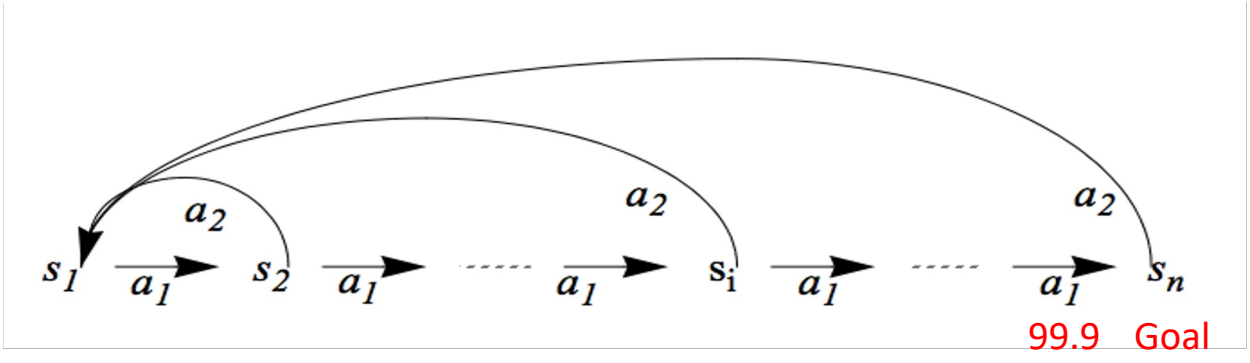
REINFORCEMENT LEARNING

Markov Decision Process

- **Markov Decision Process**
  - We learned that a Markov Decision Process (MDP) consists of
    - A set of states S (with an initial state s_0)
    - A set of actions A in each state,
    - A set of percepts that can be sensed
    - A transition model P(s'|s,a), or T(s,a,s')
    - ~~A sensor model θ = P(z|s)~~
    - The current state distribution
    - A reward function R(s,a,s') // careful here
  - The MDP's solution identifies the best action to take in each state
    - Optimal policy π(s)=a obtained by value iteration or policy iteration (aka dynamic programming)

An Example for MDP

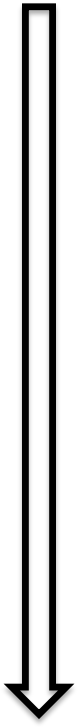
- Given
  - States: $s_1, \ldots, s_n$, Actions: a_1, a_2
  - Transition Probability Distribution: (assume $s_{n+1}=s_1$)
    - $P(s_i,a_1,s_{i+1})=0.8$, $P(s_i,a_1,s_1)=0.2$, $P(s_i,a_2,s_1)=0.9$, $P(s_i,a_2,s_{i+1})=0.1$.
  - Reward: all $R(s_i, a_i, s_j) = 0.0$, except $R(s_{n-1}, a_1, s_n) = 99.9$
  - We define the goal state to be s_n
  - The Future discount factor: gamma $\gamma = 0.7$
- State utility values: $U(s_{n-1})$, $U(s_{n-2})$, $U(s_{n-3})$, … and $U(s_2)$, $U(s_1)$
- Optimal Policy: $\pi(s_i)=a_1$



Reinforcement Learning

- **Given a Markov Decision Process (environment)**
  - A set of states S (known) ✓
  - A set of actions A in each state ✓ (known)
  - A set of percepts that can be sensed ✓ (known)
  - The current state (known) ✓
  - Transitions P(*s'*| *s*, *a*) ✗ (only known by the env)
  - Rewards R(*s*, *a*, *s'*) → 没有最后一步有 ✗ (only known by the env)
- **Could the agent still learn the optimal policy?**

Goals, Rewards, Utilities, Policies

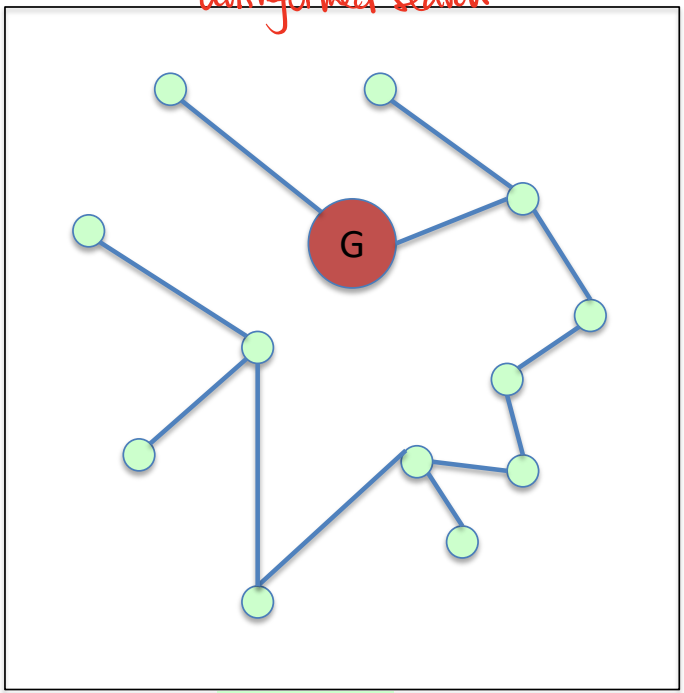


- Goals
  - Given to the agent from the problem statements
- Rewards
  - Given to the agent, designed based on the goals
- Utility values for states
  - Computed by the agent, based on the rewards
- Policies
  - Computed or learned by the agent
  - Used by the agent to select its actions
  - The better a policy, the more rewards it collects

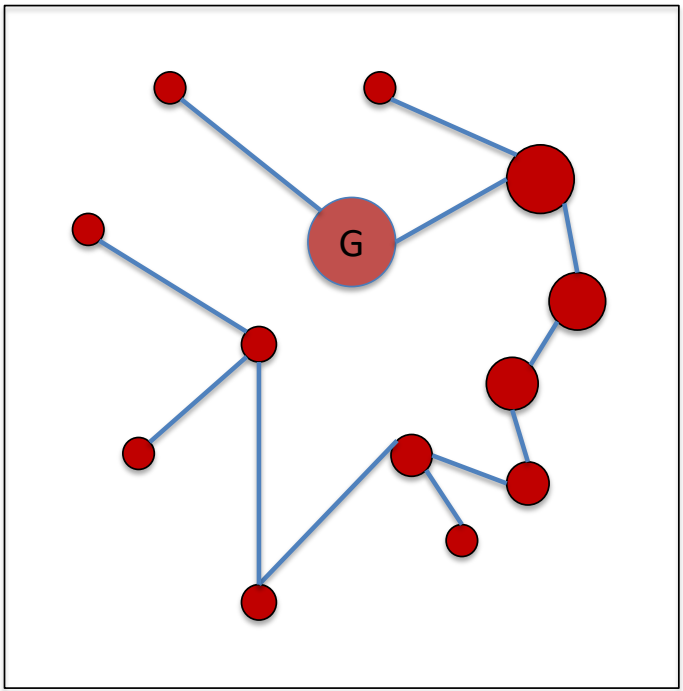
Reinforcement Learning

Propagating the Delayed Rewards

uninformed search



Before RL



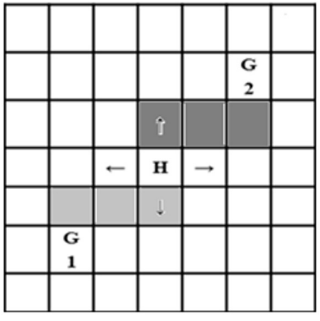
After RL

After RL, no matter where you are, you know which way to go to the Goal!
This is because all states now have utility values that will lead your agent to the goal.

Reinforcement Learning (Key Ideas)

- Can the agent still learn the optimal policy?
  - When it knows only the states S and actions A, but not the transition model P(s,a,s') and/or reward function R(s,a,s')
- Try an action on a state of the environment, and get a "sample" that includes (s, a, s', r) and maybe U:
  - A state transition (s,a)->s'
  - A reward received for the action r = R(s, a, s')
  - And maybe the utility value of the next state U(s')
- Objective: Try different actions in states to discover an optimal policy $\pi(s)=a$ and eventually tells the agent which action will lead to the most rewarding states

How to Explore the State Space



- Visit different states to discover a policy and improve it
  - Numerous strategies
  - A simple strategy is executing actions randomly
    - Initially there is no policy, but random actions eventually discover a policy
    - At every time step, act randomly with a probability (for exploration), otherwise, follow the current policy $\pi(s)=a$
      - The random action may causes unnecessary action execution when the optimal policy is already discovered
      - One solution is to lower the probability of selecting a random action over time (i.e., reduce the "temperature")

2. types

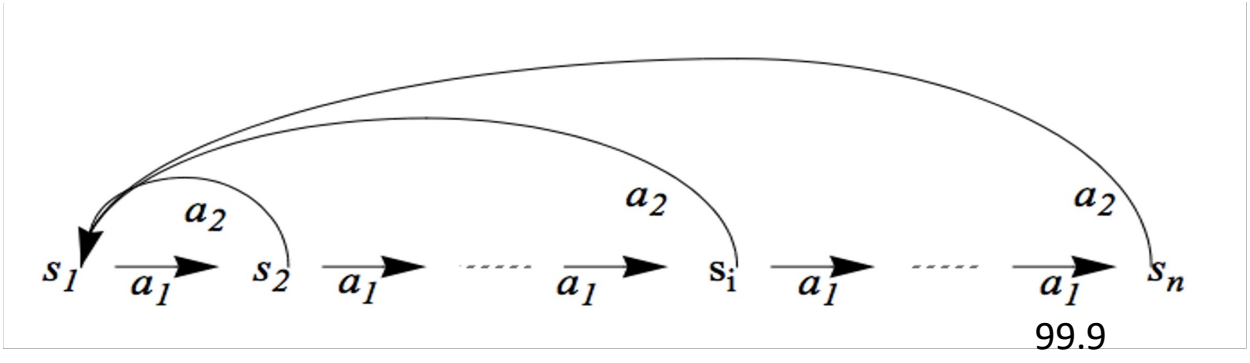
Reinforcement Learning

- Objective: Learn the optimal policy $\pi^*(s)=a$
- Two general classes of algorithms:
  1. **Model-based RL**
     - First learn the transition model and the utility values of states, then learn the policy (e.g., policy iteration)
  2. Model-free RL
     - Learn the policy without learning an explicit transition model, but by receiving "samples" from the environment
     - Three algorithms we will learn:
       - Monte Carlo
       - Temporal Difference
       - Q-Learning

3. model-based RL

An Example for Model-Based RL

- Given
  - States: $s_1, \ldots, s_n$,
  - Actions: a_1, a_2
  - Probabilistic transition model: (assume $s_{n+1}=s_1$)
    - $P(s_i,a_1,s_{i+1})=0.8$, $P(s_i,a_1,s_1)=0.2$, $P(s_i,a_2,s_1)=0.9$, $P(s_i,a_2,s_{i+1})=0.1$.
  - Reward: all $R(s_i, a_i, s_j) = 0.0$, except $R(s_{n-1}, a_1, s_n) = 99.9$
  - Future discount factor: $\gamma = 0.7$
- Try to learn: $U(s_{n-1})$, $U(s_{n-2})$, $U(s_{n-3})$, … and $U(s_2)$, $U(s_1)$



Model-Based RL

- Learn an approximate model based on random actions
  - From a state s, count outcomes of s' for each (s, a)
  - Normalize to give an estimate of P(s, a, s')
  - Learn state utility U(s) from rewards R(s, a, s') when encountered
  - Learn a policy (e.g., policy iteration) and solve the MDP

State Utility Value Iteration
(improving $U(s)$ every step)

- $U(s)$: the expected sum of maximum rewards achievable starting from a particular state s
- Bellman equations:
  - Many equations must be solved *simultaneously*

$$U^*(s) = R(s) + \gamma \max_a \sum_{s'} T(s,a,s') U^*(s')$$

- Bellman iteration:
  - Converge to $U^*(s)$ step by step

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s,a,s') U_i(s')$$

Policy Iteration
(improving $\pi(s)$ every step)

- Start with a randomly chosen initial policy π_0
- Iterate until no change in utilities:
- Policy evaluation: given a policy π_i, calculate the utility $U_i(s)$ of every state s using policy π_i by solving the system of equations:

$$U_\pi(s) = R(s) + \gamma \sum_{s'} T(s,\pi(s),s') U_\pi(s')$$

- Policy improvement: calculate the new policy π_{i+1} using one-step look-ahead based on $U_i(s)$:

$$\pi_{i+1}(s) = \operatorname*{argmax}_a \sum_{s'} T(s,a,s') U^*(s')$$

State Utility Value Iteration (review)

(improving $U(s)$ every step)

- $U(s)$: the expected sum of maximum rewards achievable starting from a particular state *s*
- Bellman equations:
  - Many equations must be solved *simultaneously*

$$U^*(s) = R(s,a,s') + \gamma \max_a \sum_{s'} T(s,a,s') U^*(s')$$

- Bellman iteration:
  - Converge to $U^*(s)$ step by step

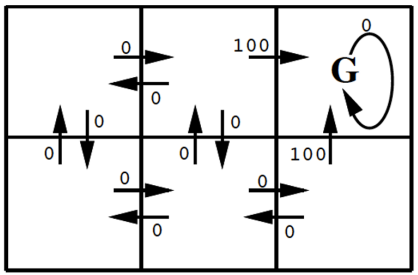
$$U_{i+1}(s) \leftarrow R(s,a) + \gamma \max_a \sum_{s'} T(s,a,s') U_i(s')$$

Utility Value Iteration (example)

Find your way to the goal

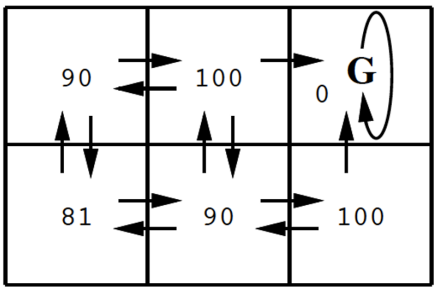
Use Bellman Iteration to compute state values from rewards

$$U_{i+1}(s) \leftarrow R(s,a) + \gamma \max_a \sum_{s'} T(s,a,s')U_i(s')$$

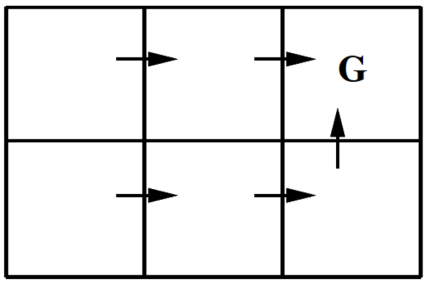


$r(s,a)$ (immediate reward) values

γ=0.9



$V^*(s)$ values



One optimal policy

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad 0 \le \gamma < 1$$

Policy Iteration (review)

(improving π(s) every step)

- Start with a randomly chosen initial policy π_0
- Iterate until no change in the utility values of the states:
- Policy evaluation: given a policy π_i, calculate the utility value $U_i(s)$ of every state *s* using policy π_i by solving the system of equations:

$$U_\pi(s) = R(s, \pi(s), s') + \gamma \sum_{s'} T(s, \pi(s), s') U_\pi(s')$$

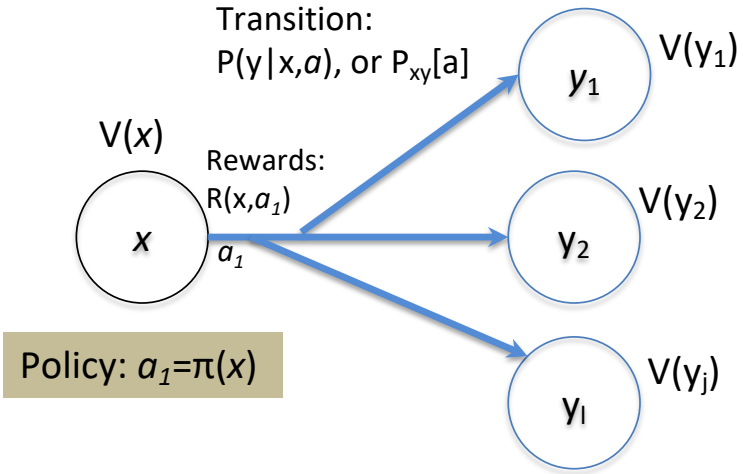
- Policy improvement: calculate the new policy π_{i+1} using one-step look-ahead based on $U_i(s)$:

$$\pi_{i+1}(s) = \operatorname*{argmax}_a \sum_{s'} T(s, a, s') U^*(s')$$

Compute Utility from Rewards and Policy

to determine an optimal policy, one that maximizes the total discounted expected reward. By *discounted reward*, we mean that rewards received m steps later are worth less than rewards received now by a factor of $\gamma^m (0 < \gamma < 1)$. Under a policy π, the value of state x (again with respect to $R(x, a)$) is

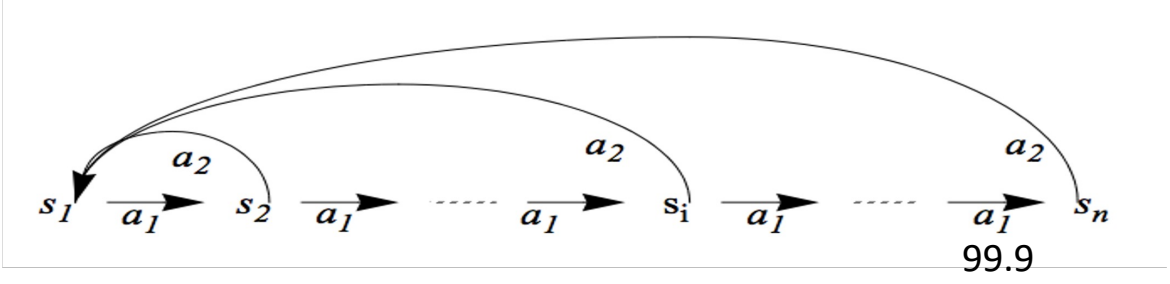
$$V^\pi(x) \equiv R(x, \pi(x)) + \gamma \sum_y P_{xy}[\pi(x)] V^\pi(y)$$



Assume the transitions are known

An Example for Model-Based RL

- Given
  - States: $s_1, \ldots, s_n$ Actions: a_1, a_2
  - Probabilistic transition model: (assume $s_{n+1}=s_1$)
  - $P(s_i,a_1,s_{i+1})=0.8$, $P(s_i,a_1,s_1)=0.2$, $P(s_i,a_2,s_1)=0.9$, $P(s_i,a_2,s_{i+1})=0.1$.
  - Reward: all $R(s_i, a_i, s_j) = 0.0$, except $R(s_{n-1}, a_1, s_n) = 99.9$
  - Future discount factor: $\gamma = 0.7$, all Initial utility values are: $U(s_{i-})=0$
- 1st iteration
  - $U(s_{n-1})=0.7\max\{0.8(99.9+0)+0.2(0+0), \ldots, \ldots\} = 55.944$; or
  - $U(s_{n-1})=\max\{0.8(99.9+0.7*0)+0.2(0+0), \ldots, \ldots\} = 79.92$;
  - $U(s_{n-2})=0$, $U(s_{n-3})=0$, …, $U(s_1)=0$
- After many iterations: $U(s_1)$ = ? Is it $99.9*(0.7*0.8)^{n-1}$?



$$U_{t+1}(s) \leftarrow \gamma \max_a \sum_{s'} P(s,a,s')[R(s,a,s') + U_t(s')]$$

$$U_{t+1}(s) \leftarrow \max_a \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma U_t(s')]$$

Model-Based RL

- Learn an approximate model based on random actions
  - From a state s, count outcomes of s' for each (s, a)
  - Normalize to give an estimate of P(s, a, s')
  - Learn state utility U(s) from rewards R(s, a, s') when encountered
  - Solve the learned MDP and obtain a policy (e.g. policy iteration)

Pros:

If and when the goal changes, one can use the learned model P(s, a, s') to recalculate U(s) and compute a new policy offline

Cons:

Larger representation than model-free RL

Model-Free Reinforcement Learning

- Objective: Learn the optimal policy $\pi^*(s)=a$
- Model-based RL
  - First learn the transition model and the utility values of states, then learn the policy (e.g., policy iteration)
- **Model-free RL**
  - Learn the policy without learning an explicit model, but by receiving "samples" from the environment
  - Three algorithms we will learn:
    - Monte Carlo
    - Temporal Difference
    - Q-Learning

Monte Carlo RL (model free)

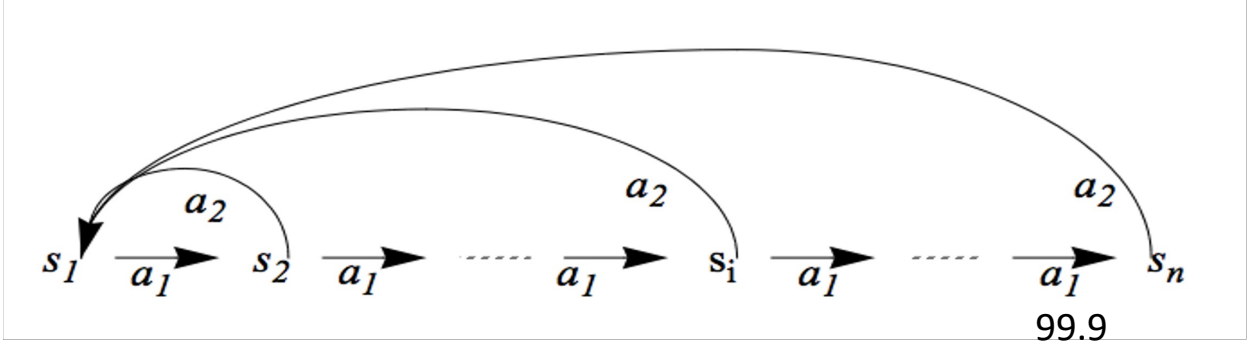
- Generate a fixed policy π(s) = a based on U(s)
  - In each episode, the learner follows the policy starting at a random state
  - Average the sampled returns originating from each state
  - Generate a policy based on U(s) as in "policy iteration"
- Cons:
  - The utility value of states U(s) is given and fixed (not learned)
  - Works only in episodic problems
  - Takes a very long time to converge as learning is from complete sample returns
  - Wastes information as it figures out state values in isolation from other states

Temporal Difference RL (model free)

- Improve Monte Carlo, still using a fixed policy
  - Update U(s) using each sample (s, a, s', r) received from the environment and each sample includes:
    1. A state transition (s,a)->s' done by the environment
    2. A reward received for the action r = R(s, a, s')
    3. And the value of the next state U(s')
  - That is, Sample = r + γU(s'), where γ = future discount
- **TD Algorithm:**
  - U(s) ← (1-α)U(s) + α*Sample = (1-α)U(s) + α[r + γU(s')]
  - Where α is the "learning rate" (how much you trust the sample)
  - You might decrease α over time when converges to the average
  - The parameter α also represents "forgetting long past values"
- Cons:
  - Only provides the utility values of states, not improve policy directly

An Example for TD-RL

- Given
  - States: $s_1, \ldots, s_n$,
  - Actions: a_1, a_2
  - Probabilistic transition model: (assume $s_{n+1}=s_1$)
    - $P(s_i,a_1,s_{i+1})=0.8$, $P(s_i,a_1,s_1)=0.2$, $P(s_i,a_2,s_1)=0.9$, $P(s_i,a_2,s_{i+1})=0.1$.
  - Reward: all $R(s_i, a_i, s_j) = 0.0$, except $R(s_{n-1}, a_1, s_n) = 99.9$
  - We define the goal state to be s_n
  - The learning rate $\alpha = 0.4$
  - The Future discount factor: $\gamma = 0.7$
- Use TD algorithm to learn: $U(s_{n-1})$, $U(s_{n-2})$, $U(s_{n-3})$, … and $U(s_2)$, $U(s_1)$



Q-Learning (Definition)

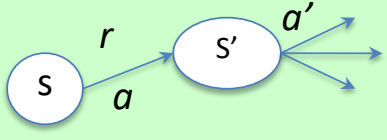
- The key idea is to use $Q(s,a)$ to represent the value of taking an action a in state s, rather than only the value of state U(s):
  - Define: $$Q(s,a) \equiv \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma U(s')]$$
- Optimal $\pi(s) = \text{argmax } U^{\pi}(s) = \text{argmax}_a Q(s, a)$

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

  - The last term using Q to replace U

Q-Learning (Algorithm)

- Initially, let $Q(s, a)=0$, given α and γ
- Sample a transition and reward: (s,a,s',r)
  - Sample = $r + \gamma \max_{a'} Q(s',a')$
  - $Q(s,a) = (1-\alpha)Q(s,a) + \alpha * \text{Sample}$



$$Q_{t+1}(s,a) = (1-\alpha)Q_t(s,a) + \alpha[R(s,a,s') + \gamma \max_{a'} Q_t(s',a')]$$

Advantages:

1. No need for a fixed policy, can do off-policy learning, or directly learns a policy
2. The optimal policy $\pi(s)=a$ is to select the action a that has the best $Q(s, a)$
3. Provably convergent to the optimal policy when t -> infinite

The Q-Learning Algorithm

Initialize all $Q(s, a)$ arbitrarily
For all episodes
  Initalize s
  Repeat
    Choose a using policy derived from Q, e.g., ϵ-greedy
    Take action a, observe r and s'
    Update $Q(s, a)$:
      $Q(s, a) \leftarrow Q(s, a) + \eta(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    $s \leftarrow s'$
  Until s is terminal state

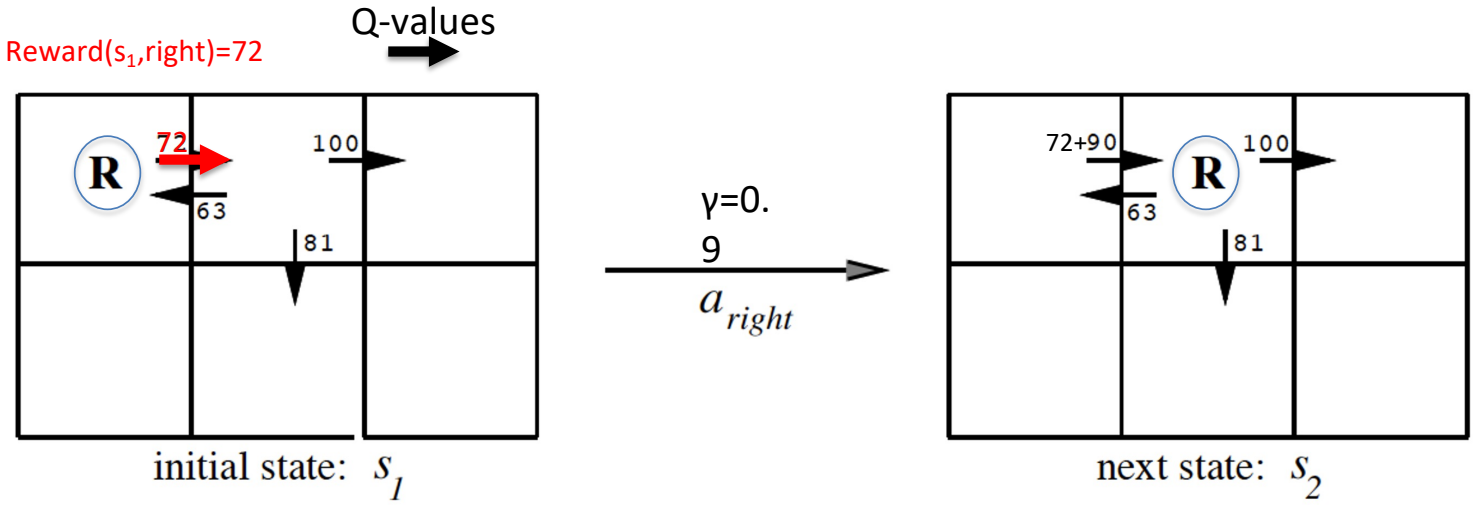
Q-Learning (in One Formula)

sample

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

$r(s)$ or $r(s,a)$

Q-Learning example from reward r(s,a)



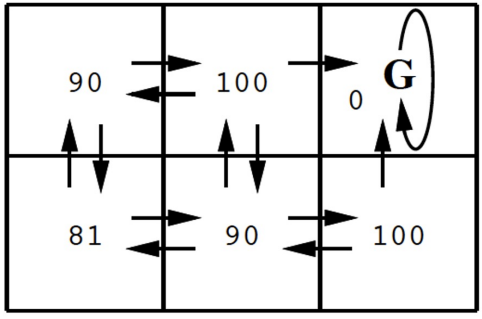
$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \qquad \text{// when } \alpha = 1$$

$$72 + 0.9 \max\{63, 81, 100\}$$

$$72 + 90$$

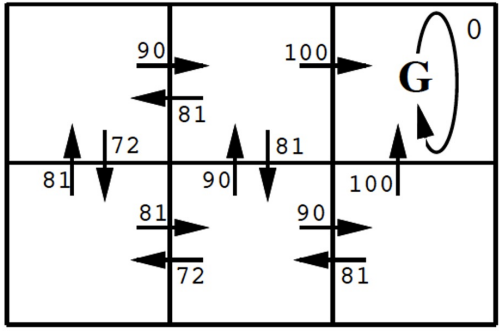
Back-propagate Q-values 1-step backwards

Q-Learning from state utility values



$V^*(s)$ values

Q-Learning

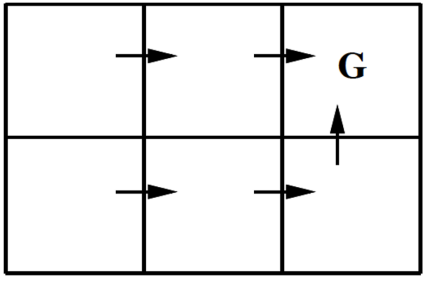


$Q(s,a)$ values

Notice these

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

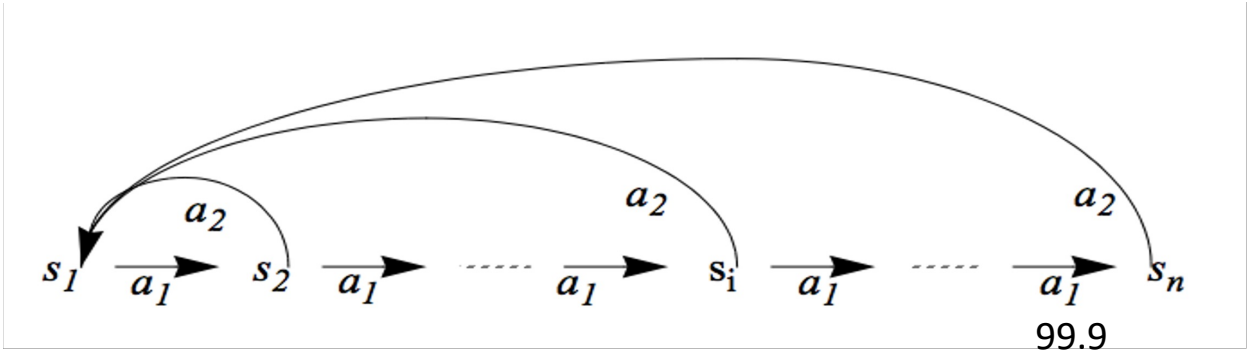
$$V^*(s) = \max_a Q(s,a)$$



One optimal policy

An Example for Q-Learning

- Given
  - States: $s_1, \ldots, s_{n,}$
  - Actions: a_1, a_2
  - Probabilistic transition model: (assume $s_{n+1}=s_1$)
    - $P(s_i,a_1,s_{i+1})=0.8$, $P(s_i,a_1,s_1)=0.2$, $P(s_i,a_2,s_1)=0.9$, $P(s_i,a_2,s_{i+1})=0.1$.
  - Reward: all $R(s_i, a_i, s_j) = 0.0$, except $R(s_{n-1}, a_1, s_n) = 99.9$
  - We define the goal state to be s_n
  - The learning rate $\alpha = 0.4$
  - The Future discount factor: $\gamma = 0.7$
- Try to learn: $Q(s_{n-1},a_1)$, $Q(s_{n-1},a_2)$, $Q(s_{n-2},a_1)$, and $Q(s_{n-2},a_2)$=?



An Example for Q-Learning

Initially: all $Q(s_i, a_j) = 0.0$

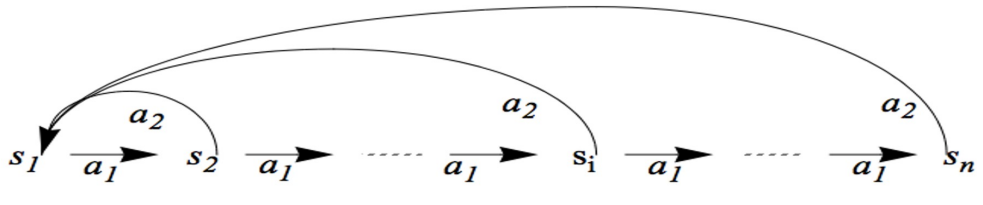
Assume you got two samples at the state s_{n-1} for action a_1

- s_{n-1}, a_1, s_n, $r=R(s_{n-1}, a_1, s_n)=99.9$, // $P(s_{n-1}, a_1) \rightarrow s_n$
- s_{n-1}, a_1, s_1, $r=R(s_{n-1}, a_1, s_1)=0.0$, // $P(s_{n-1}, a_1) \rightarrow s_1$

Updated Q value, if P(s,a) are known:

- $Q(s_{n-1}, a_1)= 0.8*[99.9 + 0.7*0.0] + 0.2[0.0+0.7*0.0] = 79.9$

- If P(s,a) are unknown, assume s_{n-1} was visited 2 times so far:
  - $sample_1=99.9+.7*0$; $Q(s_{n-1}, a_1)= (1-0.4)*0.0 + 0.4*99.9 = 39.96$
  - $sample_2=0+.7*0$; $Q(s_{n-1}, a_1)= (1-0.4)*39.96 + 0.4*0.0 = 23.976$



99.9

(4)

Discussions of RL (1)

- Pros:
  - Easy to use in problems where the data can be mapped to states easily
  - Guaranteed to find the optimal policy given enough time even with suboptimal actions
- Cons:
  - States of the environment are not always known
  - Computation time is intractable for large or continuous state spaces
    - E.g. if each cell in a grid world is a state, then state space grows exponentially with number of rows and columns (states = map size = m*n)
  - Cannot handle raw data, must use an approximation/reduction function
    - Designing approximation functions to disambiguate similar states requires human intelligence or an alternate learning technique
      - E.g. use the relative distance (states = m*n) between two agents in a hunter-prey problem as opposed to their cell coordinates (states = m*n*m*n)
  - Model-free RL cannot transfer the learned knowledge when the goal changes
    - Forgetting a learned policy is much more difficult (hysteresis), quicker to start from scratch

Discussions of RL (2)

Utility function is unknown at the beginning

- When agent visits each state, it receives a reward
  - Possibly negative

What function should it learn?

- $R(s)$: Utility-based agent
  - If it already knows the transition model
  - Then use MDP algorithm to solve for MEU actions
- $U(s,a)$: Q-learning agent
  - If it doesn't already know the transition model
  - Then pick action that has highest U in current state
- $\pi^*(s)$: reflex agent
  - Learn a policy directly, then pick the action that the policy says

Q Learning Summary

Modify Bellman equation to learn *Q* values of actions

- $U(s) = R(s)+\gamma \max_a \sum_{s1}(P(s_1|s,a)U(s_1))$
  - We don't know *R* or *P*
  - But when we perform *a'* in *s*, we move to *s'* and receive *R*(*s*)
  - We want *Q*(*s*,*a*) = Expected utility of performing *a* in state *s*

Update *Q* after each step

- If we get a good reward now, then increase *Q*
- If we later get to a state with high *Q*, then increase *Q* here too
- $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(R(s)+\gamma \max_{a'} Q(s',a'))$
  - α is the learning rate, γ is the discount factor

Converges to correct values if α decays over time

- Similar to the "temperature" in simulated annealing

5. eg.

Q-Learning in Star War Environment

1 	2	3	4 
5		6	7 
8	9	10	11

$\alpha = 0.1$

Step 1 $\gamma = 1$

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(R(s)+\gamma \max_{a'} Q(s',a'))$$

1 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	2 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	3 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	4 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**
5 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**		6 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	7 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**
8 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	9 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	10 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	11 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**

Step 1

$$Q(1,R) \leftarrow (1-\alpha)Q(1,R) + \alpha(R(1)+\gamma \max_{a'} Q(2,a'))$$

$\Rightarrow -0.004$

$Q(2,R)$ 和 $Q(3,R)$ 也相同 $= -0.004$

1 U: 0.0 D: 0.0 L: 0.0 R: ~~0.0~~ -0.004	2 U: 0.0 D: 0.0 L: 0.0 R: ~~0.0~~ -0.004	3 U: 0.0 D: 0.0 L: 0.0 R: ~~0.0~~ -0.004	4 U: 0.0 D: 0.0 L: 0.0 R: 0.0
5 U: 0.0 D: 0.0 L: 0.0 R: 0.0		6 U: 0.0 D: 0.0 L: 0.0 R: 0.0	7 U: 0.0 D: 0.0 L: 0.0 R: 0.0
8 U: 0.0 D: 0.0 L: 0.0 R: 0.0	9 U: 0.0 D: 0.0 L: 0.0 R: 0.0	10 U: 0.0 D: 0.0 L: 0.0 R: 0.0	11 U: 0.0 D: 0.0 L: 0.0 R: 0.0

Step 4

$Q(4,*) \leftarrow 1$

1 **U: 0.0** **D: 0.0** **L: 0.0** **R: -0.004**	2 **U: 0.0** **D: 0.0** **L: 0.0** **R: -0.004**	3 **U: 0.0** **D: 0.0** **L: 0.0** **R: -0.004**	4 **+1**
5 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**		6 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	7 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**
8 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	9 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	10 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	11 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**

Step 5

$$Q(1,R) \leftarrow (1-\alpha)Q(1,R) + \alpha(R(s)+\gamma \max_{a'} Q(2,a'))$$

1 **U: 0.0** **D: 0.0** **L: 0.0** **R: -0.004**	2 **U: 0.0** **D: 0.0** **L: 0.0** **R: -0.004**	3 **U: 0.0** **D: 0.0** **L: 0.0** **R: -0.004**	4 **+1**
5 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**		6 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	7 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**
8 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	9 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	10 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	11 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**

SARSA: State-Action-Reward-State-Action

Modify Q learning to use chosen action, a', not max

- $Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + \alpha\,[R(s,a,s') + \gamma\, \max_{a'} Q_t(s',a')]$
- $Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + \alpha\,[R(s,a,s') + \gamma\, Q_t(s',a')]$

On-policy, instead of off-policy

- SARSA learns based on real behavior, not optimal behavior
  - Good if real world provides obstacles to optimal behavior
- Q learning learns optimal behavior, beyond real behavior
  - Good if training phase has obstacles that won't persist

SARSA: State-Action-Reward-State-Action

Initialize all $Q(s, a)$ arbitrarily

For all episodes

  Initalize s

  Choose a using policy derived from Q, e.g., ϵ-greedy

  Repeat

    Take action a, observe r and s'

    Choose a' using policy derived from Q, e.g., ϵ-greedy

    Update $Q(s, a)$:

$$Q(s, a) \leftarrow Q(s, a) + \eta(r + \gamma Q(s', a') - Q(s, a))$$

    $s \leftarrow s', \ a \leftarrow a'$

  Until s is terminal state

Step 5: SARSA

which is closer to −0.04 than −0.076

0.9 × −0.004 + 0.1 × (−0.004 − 0.004) = −0.008

$$Q(1,R) \leftarrow (1-\alpha)Q(1,R) + \alpha(R(s)+\gamma\, Q(2,R))$$

1 **U: 0.0** **D: 0.0** **L: 0.0** **R: ~~-0.004~~**	2 **U: 0.0** **D: 0.0** **L: 0.0** **R: -0.004**	3 **U: 0.0** **D: 0.0** **L: 0.0** **R: -0.004**	4 **+1**
5 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	−0.008	6 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	7 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**
8 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	9 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	10 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**	11 **U: 0.0** **D: 0.0** **L: 0.0** **R: 0.0**

Q Learning & SARSA

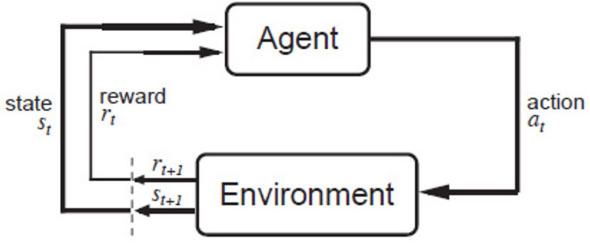
Converge to correct values

- Assuming agent tries all actions, in all states, many times
  - Otherwise, there won't be enough experience to learn *Q*
- And if α decays over time
  - Similar to simulated annealing

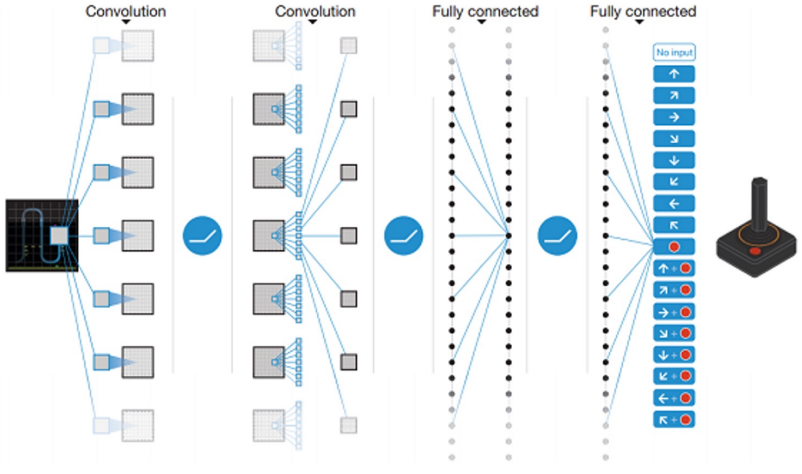
Avoids the complexity of solving an MDP

- MDP requires solving for the optimal policy before starting
- An RL agent can start right away and then learn as it goes
  - Although this might be wasteful if you already know *P* and *R*

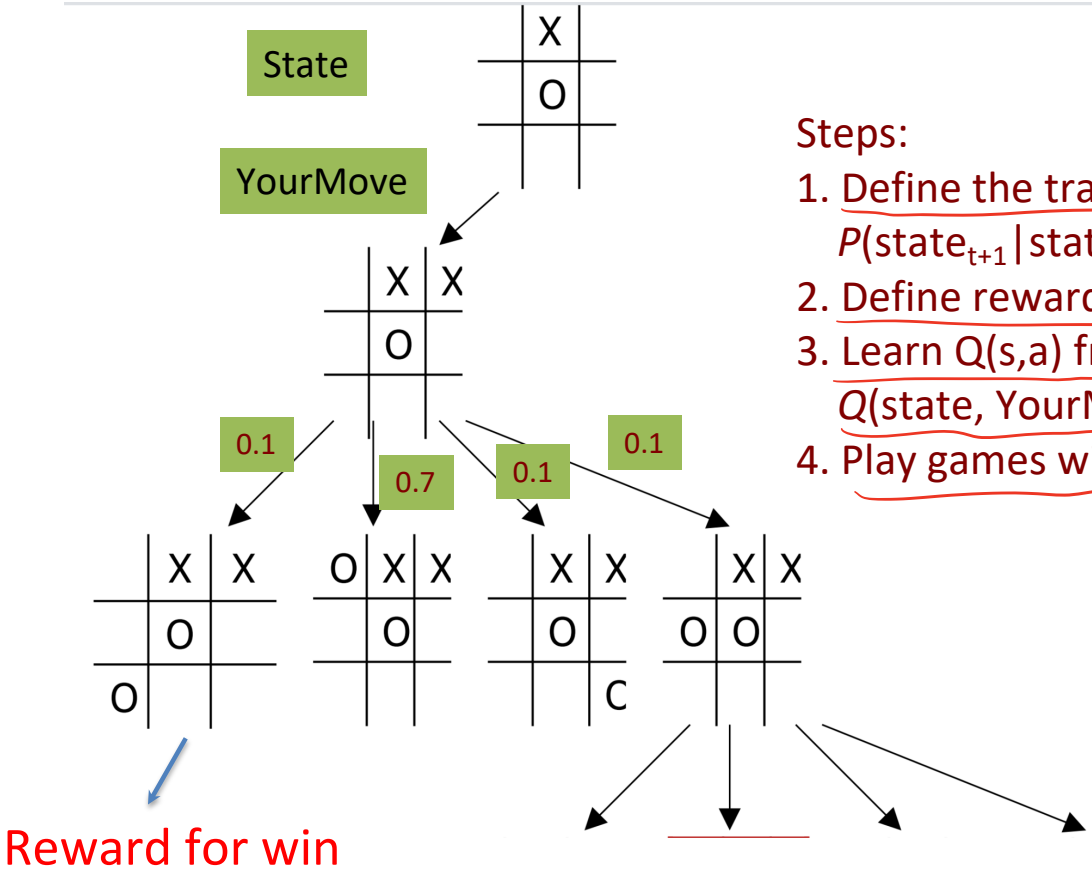
7. Q-Learning for Game Playing



AlphaGo



Q-Learning for Game Playing



Steps:

1. Define the transition model
   $P(state_{t+1} | state_t, YourMove)$
2. Define rewards
3. Learn Q(s,a) from rewards
   $Q(state, YourMove)$
4. Play games with learned Q

A Key Question

- In all search and game playing problems, what is the key information that you wish to have for finding the optimal solution?
  - You wish to have but you may not always have

- The answer: Play a lot of games and
  - Learn the state utility values for winning the games
  - Learn the Q values for winning the games

Exploitation vs. Exploration

- RL algorithms do not specify how actions are chosen by the agent
- One strategy would be in state *s* to select the action *a* that maximizes $Q(s,a)$, thereby *exploiting* its current approximation *Q*.
  - Using this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high *Q* values, while failing to *explore* other actions that have even higher values.
  - It is common in Q learning to use a probabilistic approach to selecting actions.
  - Actions with higher Q values are assigned higher probabilities, but every action is assigned a nonzero probability. One way to assign such probabilities is where $P(a_i \mid s)$ is the probability *T* of selecting action a_i, given that the agent is in state *s*, and where $T > 0$ is a constant that determines how strongly the selection favors actions with high *Q* values.

$$P(a_i \mid s) = \frac{e^{\hat{Q}(s,a_i)/T}}{\sum_j e^{\hat{Q}(s,a_j)/T}}$$

Exploitation vs. Exploration

- Hence it is good to try new things now and then
  - If *T* is large, then do more exploration,
  - If *T* is small, then follow or exploit the current policy
- One can decrease *T* over time to first explore, and then converge and exploit
  - For example *T= c/k+d* where *k* is iteration of the algorithm

$$P(a_i \mid s) = \frac{e^{\hat{Q}(s,a_i)/T}}{\sum_j e^{\hat{Q}(s,a_j)/T}}$$

- Decreasing *T* over time is also known as *simulated annealing*, which is inspired by annealing process in nature. *T* is also known as *Temperature*