Lecture 19: Reinforcement Learning - Part I

- The reinforcement learning problem
- Brief history and example applications
- Markov Decision Processes
- What to learn: policies and value functions

Control Learning

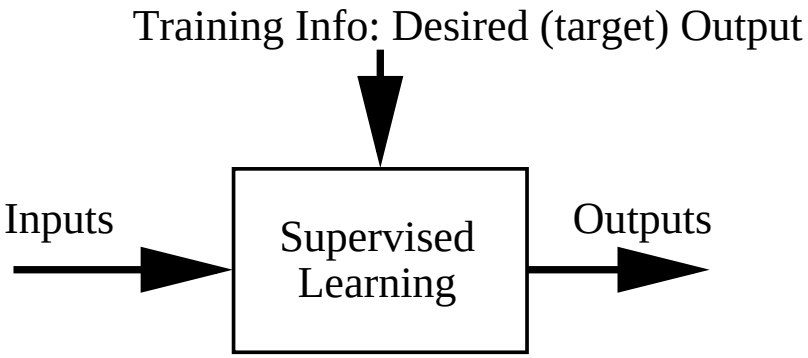
Consider *learning to choose actions*, e.g.,

- Robot learning to dock on battery charger
- Learning to choose actions to optimize factory output
- Learning to play Backgammon

Specific problem characteristics:

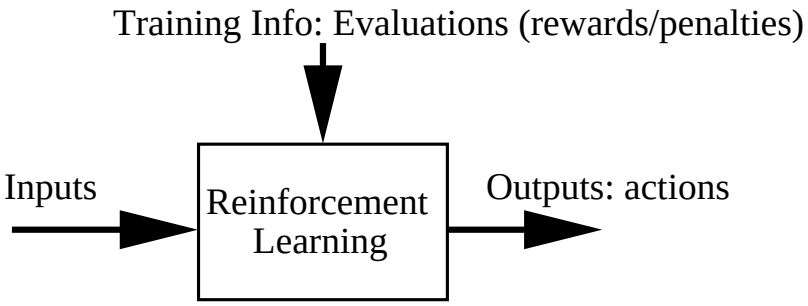
- Delayed reward
- Opportunity for active exploration
- There may not exist an adequate teacher!
- May need to learn multiple tasks using the same sensors/effectors

Supervised Learning



Error = (target output - actual output)

Reinforcement Learning (RL)



Objective: Get as much reward as possible

Key Features of RL

- The learner is not told what actions to take
- It find finds out what to do by trial-and-error search
- Possibility of delayed reward: sacrifice short-term gains for greater long-term gains
- Need to *explore* and *exploit*
- The environment is stochastic and unknown

Brief History

- Minsky's PhD thesis (1954): Stochastic Neural-Analog Reinforcement Computer
- Samuel's checkers player (1959)
- Ideas about state-action rewards from animal learning and psychology
- Dynamic programming methods developed in operations research (Bellman)
- Died down in the 70s (along with much of the learning research)
- Temporal difference (TD) learning (Sutton, 1988), for prediction
- Q-learning (Watkins, 1989), for control problems
- TD-Gammon (Tesauro, 1992) - the big success story
- Evidence that TD-like updates take place in dopamine neurons in the brain (W.Schultz et.al, 1996)
- Currently a very active research community, with links to different fields

Success Stories

- TD-Gammon (Tesauro, 1992)
- Elevator dispatching (Crites and Barto, 1995): better than industry standard
- Inventory management (Van Roy et. al): 10-15% improvement over industry standards
- Job-shop scheduling for NASA space missions (Zhang and Dietterich, 1997)
- Dynamic channel assignment in cellular phones (Singh and Bertsekas, 1994)
- Learning walking gaits in a legged robot (Huber and Grupen, 1997)
- Robotic soccer (Stone and Veloso, 1998) - part of the world-champion approach

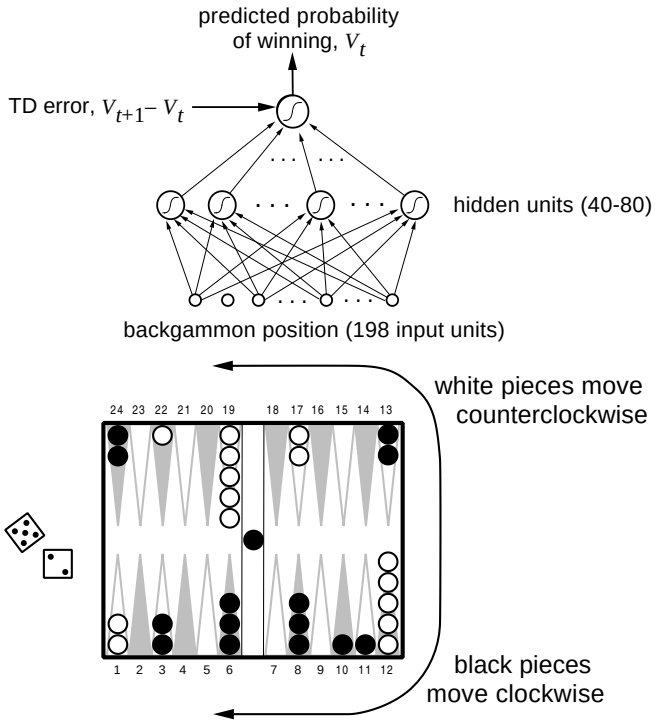
0.001in=0.401920.001in0.1in=0.401920.1in

All these are *large, stochastic optimal control problems*:

- Conventional methods require the problem to be simplified
- *RL just finds an approximate solution!*

An approximate solution can be better than a perfect solution to a simplified problem

TD-Gammon (Tesauro, 1992-1995)



TD-Gammon: Training Procedure

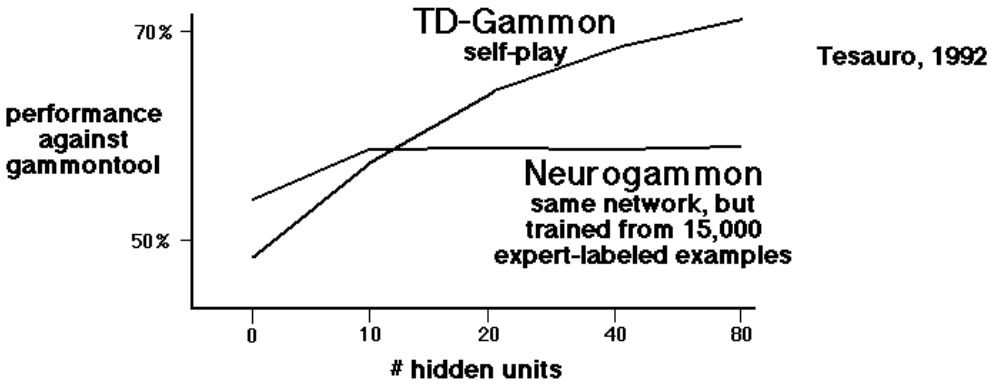
Immediate reward:

- +100 if win
- -100 if lose
- 0 for all other states

Trained by playing 1.5 million games *against itself*

Now approximately equal to best human player

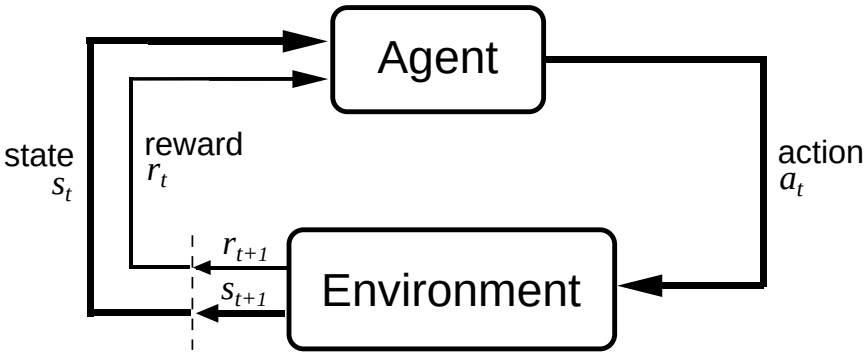
The Power of Learning from Experience



Expert examples are expensive and scarce

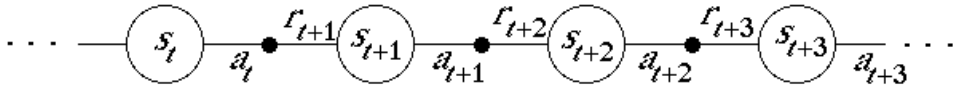
Experience is cheap and plentiful!

Reinforcement Learning Problem



- At each discrete time t, the agent observes state $s_t \in S$ and chooses action $a_t \in A$
- Then it receives an immediate reward r_{t+1} and the state changes to s_{t+1}

Markov Decision Processes (MDPs)



Assume:

- Finite set of states S (we will lift this later)
- Finite set of actions $A(s)$ available in each state s
- γ = discount factor for later rewards (between 0 and 1, usually close to 1)
- *Markov assumption:* s_{t+1} and r_{t+1} depend only on s_t, a_t and not on anything that happened before t

Models for MDPs

- r_s^a = expected value of the immediate reward if the agent is in s and does action a

$$r_s^a = E_{r_{t+1}}\{s_t = s, a_t = a\}$$

- $p_{ss'}^a$ = probability of going from s to s' when doing action a

$$p_{ss'}^a = E_{s_{t+1}=s'}\{s_t = s, a_t = a\}$$

These form the *model* of the environment, and are *usually unknown*

Agent's Learning Task

Execute actions in environment, observe results, and **learn policy** $\pi : S \times A \rightarrow [0, 1]$,

$$\pi(s, a) = \Pr\{a_T = a\} s_t = s$$

- Note that the target function is $\pi : S \rightarrow A$ but we have **no training examples** of form $\langle s, a \rangle$
  Training examples are of form $\langle \langle s, a \rangle, r ... \rangle$
- Reinforcement learning methods specify how the agent should change the policy as a function of the rewards received over time
- Roughly speaking, the agent's goal is to get as much reward as possible *in the long run*

Returns

Suppose the sequence of rewards received after time step t is $r_{t+1}, r_{t+2} \ldots$. We want to maximize the **expected return** $E\{R_t\}$ for every time step t

- Episodic tasks: the interaction with the environment takes place in episodes (e.g. games, trips through a maze etc)

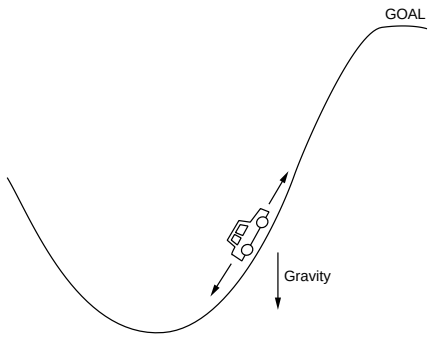
  $$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$

  where T is the time when a terminal state is reached
- Continuing tasks:

  $$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=1}^{\infty} \gamma^{t+k-1} r_{t+k}$$

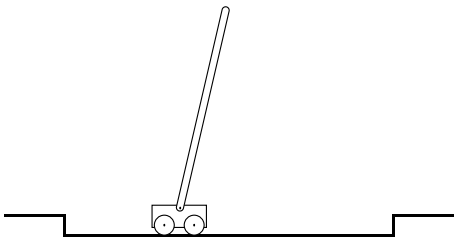
  where $0 \leq \gamma < 1$ is the discount factor for future rewards

Example: Mountain-Car



- States: position and velocity
- Actions: accelerate forward, accelerate backward, coast
- Rewards:
  - reward = -1 for every time step, until car reaches the top
  - reward = 1 at the top, 0 otherwise $\gamma < 1$
- Return is maximized by minimizing the number of steps to the top of the hill

Example: Pole Balancing



Avoid failure: pole falling beyond a given angle, or cart hitting the end of the track

- Episodic task formulation: reward = +1 for each step before failure

  $\Rightarrow$ return = number of steps before failure
- Continuing task formulation: reward = -1 upon failure, 0 otherwise, $\gamma < 1$

  $\Rightarrow$ return = $-\gamma^k$ if there are k steps before failure

Value Functions

For each possible policy π that the agent might adopt, we can define an evaluation function over states.

The **value of a state** is the expected return starting from that state, when following the policy:

$$V^\pi(s) = E_\pi\{R_t \mid s_t = s\} = E_\pi\left\{\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \mid s_t = s\right\}$$

The value of taking action a in state s and following policy π afterwards is the expected return when starting in that state, taking the action and following π afterwards:

$$Q^\pi(s,a) = E_\pi\left\{\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \mid s_t = s, a_t = a\right\}$$

Bellman Equation for Policy π

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \\ &= r_{t+1} + \gamma\left(r_{t+2} + \gamma r_{t+3} + \cdots\right) \\ &= r_{t+1} + \gamma R_{t+1} \end{aligned}$$

Based on this observation, V^π becomes:

$$V^\pi(s) = E_\pi\{R_t \mid s_t = s\} = E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s\}$$

Without the expectation:

$$V^\pi(s) = \sum_a \pi(s,a)\left(r_s^a + \gamma \sum_{s'} p_{ss'}^a V^\pi(s')\right)$$

This is a system of *linear equations* whose unique solution is V^π.

Iterative Policy Evaluation

Main idea: turn Bellman equation into an update rule.

1. Start with some initial guess V_0
2. During every iteration k, perform a **full backup of the value function**:

$$V_{k+1}(s) \leftarrow \sum_a \pi(s,a) \left(r_s^a + \gamma \sum_{s'} p_{ss'}^a V_k(s') \right)$$

3. Stop when the maximum change between two iterations is smaller than a desired threshold (the values stop changing)

Key idea: bootstrapping!

The value of one state is updated based on the values of the other states

Optimal Value Functions

- Policies can be partially ordered: $\pi \geq \pi'$ iff $V^\pi(s) \geq V^{\pi'}(s) \forall s$
- In an MDP there always exists at least one policy better than all others. This is called the **optimal policy**, π^*.
- The **optimal state-value function** is the value function shared by all optimal policies:

$$V^*(s) = \max_\pi V^\pi(s), \forall s \in S$$

- Similarly, we can define the **optimal action-value function**:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a), \forall s \in S, \forall a \in A$$

This is the expected value for taking action a in state s and following an optimal policy afterwards

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

Bellman Optimality Equation for Q^*

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

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

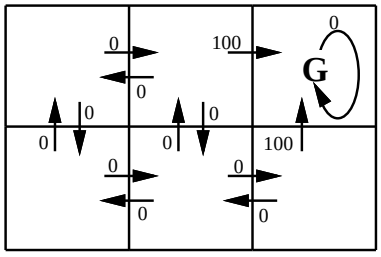
Why Optimal Value Functions are Useful

Any policy that is greedy with respect to V^* is an optimal policy!

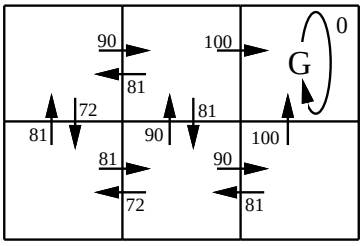
- If we know V^* and the model of the environment, one step of look-ahead will tell us what the optimal action is
- If we know Q^*, look-ahead is not even needed!

$$\pi^*(s) = \arg\max_a Q^*(s,a), \forall s$$

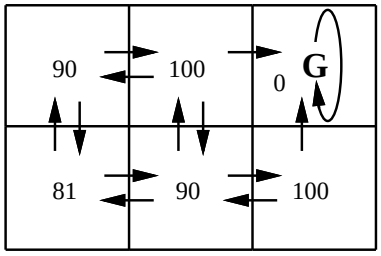
Illustration: A Deterministic Gridworld



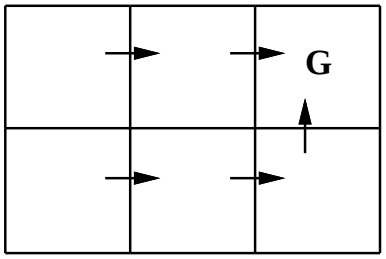
$r(s,a)$ (immediate reward) values



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

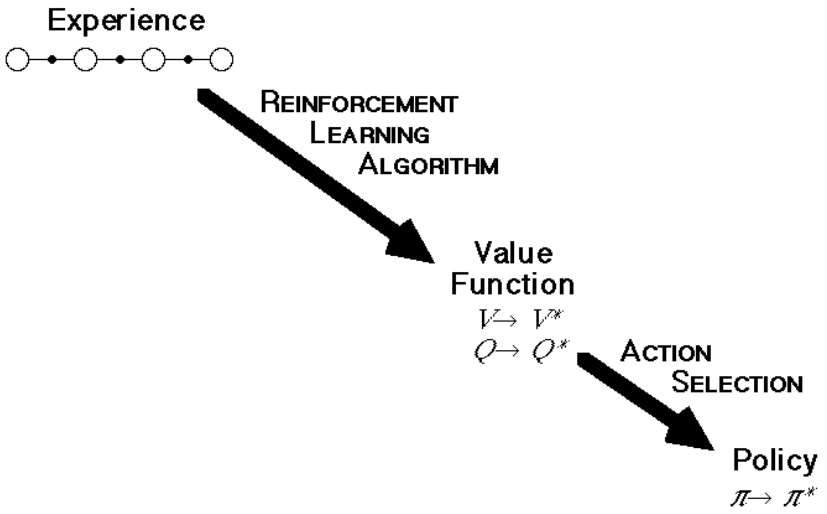


$V^*(s)$ values



One optimal policy

What RL Algorithms Do



Continual, on-line learning

Many RL methods can be understood as trying to solve the Bellman optimality equations in an approximate way.

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.

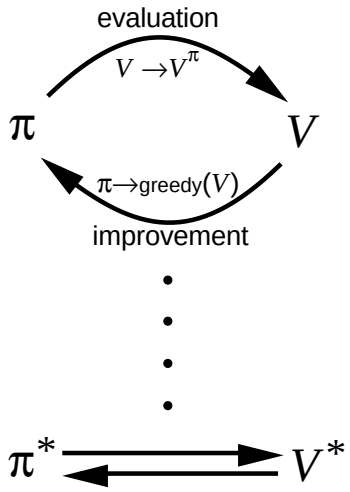
Policy Iteration

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



Value Iteration

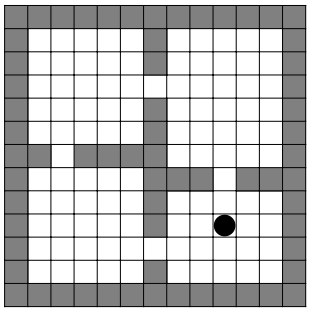
Main idea: Turn the Bellman optimality equation into an update rule (same as done in policy evaluation):

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

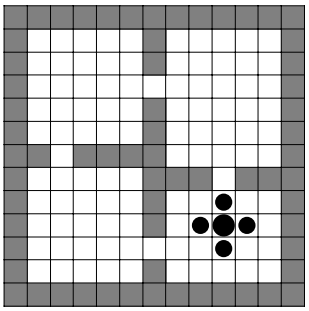
Illustration: Rooms Example

Four actions, fail 30% of the time

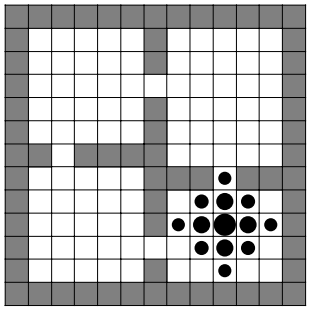
No rewards until the goal is reached, $\gamma = 0.9$.



Iteration #1



Iteration #2



Iteration #3

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