge at optimal value function and policy.





Policy Improvment

- Consider a determinisitic policy, $a = \pi(s)$ - as in we skip the first step of randomly moving and consider the first iteration policy.
- We can improve the policy by acting greedily
  - $$\pi'(s) = argmax_{a\epsilon A} q_\pi(s,a)$$
  - i.e we pick actions in a way that gives us the maximum action value (q = immediate reward + value function depending on state where we end up)
- Doing this improves the value from any state s over one step,
  - $$q_\pi(s, \pi\prime(s)) = max_{a\in A} q_\pi(s,a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$$
  - Effectively here, we take the greedy action at the state s and follow the rest of the old policy from that statev_π
- From this we can see it improves the value function $v_{\pi'}(s) \geq v_\pi(s)$ as such:

$$\begin{aligned}\pi(s) \leq q_\pi(s, \pi\prime(s)) = E_{\pi\prime}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \\ \leq E_{\pi\prime}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi\prime(S_{t+1}))|S_t = s] \\ \leq E_{\pi\prime}[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi\prime(S_{t+2}))|S_t = s] \\ \leq E_{\pi\prime}[R_{t+1} + \gamma R_{t+2} + ...|St = s] = v_{\pi\prime}(s)\end{aligned}$$

- So, when we act greedily the value of the policy is atleast as good as the old one. It won't ever get worse, ever.
- But what about when the process stops, as the improvments stop (equality).
  - $$q_\pi(s, \pi\prime(s)) = max_{a\in A} q_\pi(s,a) = q_\pi(s, \pi(s)) = v_\pi(s)$$
- At that point, if we observe the equation we see that the bellman optimality equation gets satisfied.

- $v_\pi(s) = max_{a \in A} q_\pi(s, a)$

- Thus, at that point $v_\pi(s) = v_*(s), \forall s \in S$

Modified Policy Iteration

- In policy iteration, do we always needed to get to convergence for optimal policy ? NO. Sometimes we get ideal policy before convergence.
- At this point, all the extra iterations are wasted computation.
- So, two ways to handle this
  - Stopping condition - e.g: ϵ-convergence of value function
    - i.e check how minor the bellman equation updates values, if below threshold then we stop.
  - Improve policy only after a certain number of iterations (say k=3), naive approach - but stil converges to optimum policy.
  - Improving policy after every iteration, i.e k=1. This is called **value iteration**

Value Iteration

- An optimal policy can be divided into two components
  - The optimal first action A_*
  - And following the optimal policy from the sucessor state s' it ends up in

> Theorem (Principle of Optimality)
> A policy $\pi(a|s)$ achieves the optimal value from state s ,
> $v_\pi(s) = v_*(s)$, if and only if
>
> - For any state $s\prime$ reachable from s
> - π achieves the optimal value from state $s\prime$, $v_\pi(s\prime) = v_*(s\prime)$

Deterministic Value Iteration

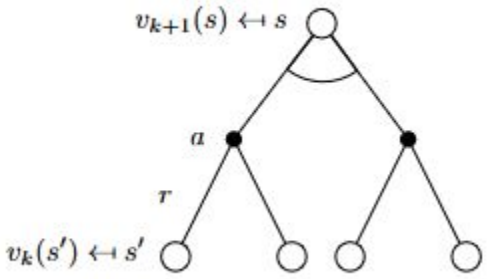
- Assuming we know the solution to subproblems $v_*(s')$
- Then solution for current step can be found by just one step lookahead at all actions and their subsequent rewards and value functions.

$$v_*(s) \leftarrow max_{a \in A} R_s^a + \gamma \sum_{s\prime \in S} P_{ss\prime}^a v_*(s\prime)$$

- We keep applying this iteratively, by first assigning some random value to $v_*(s')$ and then keep applying this equation and updating the value function and so on.
- Intuitively, you start with the final rewards and work backwards.

Value Iteration

- Given MDP , we have to find optimal value policy π
- Solution: iterative application of the bellman optimality backup.
- So we evaluate $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_*$, where v_i is the value function for all states at the i^{th} iteration.
- We use synchronus backups to update these states:
  - At every iteration $k+1$
  - For all states $s \epsilon S$
  - Update $v_{k+1}(s)$ from $v_k(s')$
    - where s' is a successor state of s
- Convergence to v_*, proven later in contraction mapping
- Unlike policy iteration, no explicit policy here.
- Intermediate value functions may not correspond to any policy.



- So at every state, we do a look ahead at the next action state pairs to evaluate the bellman equation as given below.
- Iterate until convergence.

$$v_{k+1}(s) = max_{a \in A}(R_s^a + \gamma \sum_{s\prime \in S} P_{ss\prime}^a v_k(s\prime))$$

$$v_{k+1} = max_{a \in A}(R^a + \gamma P^a v_k)$$

Summary of DP algorithms

Synchronous Dynamic Programming Algorithms

Problem	Bellman Equation	Algorithm
Prediction (v_π)	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

- Algorithms mentioned here are based on state-value functions $v_\pi(s)$ or $v_*(s)$
- Complexity $O(mn^2)$ per iteration, for m actions and n states.
    - Not great for big problems, but better than the alternative.
- Could apply to action-value function $q_\pi(s, a)$ or $q_*(s, a)$
    - But, we do not use them as they are worse w.r.t complexity - $O(m^2n^2)$ per iteration.

Asynchronous Dynamic Programming

- DP methods so far use synchronous backups.
- i.e all states are backed up in parallel, in one huge sweep as we iterate through them all.
- But in asynchronous, we select one particular state as our root and backup for that state and move on immediately, plug in that new value function back in the equation without having to wait till every single state in the table is updated.
- Can significantly reduce computation
- And that if the algorithm continues to select all states, regardless of the order, it is guarenteed to converge.
- Three ideas for asynchronous dynamic programming

- In-place dynamic programming
- Prioritised sweeping
- Real-time dynamic programming

In-place dynamic programming

- Synchronous value iteration stores two copies of value function (one for root, one for old iteration value)

for all s in S

$$v_{new}(s) \leftarrow max_{a \in A}(R_s^a + \gamma \sum_{s\prime \in S} P_{ss\prime}^a v_{old}(s\prime))$$

$$v_{old} \leftarrow v_{new}$$

- In-place value iteration only stores one copy of value function.

for all s in S

$$v(s) \leftarrow max_{a \in A}(R_s^a + \gamma \sum_{s\prime \in S} P_{ss\prime}^a v(s\prime))$$

- You just have to ensure you sweep in a way, the computations won't require old value again

Prioritised Sweeping

- We come up with a measure to maintain the priority of states, since we can sweep in any order.
- So a priority queue is maintained to update some states before others(an order essentially)
- Intuition is to keep the largest change of magnitude brought about in the state value function - also called the **Bellman error**
- Update bellman error of affected states after each backup
- Requires knowledge of reverse dynamics(predecessor state)

Bellman error

$$|max_{a \in A}(R_s^a + \gamma \sum_{s\prime \in S} P_{ss\prime}^a v(s\prime)) - v(s)|$$

Real-time Dynamic Programming

- Idea: only focus on states that are relevant to agent/ the ones that agent visits
- These states are collected based on agent's experiences in the real world/samples from its experiences.
- After each time-step S_t, A_t, R_{t+1}
- Backup the state S_t.

$$v(S_t) \leftarrow max_{a \in A}(R^a_{s_t} + \gamma \sum_{s\prime \in S} P^a_{S_t s\prime} v(s\prime))$$

Full-Width Backups

- DP uses full-width backups
  - i.e considers out all branches and states
  - This would be more expensive
- For large problems DP suffers from Bellman;s curse of dimensionality
  - Number of states $n = |S|$ growns exponentially with number of state variables.
- To solve this, we sample different trajectories throught the tree.
- We consider sample backups- i.e sample one action, one state and reward.
  - This cuts down on the size a lot
  - And because we're sampling we don't need to know the model.

Problems for the week

Problem 1: Parallelizing Value Iteration (Taken from Stanform CS234)

During a single iteration of the Value Iteration algorithm, we typically iterate over the states in S in some order to update $V_t(s)$ to $V_{t+1}(s)$ for all states s. Is it possible to do this iterative process in parallel? Explain why or why not.

Ans:

Value iteration update for a state s is given by

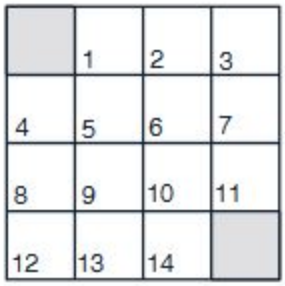
$$v_{k+1}(s) = max_{a \in A}(R^a_s + \gamma \sum_{s\prime \in S} P^a_{ss\prime} v_k(s\prime))$$

We can observe that $V_{t+1}(s)$ only depends on values in $V_t(.)$ and on not on any other value function in the same iteration. Thus, we can parallelize the calculations.

Problem 2: Taken from Sutton and Barto (Chapter 4)

Example 4.1 Consider the 4×4 gridworld shown below.

actions



$R_t = -1$
on all transitions

The nonterminal states are $\mathcal{S} = \{1, 2, \ldots, 14\}$. There are four actions possible in each state, $\mathcal{A} = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. Thus, for instance, $p(6, -1|5, \texttt{right}) = 1$, $p(7, -1|7, \texttt{right}) = 1$, and $p(10, r|5, \texttt{right}) = 0$ for all $r \in \mathcal{R}$. This is an undiscounted, episodic task. The reward is -1 on all transitions until the terminal state is reached. The terminal state is shaded in the figure (although it is shown in two places, it is formally one state). The expected reward function is thus $r(s, a, s') = -1$ for all states s, s' and actions a. Suppose the agent follows the equiprobable random policy (all actions equally likely). The left side of Figure 4.1 shows the sequence of value functions $\{v_k\}$ computed by iterative policy evaluation. The final estimate is in fact v_π, which in this case gives for each state the negation of the expected number of steps from that state until termination. ■

Exercise 4.1 In Example 4.1, if π is the equiprobable random policy, what is $q_\pi(11, \texttt{down})$? What is $q_\pi(7, \texttt{down})$? □

Ans:

We assume $\gamma = 1$ for this problem.

From the bellman equation, we know

$$q_\pi(s, a) = R_s^a + \gamma \sum_{s' \epsilon S} P_{ss'}^a v_\pi(s')$$

So, $q_\pi(11, down) = R_{11}^{down} + 0$ given terminal state right below 11.

Thus,

$$q_\pi(11, down) = -1$$

As for

$q_\pi(7, down) = R_7^{down} + \gamma P_{7,11}^{down} v_\pi(11)$

$q_\pi(7, down) = -1 + v_\pi(11)$

From the question, we know v_π is negation of expected number of steps to termination. So $v_\pi(11) = -1$

Thus,

$$q_\pi(7, down) = -1 - 1 = -2$$

Program for the week

Given any 2-D maze, design a solver for it.