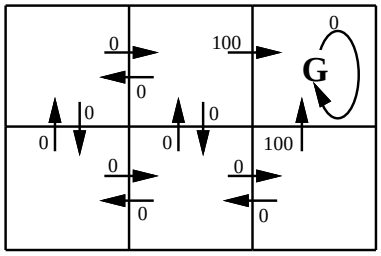
Lecture 20: Reinforcement Learning - Part II

- Methods for computing value functions
  - Dynamic programming
  - Monte Carlo
  - Temporal-difference learning

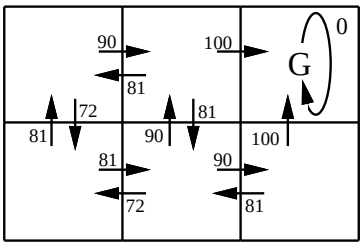
Recall from last time

- Reinforcement learning is learning from interaction with an environment
- No labeled examples; agent tries to optimize a *long-term* measure of performance
- Markov decision processes: states S, actions A, rewards r_s^a, next-state transition probabilities $p_{ss'}^a$, discount factor γ.
- The goal is to learn a *policy* $\pi : S \times A \to [0, 1]$ which maximizes the expected return (total reward)
- Value functions measure the expected total return
- In an MDP, there exists a unique optimal value function, which has at least one correspondig optimal policy
- How to compute the optimal value function/policy?

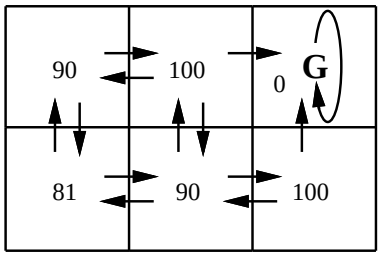
Illustration: A Deterministic Gridworld



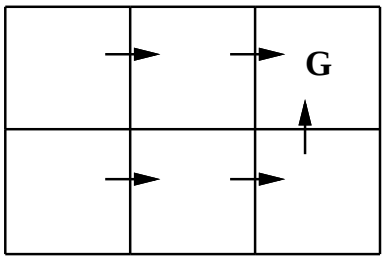
$r(s,a)$ (immediate reward) values



$Q^*(s,a)$ values ($\gamma = 0.9$)



$V^*(s)$ values



One optimal policy

Bellman Optimality Equation for V^*

The value of a state under the optimal policy must be equal to the expected return for the best action in the state:

$$\begin{aligned} V^*(s) &= \max_a Q^*(s,a) \\ &= \max_a E\left\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\right\} \\ &= \max_a \left(r_s^a + \gamma \sum_{s'} p_{ss'}^a V^*(s') \right) \end{aligned}$$

V^* is the **unique solution** of this system of non-linear equations

Value Iteration

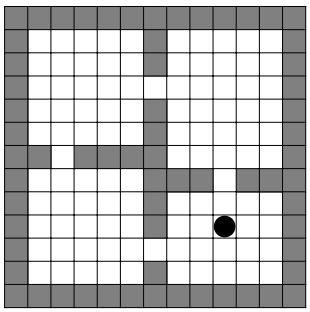
Main idea: Turn the Bellman optimality equation into an update rule (same as done in policy evaluation):

1. Start with an arbitrary initial approximation V_0
2. $V_{k+1}(s) \leftarrow \max_a r_s^a + \gamma \sum_{s'} p_{ss'}^a V_k(s), \forall s$

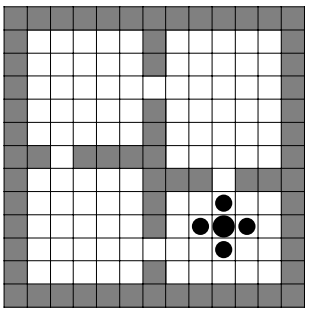
Illustration: Rooms Example

Four actions, fail 30% of the time

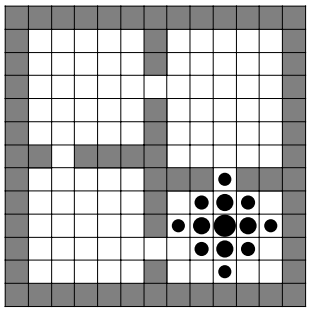
No rewards until the goal is reached, $\gamma = 0.9$.



Iteration #1



Iteration #2



Iteration #3

Policy Improvement

Suppose we have computed V^π for some deterministic policy π

When is it better to do an action $a \neq \pi(s)$?

$$Q^\pi(s,a) > V^\pi(s)$$

If we make the change at all states, we get a policy π' which is **greedy** with respect to Q^π:

$$\pi'(s) = \arg\max_a Q^\pi(s,a) = \arg\max_a r_s^a + \gamma \sum_{s'} p_{ss'}^a V^\pi(s')$$

Then $V^{\pi'}(s) \geq V^\pi(s), \forall s$

Policy Improvement (continued)

What if at some point $V^{\pi'} = V^\pi$?

Then we have:

$$V^\pi(s) = \max_a r_s^a + \gamma \sum_{s'} p_{ss'}^a V^\pi(s')$$

But this is the Bellman optimality equation!

So if the value does not change at some point, both π and π' are optimal.

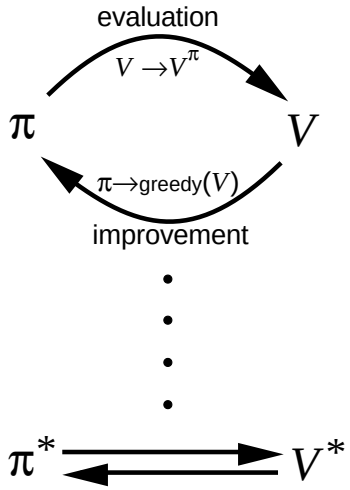
Policy Iteration

1. Start with an initial policy π_0
2. Repeat:
   (a) Compute V^{π_i} using policy evaluation
   (b) Compute a new policy π_{i+1} that is greedy with respect to V^{π_i}

   until $V^{\pi_i} = V^{\pi_{i+1}}$

Generalized Policy Iteration

Any combination of policy evaluation and policy improvement steps, even if they are not complete



How do we tie learning with dynamic programming?

- Observe transitions in the environment, learn an *approximate model* $\hat{r}_s^a, \hat{p}_{ss'}^a$

  Note that this is just a supervised learning problem!
- Pretend the approximate model is correct and use it for any dynamic programming method
- This approach is called **model-based RL**
- Many believers, especially in the robotics community

Asynchronous dynamic programming

- All the methods described so far require sweeps over the entire state space
- A more efficient idea: repeatedly pick states at random, and apply a backup, until some convergence criterion is met
- How should states be selected?

  Based on the agent's experience! I.e. along trajectories.
- Still needs lots of computation, but does not get locked into very long sweeps

Efficiency of DP

- Good news: finding an optimal policy is polynomial in the number of states
- Bad news: finding an optimal policy is polynomial in the number of states!

  Number of states is often astronomical; typically number of states is exponential in the number of state variables
- In practice, classical DP can be applied to problems with a few millions states
- Asynchronous DP can be applied even to larger problems, and is appropriate for parallel computation
- It is surprisingly easy to find problems for which DP methods are not feasible

Monte Carlo Methods

- Suppose we have an episodic task (trials terminate at some point)
- The agent behave according to some policy π for a while, generating several trajectories. How can we compute V^π?
- Compute $V^\pi(s)$ by **averaging the observed returns** after s on the trajectories in which s was visited.
- Two main approaches:
  - Every-visit: average returns for every time a state is visited in a trial
  - First-visit: average returns only for the first time a state is visited in a trial

Implementation of Monte Carlo Policy Evaluation

$$\begin{aligned} V_{n+1} &= \frac{1}{n+1}\sum_{i=1}^{n+1} R_i = \frac{1}{n+1}(\sum_{i=1}^{n} R_i + R_{n+1}) \\ &= \frac{n}{n+1}\frac{1}{n}\sum_{i=1}^{n} R_i + \frac{1}{n+1}R_{n+1} \\ &= \frac{n}{n+1}V_n + \frac{1}{n+1}R_{n+1} \end{aligned}$$

If we do not want to keep counts of how many times states have been visited, we can use a *learning rate* version:

$$V(s_t) \leftarrow V(s_t) + \alpha(R_t - V(s_t))$$

Monte Carlo Estimation of Q values

- We use the same idea: $Q^\pi(s,a)$ is the average of the returns obtained by starting in state s, doing action a and then following π
- Like the state-value version, it converges asymptotically *if every state-action pair is visited*
- But π might not choose every action in every state!
- *Exploring starts:* Every state-action pair has a non-zero probability of being the starting pair

Dynamic Programming vs. Monte Carlo

	DP	MC
Need model	yes	no (+)
Bootstrapping	yes (+)	no
Learn directly from interaction	no	yes (+)
Focus on visited states	no	yes (+)

Can we combine the advantages of both methods?