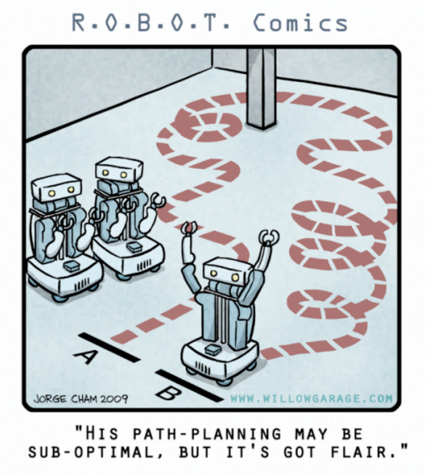
Sequential Decision Making Under Uncertainty

material from Marie desJardin, Lise Getoor, Jean-Claude Latombe, Daphne Koller, Stuart Russell, Dawn Song, Mark Hasegawa-Johnson, Svetlana Lazebnik, Pieter Abbeel, Dan Klein, Lisa Torrey



1

Bookkeeping

- HW5 out tonight, due 12/3
  - Planning
  - Sequential decision making
  - Reinforcement learning
- Today
  - Finding policies
  - Reinforcement learning
- Next class: (some) project work day
  - Bring computers

2

Review: The Big Idea

- "Planning": Find a sequence of steps to accomplish a goal.
  - Given start state, transition model, goal functions…
- This is a kind of **sequential decision making**.
  - Transitions are deterministic.
- What if they are stochastic (probabilistic)?
  - One time in ten, you drop your sock instead of putting it on
- **Probabilistic Planning:** Make a plan that accounts for probability by carrying it through the plan.

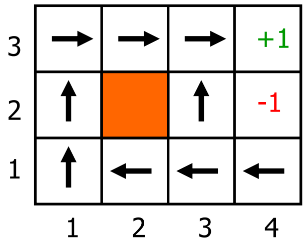
3

Review: Transition Model

- A transition model is a specification of the outcome probabilities for each action in each possible state.
- T(s,a,s') denotes the probability of reaching state s' if action a is done on state s.
- Make Markov Assumption, i.e., the probability of reaching state s' from s depends only on s and not on the history of earlier states.

4

Review: Policies



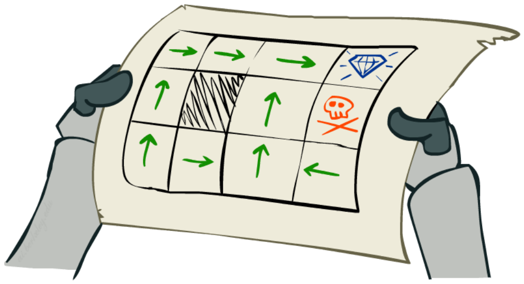
- In every state, we need to know what to do
- The **goal** doesn't change
- A policy (Π) is a complete mapping from *states* to *actions*
  - "If in [3,2], go up; if in [3,1], go left; if in…"

Review: Optimal Policy

- An *Optimal* policy is a policy that yields the highest expected utility.
- Optimal policy is denoted by π*.
- Once a π* is computed for a problem, then the agent, once identifying the state (s) that it is in, consults π*(s) for the next action to execute.

Review: Policies

- A policy π gives an action for each state, $\pi: S \rightarrow A$
- In deterministic single-agent search problems, we wanted an optimal ***plan***, or sequence of actions, from start to a goal
- For MDPs, we want an optimal ***policy*** $\pi^*: S \rightarrow A$
  - An optimal policy maximizes expected utility
  - An explicit policy defines a reflex agent



7

Computing the optimal policy π*

- Additive utility
- State utilities
- Action sequences
- The Bellman equation
- Value iteration
- Policy iteration

8

Additive Utility

- History H = $(s_0,s_1,\ldots,s_n)$
- The utility of H is additive iff:

$$\mathcal{U}(s_0,s_1,\ldots,s_n) = \mathcal{R}(0) + u(s_1,\ldots,s_n) = \sum \mathcal{R}(i)$$

Reward

- The reward accumulates as you step through states.

Additive Utility

- History H = $(s_0,s_1,\ldots,s_n)$
- The utility of H is additive iff:

$$\mathcal{U}(s_0,s_1,\ldots,s_n) = \mathcal{R}(0) + u(s_1,\ldots,s_n) = \sum \mathcal{R}(i)$$

- Robot navigation example:
  - $\mathcal{R}(n) = +1$ if $s_n = [4,3]$
  - $\mathcal{R}(n) = -1$ if $s_n = [4,2]$
  - $\mathcal{R}(i) = -1/25$ if $i = 0, \ldots, n-1$

→	→	→	+1
↑	🟧	↑	-1
↑	←	←	←

Defining the optimal policy

- Given a policy π, we can define the **expected utility** over all possible state sequences produced by following that policy:

$$U^{\pi}(s_0) = \sum_{\substack{\text{state sequences} \\ \text{starting from } s_0}} P(\text{sequence})U(\text{sequence})$$

- The optimal policy should maximize this utility
- **But how to define the utility of a state sequence?**
  - Sum of rewards of individual states
  - Problem: infinite state sequences

11

Utilities of state sequences

- Normally, we would define the utility of a state sequence as the sum of the rewards of the individual states
- **Problem**: infinite state sequences
- **Solution**: discount the individual state rewards by a factor γ between 0 and 1:

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$

$$= \sum_{t=0}^{\infty} \gamma^t R(s_t) \le \frac{R_{\max}}{1-\gamma} \qquad (0 < \gamma < 1)$$

- Sooner rewards “count” more than later rewards
- Makes sure the total utility stays bounded
- Helps algorithms converge

12

Sum of discounted rewards

- To define the utility of a state sequence, discount the individual state rewards by a factor γ between 0 and 1:

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$

- When γ = 1 this is just additive utility

13

Utilities of states

- Expected utility obtained by policy π starting in state s:

$$U^{\pi}(s) = \sum_{\substack{state\ sequences \\ starting\ from\ s}} P\big(sequence|s, a = \pi(s)\big)U(sequence)$$

- The "true" utility of a state, denoted U(s), is the **best possible** expected sum of discounted rewards
  - if the agent executes the **best possible policy** starting in state s
- Reminiscent of minimax values of states

14

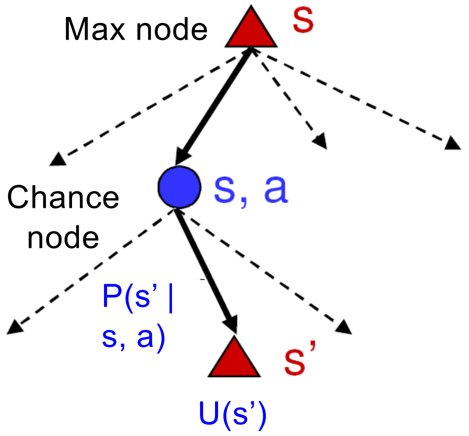
Defining State Utility

Problem:

- When making a decision, we only know the reward so far, and the possible actions
- We've defined utility retroactively (i.e., the utility of a history is known *once we finish it*)
- What is the **utility** of a **particular state** in the middle of decision making?
- Need to compute **expected utility** of possible future histories

17

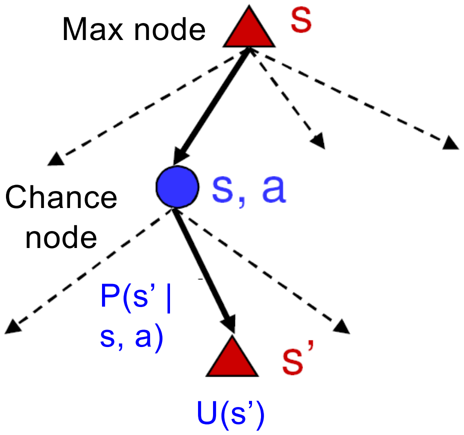
Finding the utilities of states



- If state s' has utility U(s'), then what is the expected utility of taking action **a** in state **s**?
- How do we choose the optimal action?
- What is the recursive expression for **U(s)** in terms of the utilities of its successor states?

18

Finding the utilities of states



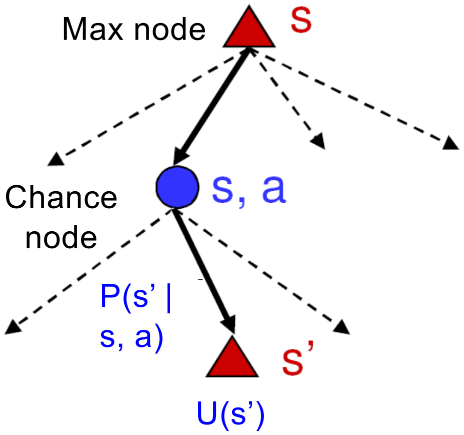
- If state s' has utility U(s'), then what is the expected utility of taking action **a** in state **s**?

$$\sum_{s'} P(s' | s, a) U(s')$$

- How do we choose the optimal action?

- What is the recursive expression for **U(s)** in terms of the utilities of its successor states?

Finding the utilities of states



- If state s' has utility U(s'), then what is the expected utility of taking action **a** in state **s**?

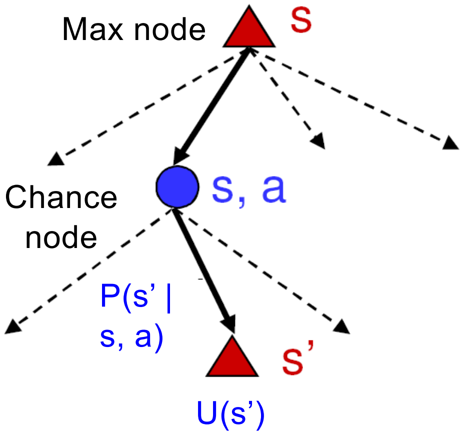
$$\sum_{s'} P(s' | s, a) U(s')$$

- How do we choose the optimal action?

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')$$

- What is the recursive expression for **U(s)** in terms of the utilities of its successor states?

Finding the utilities of states


Max node

s

Chance node

s, a

P(s' | s, a)

s'

U(s')

- If state s' has utility U(s'), then what is the expected utility of taking action **a** in state **s**?

$$\sum_{s'} P(s' \mid s, a) U(s')$$

- How do we choose the optimal action?

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')$$

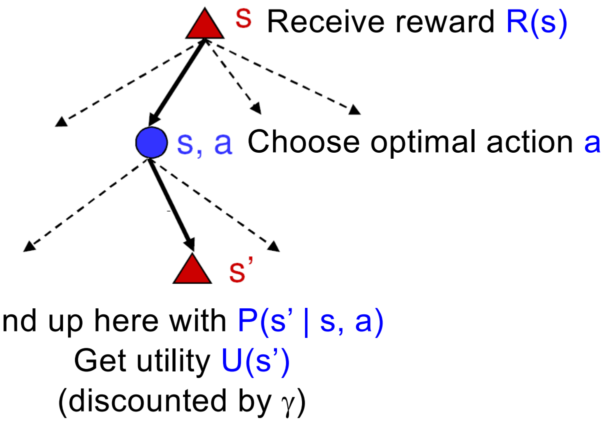
- What is the recursive expression for **U(s)** in terms of the utilities of its successor states?

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s' \mid s, a) U(s')$$

The Bellman equation

- Recursive relationship between the utilities of successive states:

$$\boxed{U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')}$$


s Receive reward R(s)

s, a Choose optimal action a

s'

End up here with P(s' | s, a)
Get utility U(s')
(discounted by γ)

The Bellman equation

- Recursive relationship between the utilities of successive states:

$$\boxed{U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')}$$

- For N states, we get N equations in N unknowns
  - Solving them solves the MDP
  - The “max” means that there is no closed-form solution. Need to use an iterative solution method, which might not converge to the globally optimum solution.
  - Two solution methods: value iteration and policy iteration

23

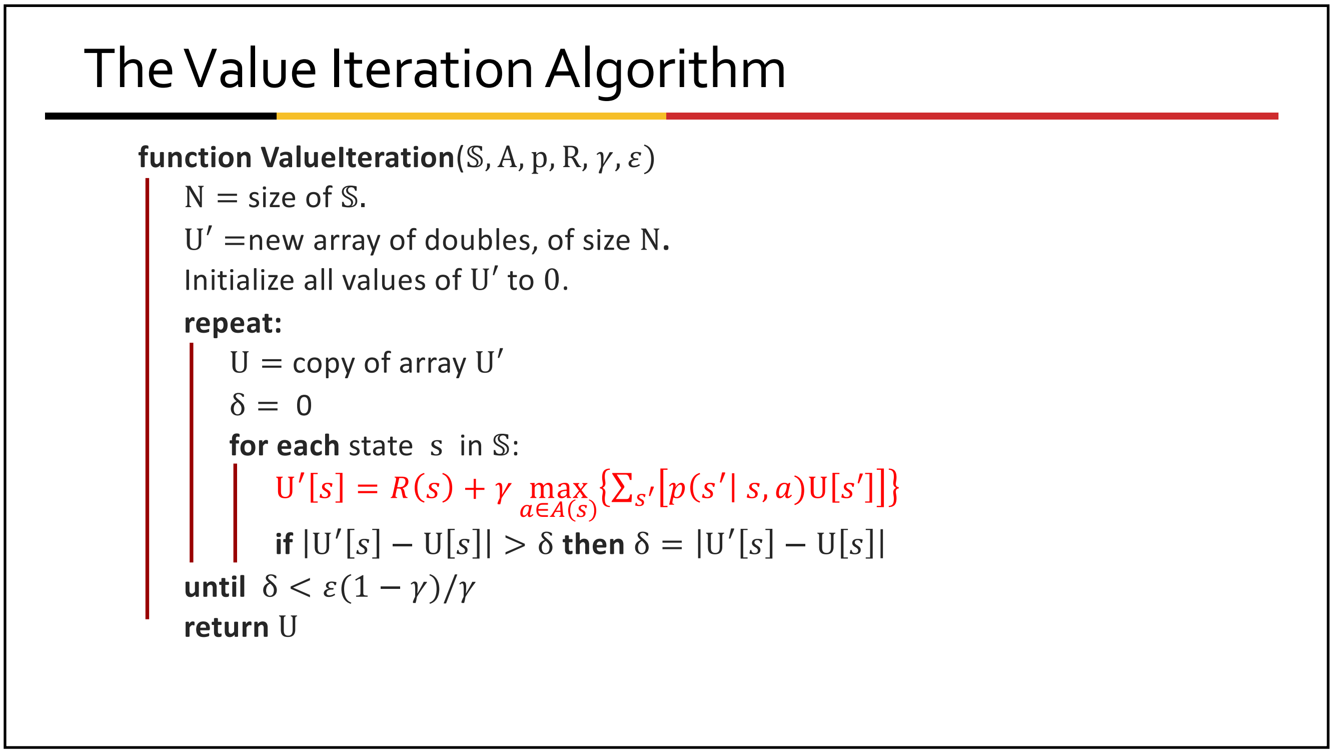
Method 1: Value iteration

- Start out with iteration $i = 0$, every $U_i\ (s)\ =\ 0$
- Iterate until convergence
  - During the *i*th iteration, update the utility of each state according to this rule:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U_i(s')$$

- So we’re looking at utility of each state based on its successors
- In the limit of infinitely many iterations, guaranteed to find the correct utility values.
  - Error decreases exponentially, so in practice, don’t need infinite iterations

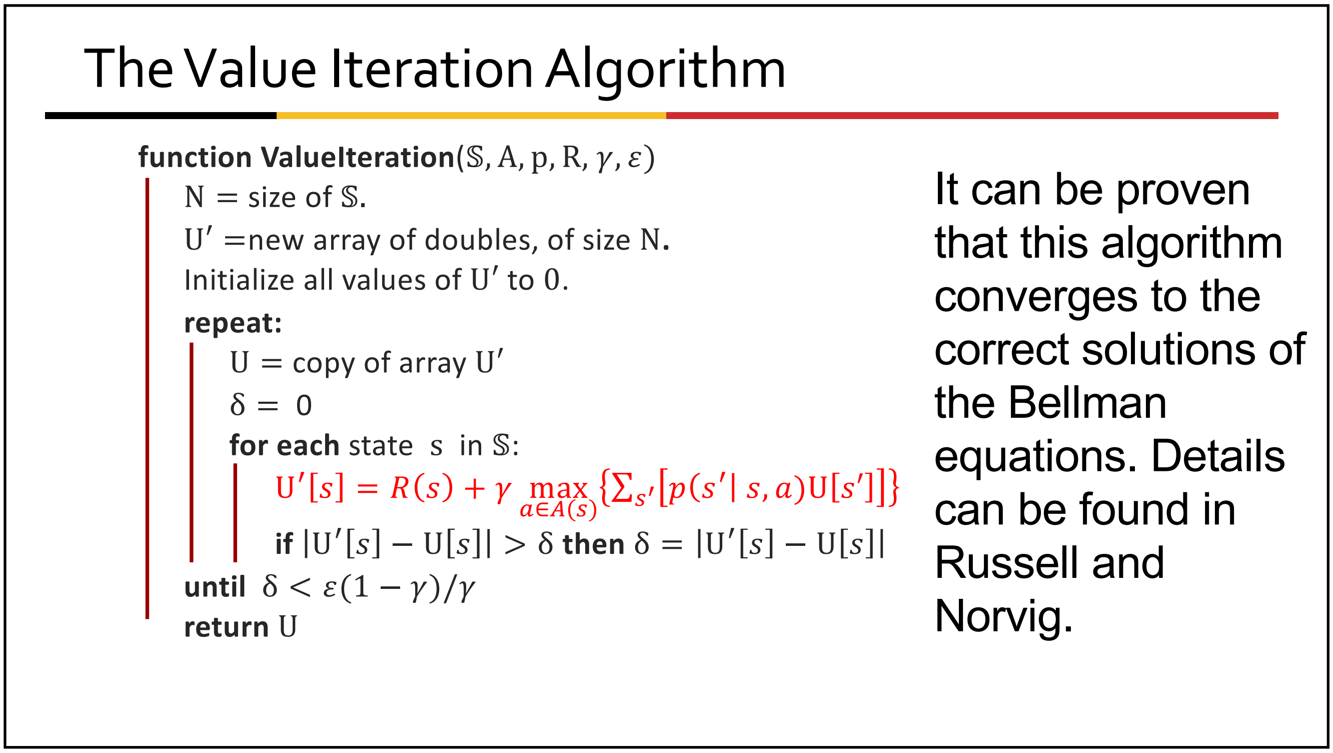
24



The Value Iteration Algorithm

```
function ValueIteration($\mathbb{S}, \mathrm{A}, \mathrm{p}, \mathrm{R}, \gamma, \varepsilon$)
    $\mathrm{N}$ = size of $\mathbb{S}$.
    $\mathrm{U}'$ =new array of doubles, of size $\mathrm{N}$.
    Initialize all values of $\mathrm{U}'$ to $0$.
    repeat:
        $\mathrm{U}$ = copy of array $\mathrm{U}'$
        $\delta = 0$
        for each state $\mathrm{s}$ in $\mathbb{S}$:
            $\mathrm{U}'[s] = R(s) + \gamma \max_{a \in A(s)} \{\sum_{s'} [p(s' | s, a) \mathrm{U}[s']]\}$
            if $|\mathrm{U}'[s] - \mathrm{U}[s]| > \delta$ then $\delta = |\mathrm{U}'[s] - \mathrm{U}[s]|$
    until $\delta < \varepsilon(1 - \gamma)/\gamma$
    return $\mathrm{U}$
```

25

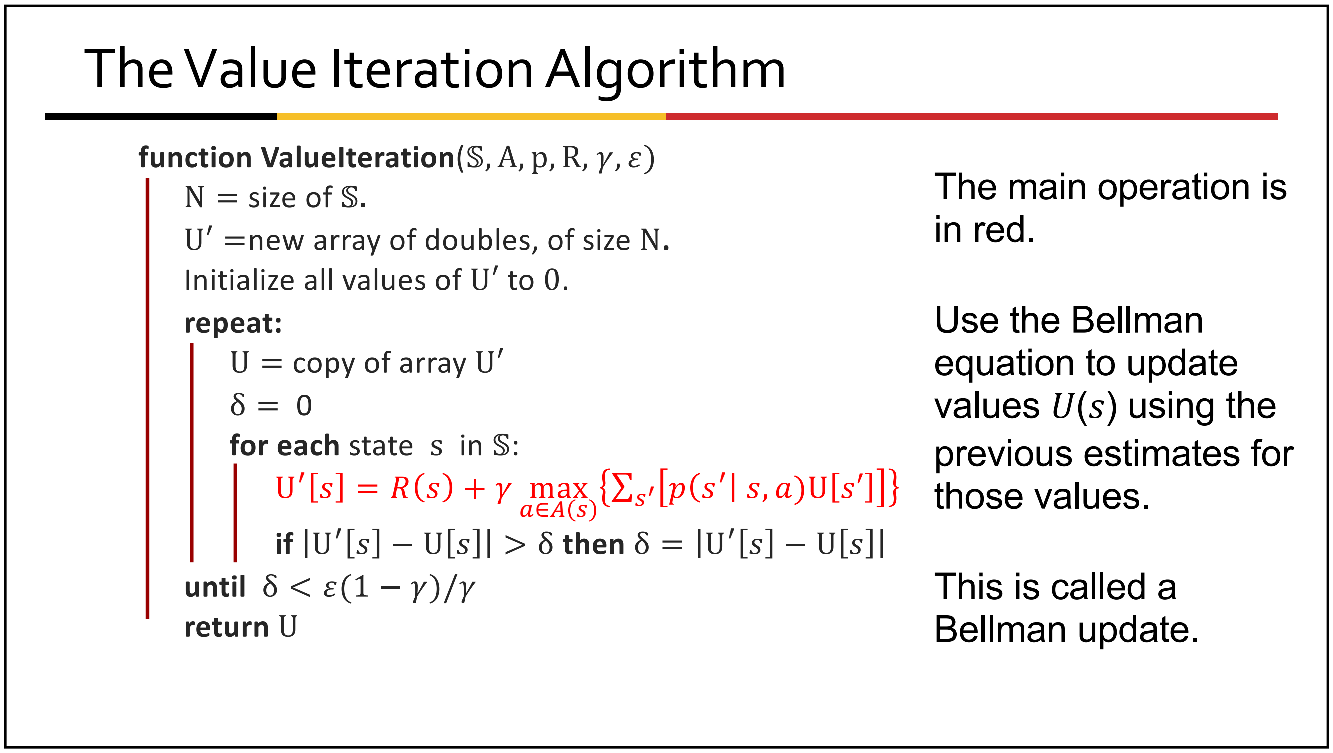


The Value Iteration Algorithm

```
function ValueIteration($\mathbb{S}, \mathrm{A}, \mathrm{p}, \mathrm{R}, \gamma, \varepsilon$)
    $\mathrm{N}$ = size of $\mathbb{S}$.
    $\mathrm{U}'$ =new array of doubles, of size $\mathrm{N}$.
    Initialize all values of $\mathrm{U}'$ to $0$.
    repeat:
        $\mathrm{U}$ = copy of array $\mathrm{U}'$
        $\delta = 0$
        for each state $\mathrm{s}$ in $\mathbb{S}$:
            $\mathrm{U}'[s] = R(s) + \gamma \max_{a \in A(s)} \{\sum_{s'} [p(s' | s, a) \mathrm{U}[s']]\}$
            if $|\mathrm{U}'[s] - \mathrm{U}[s]| > \delta$ then $\delta = |\mathrm{U}'[s] - \mathrm{U}[s]|$
    until $\delta < \varepsilon(1 - \gamma)/\gamma$
    return $\mathrm{U}$
```

It can be proven that this algorithm converges to the correct solutions of the Bellman equations. Details can be found in Russell and Norvig.

26



The Value Iteration Algorithm

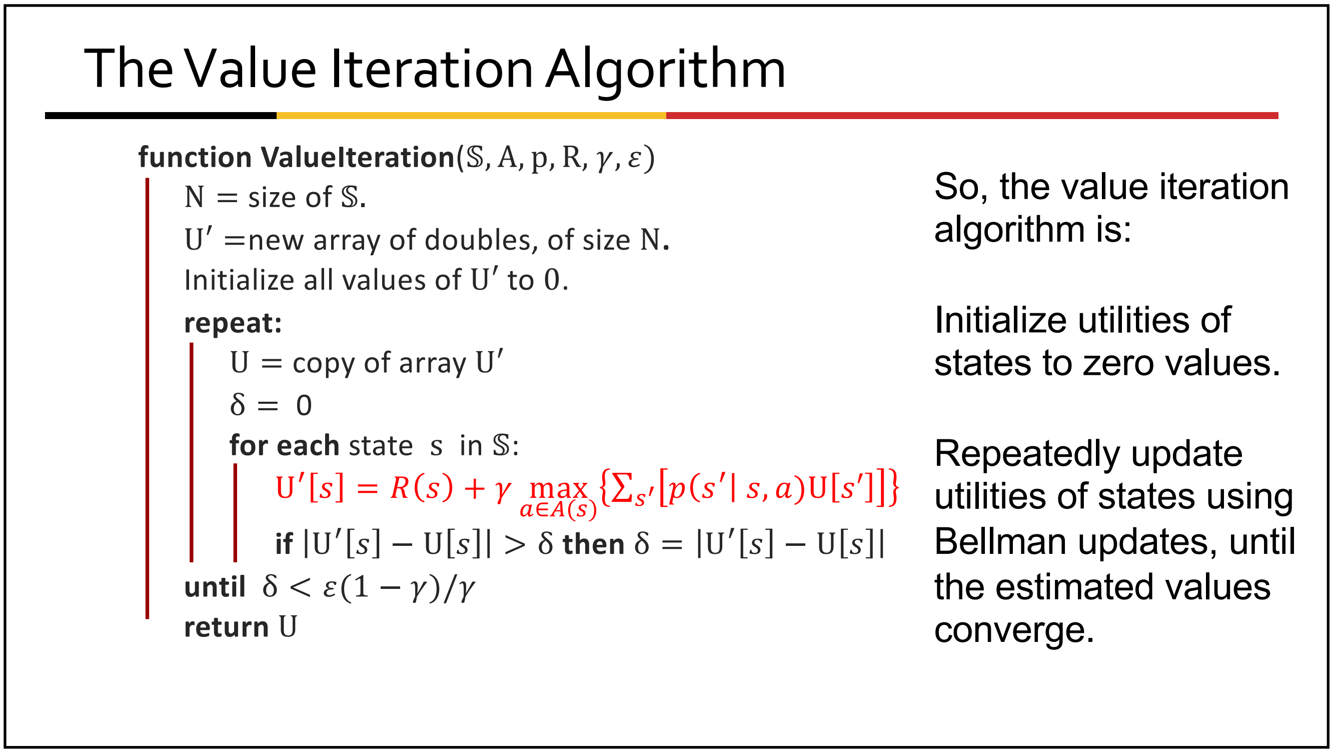
```
function ValueIteration($\mathbb{S}, \mathrm{A}, \mathrm{p}, \mathrm{R}, \gamma, \varepsilon$)
    $\mathrm{N}$ = size of $\mathbb{S}$.
    $\mathrm{U}'$ =new array of doubles, of size $\mathrm{N}$.
    Initialize all values of $\mathrm{U}'$ to $0$.
    repeat:
        $\mathrm{U}$ = copy of array $\mathrm{U}'$
        $\delta = 0$
        for each state $\mathrm{s}$ in $\mathbb{S}$:
            $\mathrm{U}'[s] = R(s) + \gamma \max_{a \in A(s)} \{\sum_{s'} [p(s' | s, a) \mathrm{U}[s']]\}$
            if $|\mathrm{U}'[s] - \mathrm{U}[s]| > \delta$ then $\delta = |\mathrm{U}'[s] - \mathrm{U}[s]|$
    until $\delta < \varepsilon(1-\gamma)/\gamma$
    return $\mathrm{U}$
```

The main operation is in red.

Use the Bellman equation to update values $U(s)$ using the previous estimates for those values.

This is called a Bellman update.

27



The Value Iteration Algorithm

```
function ValueIteration($\mathbb{S}, \mathrm{A}, \mathrm{p}, \mathrm{R}, \gamma, \varepsilon$)
    $\mathrm{N}$ = size of $\mathbb{S}$.
    $\mathrm{U}'$ =new array of doubles, of size $\mathrm{N}$.
    Initialize all values of $\mathrm{U}'$ to $0$.
    repeat:
        $\mathrm{U}$ = copy of array $\mathrm{U}'$
        $\delta = 0$
        for each state $\mathrm{s}$ in $\mathbb{S}$:
            $\mathrm{U}'[s] = R(s) + \gamma \max_{a \in A(s)} \{\sum_{s'} [p(s' | s, a) \mathrm{U}[s']]\}$
            if $|\mathrm{U}'[s] - \mathrm{U}[s]| > \delta$ then $\delta = |\mathrm{U}'[s] - \mathrm{U}[s]|$
    until $\delta < \varepsilon(1-\gamma)/\gamma$
    return $\mathrm{U}$
```

So, the value iteration algorithm is:

Initialize utilities of states to zero values.

Repeatedly update utilities of states using Bellman updates, until the estimated values converge.

28

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- We initialize all utility values to 0.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	0	0	0	0
2	0	▨	0	0
1	0	0	0	0
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after one round of updates:
  - The current estimate for each state s is $R(s)$.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	-0.04	-0.04	-0.04	+1
2	-0.04	▨	-0.04	-1
1	-0.04	-0.04	-0.04	-0.04
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after two rounds of updates:
  - Information about the +1 reward reached state (3,3).

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	-0.08	-0.08	0.67	+1
2	-0.08	▨	-0.08	-1
1	-0.08	-0.08	-0.08	-0.08
	1	2	3	4

Utility Values

31

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after three rounds of updates:
  - Information about the +1 reward reached more states.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	-0.11	0.43	0.73	+1
2	-0.11	▨	0.35	-1
1	-0.11	-0.11	-0.11	-0.11
	1	2	3	4

Utility Values

32

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after four rounds of updates:
  - Information about the +1 reward reached more states.

3				+1
2				-1
1	START			
	1	2	3	4

3	0.25	0.57	0.78	+1
2	-0.14		0.43	-1
1	-0.14	-0.14	0.19	-0.14
	1	2	3	4

Utility Values

33

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after five rounds of updates:
  - Information about the +1 reward reached more states.

3				+1
2				-1
1	START			
	1	2	3	4

3	0.38	0.62	0.79	+1
2	0.12		0.47	-1
1	-0.16	0.07	0.24	-0.01
	1	2	3	4

Utility Values

34

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after six rounds of updates:
  - Information about the +1 reward has reached all states.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	0.45	0.64	0.79	+1
2	0.25	▨	0.48	-1
1	0.04	0.15	0.30	0.05
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after seven rounds of updates:
  - Values keep getting updated.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	0.48	0.65	0.79	+1
2	0.33	▨	0.48	-1
1	0.16	0.21	0.32	0.09
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after eight rounds of updates:
  - Values continue changing.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	0.50	0.65	0.80	+1
2	0.37	▨	0.49	-1
1	0.23	0.23	0.34	0.11
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after 13 rounds of updates:
  - Values don't change much anymore after this round.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	0.51	0.65	0.80	+1
2	0.40	▨	0.49	-1
1	0.30	0.25	0.34	0.13
	1	2	3	4

Utility Values

Computing the Optimal Policy

- The value iteration algorithm computes $\mathrm{U}(s)$ for every state s.
- Once we have computed all values $\mathrm{U}(s)$, we can get the optimal policy π^* using this equation:
- $\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}}\{\sum_{s\prime}[p(s' \mid s, a)U(s')]\}$
- Thus, $\pi^*(s)$ identifies the action that leads to the highest expected utility for the next state, as measured over all possible outcomes of that action.
- This approach is called one-step look-ahead.

39

Approach 2: Policy Iteration

- There is a more efficient algorithm for computing optimal policies
- Remember that, if we know the utility of each state, we can compute the optimal policy π^* using:

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}}\left\{\sum_{s\prime}[p(s' \mid s, a)U(s')]\right\}$$

- However, to get the right $\pi^*(s)$, we don't need to know the utilities very accurately.
- We just need to know the utilities accurately enough so that, for each state s, argmax chooses the right action.

40

Method 2: Policy Iteration

- Start with some initial policy π_0 and alternate between the following steps:
  - **Policy Evaluation:** calculate the utility of every state under the assumption that the given policy is fixed and unchanging.
  - **Policy Improvement:** calculate a new policy π_{i+1} based on the updated utilities.
- Kind of like gradient descent in neural networks:
  - Policy evaluation: Find ways in which the current policy is suboptimal
  - Policy improvement: Fix those problems
- Unlike Value Iteration, this is guaranteed to converge in a finite number of steps, as long as the state space and action set are both finite.

41

The Policy Iteration Algorithm

- This alternative algorithm for computing optimal policies is called the **policy iteration algorithm**.
- It is an iterative algorithm.
- Initialization:
  - Initiate some policy π_0 with random choices for the best action at each state.
- Main loop:
  - **Policy evaluation**: given the current policy π_i, calculate utility values $U^{\pi_i}(s)$, corresponding to the utility of each state s if the agent follows policy $\boldsymbol{\pi_i}$.
  - **Policy improvement**: Given current utility values $U^{\pi_i}(s)$, use one-step look-ahead to compute new policy π_{i+1}.

42

The Policy Evaluation Step

- Task: calculate utility values $U^{\pi_i}(\mathrm{s})$, corresponding to the assumption that the agent follows policy $\boldsymbol{\pi_i}$.
- When the policy was not known, we used the Bellman equation:

$$\mathrm{U}(s) = R(s) + \gamma \max_{a \in A(s)} \left\{ \sum_{s'} [p(s' | \, s, a) U(s')] \right\}$$

- Now that the policy π_i is specified, we can instead use a simplified version of the Bellman equation:

$$U^{\pi_i}(s) = R(s) + \gamma \sum_{s'} [p(s' | \, s, \pi_i(s)) U^{\pi_i}(s')]$$

- Key difference: now $\pi_i(s)$ specifies the action for each state s, so we do not need to look for the max over all possible actions.

43

The Policy Evaluation Step

- $U^{\pi_i}(s) = R(s) + \gamma \sum_{s'} [p(s' | \, s, \pi_i(s)) U^{\pi_i}(s')]$
- This is a linear equation.
  - The original Bellman equation, taking the max out of all possible actions, is not linear.
- If we have N states, we get N linear equations of this form, with N unknowns.
- We can solve those N linear equations in $\mathrm{O}(N^3)$ time, using standard linear algebra methods.

44

The Policy Evaluation Step

- For large state spaces, $\mathrm{O}(N^3)$ is prohibitive.
- Alternative: do some rounds of iterations.

```
function PolicyEvaluation($\mathbb{S}, \mathrm{p}, \mathrm{R}, \gamma, \pi_i, K, \mathrm{U}$)
    $\mathrm{U}_0$ = copy of $\mathrm{U}$
    for $\boldsymbol{k} = 1$ to $\boldsymbol{K}$:
        for each state $s$ in $\mathbb{S}$:
            $U_k(s) = R(s) + \gamma \sum_{s'} [p(s' | s, \pi_i(s)) U_{k-1}(s')]$
    return $U_k$
```

- Obviously, doing K iterations does not guarantee that the utilities are computed correctly.
- Parameter K allows us to trade speed for accuracy. Larger values lead to slower runtimes and higher accuracy.

45

The Policy Evaluation Step

- For large state spaces, $\mathrm{O}(N^3)$ is prohibitive.
- Alternative: do some rounds of iterations.

```
function PolicyEvaluation($\mathbb{S}, \mathrm{p}, \mathrm{R}, \gamma, \pi_i, K, \mathrm{U}$)
    $\mathrm{U}_0$ = copy of $\mathrm{U}$
    for $\boldsymbol{k} = 1$ to $\boldsymbol{K}$:
        for each state $s$ in $\mathbb{S}$:
            $U_k(s) = R(s) + \gamma \sum_{s'} [p(s' | s, \pi_i(s)) U_{k-1}(s')]$
    return $U_k$
```

- The PolicyEvaluation function takes as argument a current estimate U.

46

Policy Iteration

- Pick a policy π at random
- Repeat:
  - Compute the utility of each state for π

    $\mathcal{U}_{t+1}(i) \leftarrow \mathcal{R}(i) + \sum_k \mathcal{P}(k \mid \pi(i).i)\ \mathcal{U}_t(k)$
  - Compute the policy π’ given these utilities

    $\pi'(i) = \arg\max_a \sum_k \mathcal{P}(k \mid a.i)\ \mathcal{U}(k)$
  - If π’ = π then return π

Policy Iteration: Convergence

- Convergence assured in a finite number of iterations
  - Since finite number of policies and each step improves value, then must converge to optimal
- Gives exact value of optimal policy

Policy Iteration Complexity

- Each iteration runs in polynomial time in the number of states and actions
- There are at most |A|n policies and PI never repeats a policy
  - So at most an exponential number of iterations
  - Not a very good complexity bound
- Empirically O(n) iterations are required – often it seems like O(1)
- Recent polynomial bounds.

49

Value Iteration: Summary

- Value iteration:
  - Initialize state values (expected utilities) randomly
  - Repeatedly update state values using best action, according to current approximation of state values
  - Terminate when state values stabilize
  - Resulting policy will be the best policy because it's based on accurate state value estimation

50

Policy Iteration: Summary

- Policy iteration:
  - Initialize policy randomly
  - Repeatedly update state values using best action, according to current approximation of state values
  - Then update policy based on new state values
  - Terminate when policy stabilizes
  - Resulting policy is the best policy, but state values may not be accurate (may not have converged yet)
  - Policy iteration is often faster (because we don’t have to get the state values right)

Value Iteration vs. Policy Iteration

- Which is faster? VI or PI
  - It depends on the problem
- VI takes more iterations than PI, but PI requires more time on each iteration
  - PI must perform policy evaluation on each iteration which involves solving a linear system
- VI is easier to implement since it does not require the policy evaluation step
- Both methods have a major weakness: They require us to know the transition function exactly in advance!

Reinforcement Learning: Overview

- Machine Learning: A quick retrospective
- Reinforcement Learning
- Next time:
  - The EM algorithm
  - Monte Carlo and Temporal Difference

53

Review: What is ML?

- ML is a way to get a computer to do things without having to explicitly describe what steps to take.
- By giving it **examples** (training data)
- Or by giving it **feedback**
- It can then look for patterns which explain or predict what happens.
- The learned system of beliefs is called a **model**.

54

Review: Architecture of an ML System

- Every machine learning system has four parts:
  - A **representation or model** of what is being learned.
  - An **actor**: Uses the representation and actually does something.
  - A **critic**: Provides feedback.
  - A **learner**: Modifies the representation / model, using the feedback.

55

Review: Representation

- A learning system must have a **representation or model** of what is being learned.
- This is what changes based on experience.
- In a machine learning system this may be:
  - A mathematical model or formula
  - A set of rules
  - A decision tree
  - A policy
  - Or some other form of information

57

Review: Formalizing Agents

- Given:
  - A state space S
  - A set of actions a_1, …, a_k including their results
  - Reward value at the **end of each trial** (series of action) (may be positive or negative)
- Output:
  - A **mapping from states to actions**
  - Which is a **policy**, π

58

Learning Without a Model

- We saw how to learn a value function and/or a policy from a transition model
- What if we don’t have a transition model?
- Idea #1: Build one
  - Explore the environment for a long time
  - Record all transitions
  - Learn the transition model
  - Apply value iteration/policy iteration
  - Slow, requires a lot of exploration, no intermediate learning
- Idea #2: Learn a value function (or policy) directly from interactions with the environment, while exploring

59

Reinforcement Learning

- We often have an agent which has a **task** to perform
  - It takes some actions in the world
  - At some later point, gets feedback on how well it did
  - The agent performs the same task repeatedly
- This problem is called **reinforcement learning**:
  - The agent gets positive reinforcement for tasks done well
  - And gets negative reinforcement for tasks done poorly
  - Must somehow figure out which actions to take next time

Reinforcement Learning (RL)

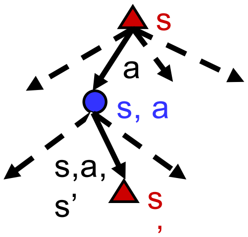
- RL algorithms attempt to find a policy
  - Maximizing cumulative reward for the agent over the course of the problem
- Typically represented by a **Markov Decision Process**
- RL differs from supervised learning:
  - Correct input/output pairs are never presented
  - Sub-optimal actions never explicitly corrected

Typical Applications

- Robotics
  - Helicopter control
  - Robo-soccer
- Board games
  - Checkers
  - Backgammon
  - Go/Atari
- Scheduling
  - Dynamic channel allocation
  - Inventory problems

Recap: Defining MDPs

- Markov decision processes:
  - States S
  - Start state s_0
  - Actions A
  - Transitions P(s'|s,a) (or T(s,a,s'))
  - Rewards R(s,a,s') (and discount γ)



- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility (or return) = sum of discounted rewards

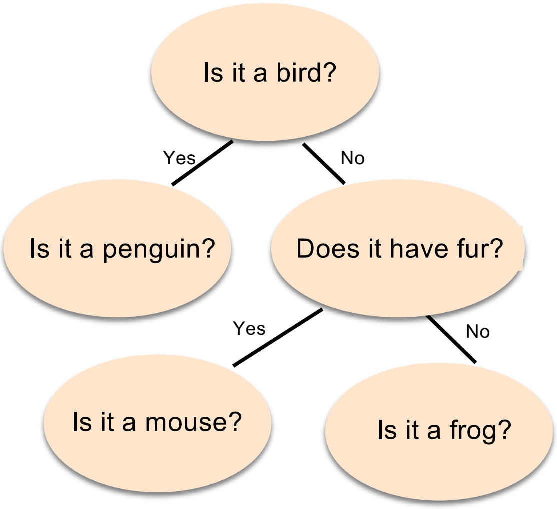
Animals Game

- Human: I’m thinking of an animal.
- Computer: Is it a bird?
- Human: No.
- Computer: Is it a frog?
- Human: No.
- Computer: What is it?
- Human: A mouse.
- Computer: **Tell me a question that distinguishes a frog from a mouse.**
- Human: Does it have fur?
- Computer: What is the answer for a frog?
- Human: No.

64

Animals Behind the Scene

Computer: Is it a bird?
Human: No
Computer: Is it a frog?
Human: No
Computer: What is it?
Human: A mouse
Computer: Tell me a question that distinguishes a frog from a mouse.
Human: Does it have fur?
Computer: What is the answer for a frog?
Human: no



Is it a bird?
Yes
No
Is it a penguin?
Does it have fur?
Yes
No
Is it a mouse?
Is it a frog?

After several rounds…

65

Animals Guessing Game Architecture

- All of the parts of ML Architecture:
  - The Representation is a sequence of questions and pairs of yes/no answers (called a binary decision tree).
  - The Actor "walks" the tree, interacting with a human; at each question it chooses whether to follow the "yes" branch or the "no" branch.
  - The Critic is the human player telling the game whether it has guessed correctly.
  - The Learner elicits new questions and adds questions, guesses and branches to the tree.

66

Reinforcement Learning

- This is a simple form of **Reinforcement Learning**
- Feedback is at the end, on a **series** of actions.
- Very early concept in Artificial Intelligence!
- Arthur Samuels' checker program was a simple reinforcement based learner, initially developed in 1956.
- In 1962 it beat a human checkers master.

www-03.ibm.com/ibm/history/ibm100/us/en/icons/ibm700series/impacts/

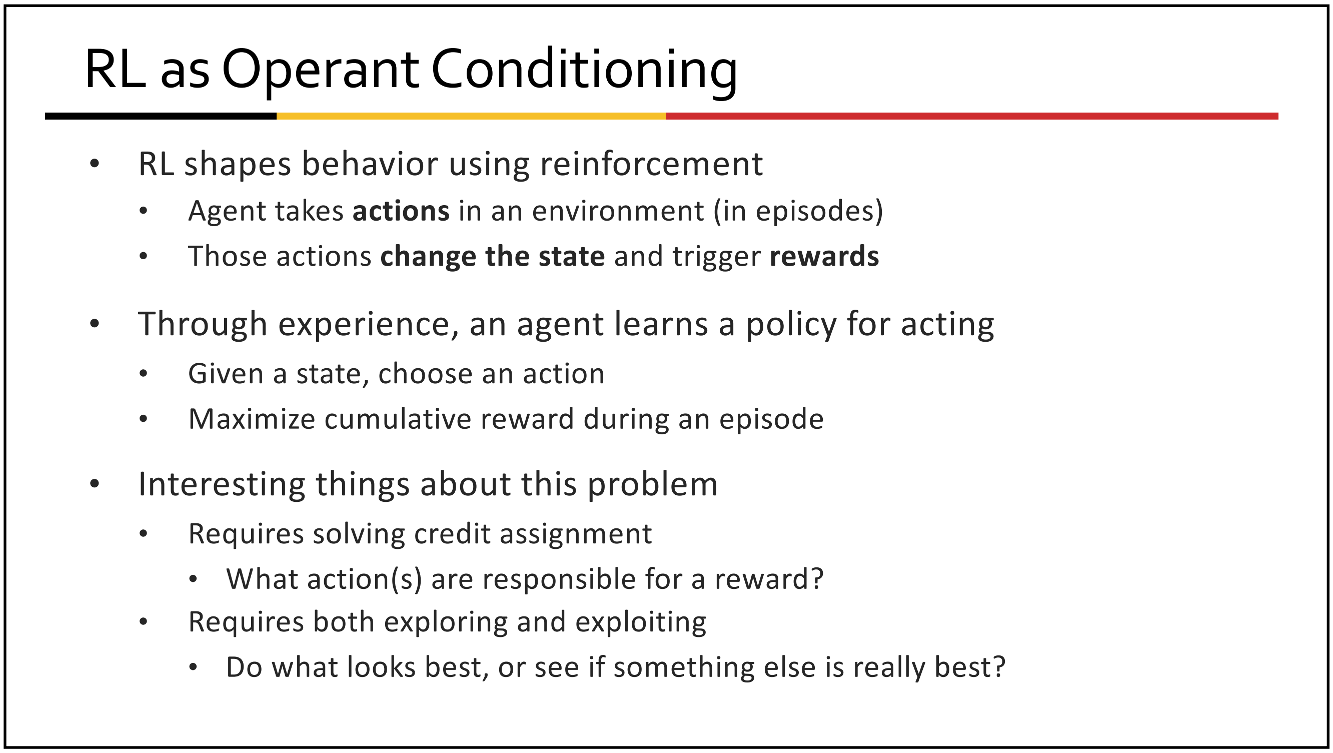
67

Reinforcement Learning (cont.)

- Goal: agent acts in the world to maximize its rewards
- Agent has to figure out what it did that made it get that reward/punishment
  - This is known as the credit assignment problem
- RL can be used to train computers to do many tasks
  - Backgammon and chess playing
  - Job shop scheduling
  - Controlling robot limbs

Procedural Learning

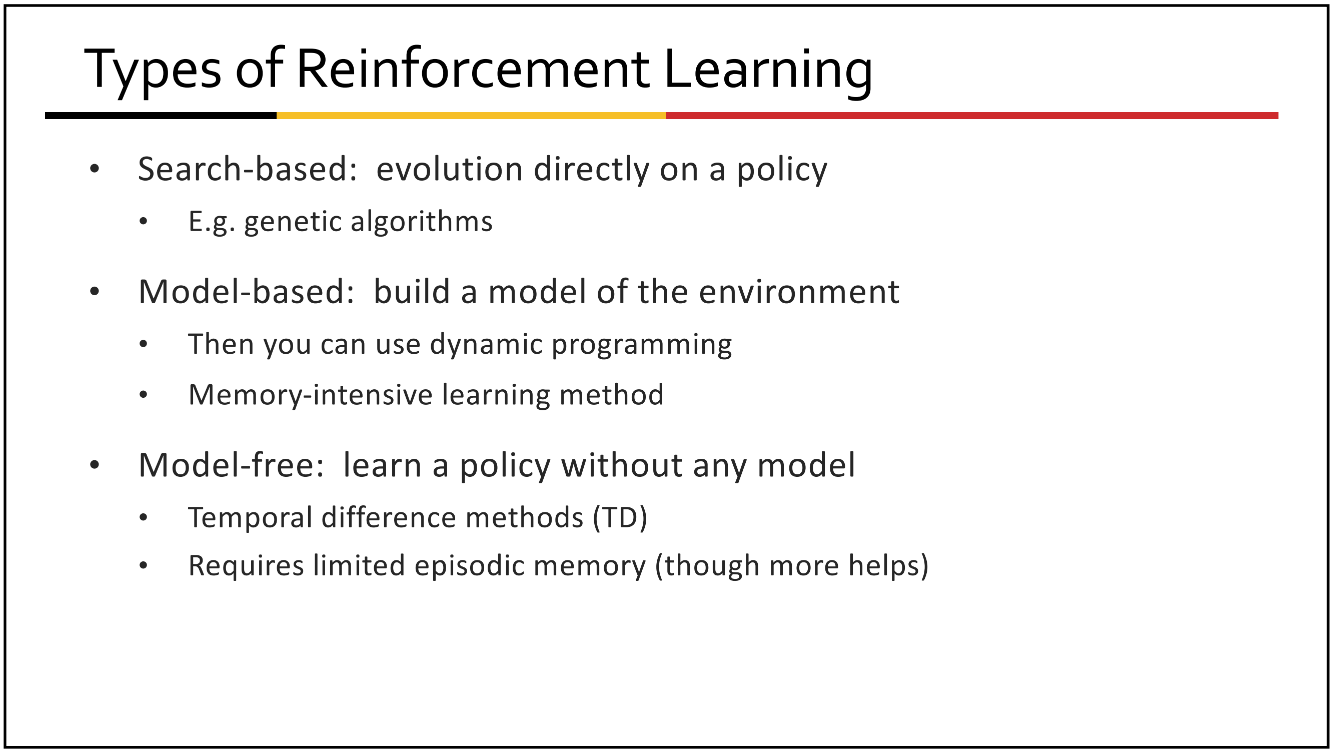
- Learning how to act to accomplish goals
  - **Given**: Environment that contains rewards
  - **Learn**: A policy for acting
- Important differences from classification
  - You don’t get examples of correct answers
  - You have to try things in order to learn



RL as Operant Conditioning

- RL shapes behavior using reinforcement
  - Agent takes **actions** in an environment (in episodes)
  - Those actions **change the state** and trigger **rewards**
- Through experience, an agent learns a policy for acting
  - Given a state, choose an action
  - Maximize cumulative reward during an episode
- Interesting things about this problem
  - Requires solving credit assignment
    - What action(s) are responsible for a reward?
  - Requires both exploring and exploiting
    - Do what looks best, or see if something else is really best?

70



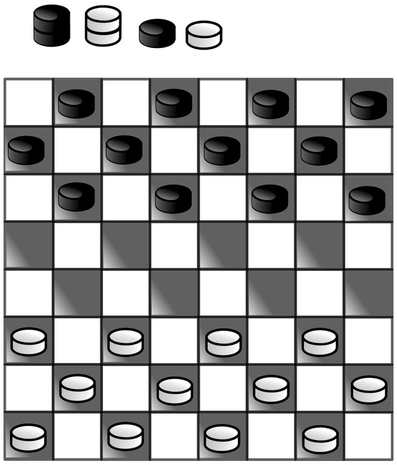
Types of Reinforcement Learning

- Search-based: evolution directly on a policy
  - E.g. genetic algorithms
- Model-based: build a model of the environment
  - Then you can use dynamic programming
  - Memory-intensive learning method
- Model-free: learn a policy without any model
  - Temporal difference methods (TD)
  - Requires limited episodic memory (though more helps)

71

Simple Example

- Learn to play checkers
  - Two-person game
  - 8x8 boards, 12 checkers/side
  - relatively simple set of rules: http://www.darkfish.com/checkers/rules.html
  - Goal is to eliminate all your opponent's pieces



https://pixabay.com/en/checker-board-black-game-pattern-29911

72

Representing Checkers

- First we need to represent the game
- To completely describe one step in the game you need
  - A representation of the game board.
  - A representation of the current pieces
  - A variable which indicates whose turn it is
  - A variable which tells you which side is "black"
- There is no history needed
- A look at the current board setup gives you a complete picture of the state of the game

which makes it a ____ problem?

73

Representing Rules

- Second, we need to represent the rules
- Represented as a **set of allowable moves** given board state
  - If a checker is at row x, column y, and row x+1 column y±1 is empty, it can move there.
  - If a checker is at (x,y), a checker of the opposite color is at (x+1, y+1), and (x+2,y+2) is empty, the checker must move there, and remove the “jumped” checker from play
- There are additional rules, but all can be expressed in terms of the state of the board and the checkers
- Each rule includes the outcome of the relevant action in terms of the state

74

What Do We Want to Learn?

- Given
  - A description of some state of the game
  - A list of the moves allowed by the rules
  - **What move should we make?**
- Typically more than one move is possible
  - Need strategies, heuristics, or hints about what move to make
  - **This is what we are learning**
- We learn **from** whether the game was won or lost
  - Information to learn from is sometimes called “training signal”

77

Simple Checkers Learning

- Can represent some heuristics in the same formalism as the board and rules
  - If there is a legal move that will create a king, take it.
    - If checkers at (7,y) and (8,y-1) or (8,y+1) is free, move there.
  - If there are two legal moves, choose the one that moves a checker farther toward the top row
    - If checker(x,y) and checker(p,q) can both move, and x>p, move checker(x,y).
  - But then each of these heuristics needs some kind of priority or **weight**.

78

Formalization for RL Agent

- Given:
  - A state space S
  - A set of actions a_1, …, a_k including their results
  - **A set of heuristics for resolving conflict among actions**
  - Reward value at the end of each trial (series of action) (may be positive or negative)
- Output:
  - A policy (a mapping from states to preferred actions)

79

Learning Agent

- The general algorithm for this learning agent is:
  - Observe some state
  - If it is a terminal state
    - Stop
    - If won, **increase** the weight on **all** heuristics used
    - If lost, **decrease** the weight on **all** heuristics used
  - Otherwise choose an action from those possible in that state, using heuristics to select the preferred action
  - Perform the action

80

Policy

- A complete mapping from states to actions
  - There must be an action for each state
  - There may be more than one action
  - Not necessarily optimal
- The goal of a learning agent is to **tune** the policy so that the preferred action is optimal, or at least good.
  - Analogous to training a classifier
- Checkers
  - Trained policy includes all legal actions, with **weights**
  - “Preferred” actions are **weighted up**

81

Approaches

- Learn policy directly: Discover function mapping from states to actions
    - Could be directly learned values
        - Ex: Value of state which removes last opponent checker is +1.
    - Or a heuristic function which has itself been trained
- Learn utility values for states (value function)
    - Estimate the value for each state
    - Checkers:
        - How happy am I with this state that turns a piece into a king?

Value Function

- The agent knows what state it is in
- It has actions it can perform in each state
- Initially, don’t know the value of any of the states
- If the outcome of performing an action at a state is deterministic, then the agent can update the utility value U() of states:
    - U(oldstate) = reward + U(newstate)
- The agent learns the utility values of states as it works its way through the state space

Learning States and Actions

- A typical approach is:
- At state S choose, some action A ← How?
- Taking us to new State S_1
  - If S_1 has a positive value: increase value of A at S.
  - If S_1 has a negative value: decrease value of A at S.
  - If S_1 is new, initial value is unknown: value of A unchanged.
- One complete learning pass or **trial** eventually gets to a terminal, deterministic state. (E.g., "win" or "lose")
- Repeat until? Convergence? Some performance level?

84

Selecting an Action

- Simply choose action with highest (current) expected utility?
- Problem: each action has two effects
  - Yields a **reward** on current sequence
  - Gives **information** for learning future sequences
- Trade-off: immediate good for long-term well-being
  - Like trying a shortcut: might get lost, might find quicker path
- **Exploration** vs. **exploitation**

85

Exploration vs. Exploitation

- Problem with naïve reinforcement learning:
  - What action to take?
  - **Best apparent action, based on learning to date** } Exploitation
    - Greedy strategy
    - Often prematurely converges to a suboptimal policy!
  - **Random (or unknown) action** } Exploration
    - Will cover entire state space
    - Very expensive and slow to learn!
    - When to stop being random?
- Balance exploration (try random actions) with exploitation (use best action so far)

86

More on Exploration

- Agent may sometimes choose to explore suboptimal moves in hopes of finding better outcomes
  - Only by visiting all states frequently enough can we guarantee learning the true values of all the states
- When the agent is **learning**, ideal would be to get accurate values for all states
  - Even though that may mean getting a negative outcome
- When agent is **performing**, ideal would be to get optimal outcome
- A learning agent should have an **exploration policy**

87

Exploration Policy

- Wacky approach (exploration): act randomly in hopes of eventually exploring entire environment
  - Choose any legal checkers move
- Greedy approach (exploitation): act to maximize utility using current estimate
  - Choose moves that have in the past led to wins
- Reasonable balance: act more wacky (exploratory) when agent has little idea of environment; more greedy when the model is close to correct
  - Suppose you know no checkers strategy?
  - What's the best way to get better?

88

Example: N-Armed Bandits

- A row of slot machines
- Which to play and how often?
- State Space is a set of machines
  - Each has cost, payout, and percentage values
- Action is pull a lever.
- Each action has a positive or negative result
  - …which then adjusts the perceived utility of that action (pulling that lever)

¢25	¢95	$10
$100	$200	$900
0.1%	0.6%	10%

89

N-Armed Bandits Example

- Each action initialized to a standard payout
- Result is either some cash (a win) or none (a lose)
- **Exploration**: Try things until we have estimates for payouts
- **Exploitation**: When we have some idea of the value of each action, choose the best.

After some # of successful trials, or with some statistical **confidence**, or when our value function isn't changing (much), or…

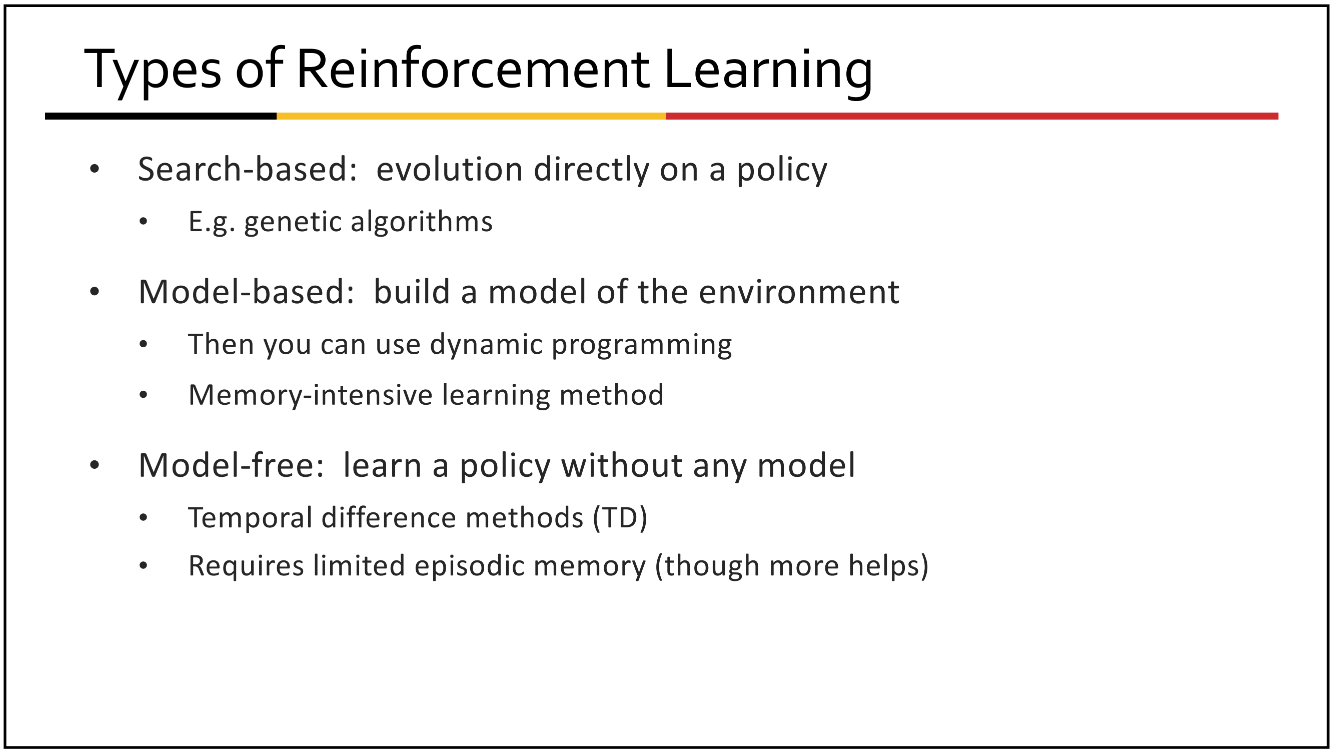
- Clearly this is a heuristic.
- No proof we ever find the best lever to pull!
  - The more exploration we can do the better our model
  - But the higher the cost over multiple trials

90

Mathematical Model - MDP

- Markov decision processes
- S - set of states
- A - set of actions
- δ - Transition probability
- R - Reward function

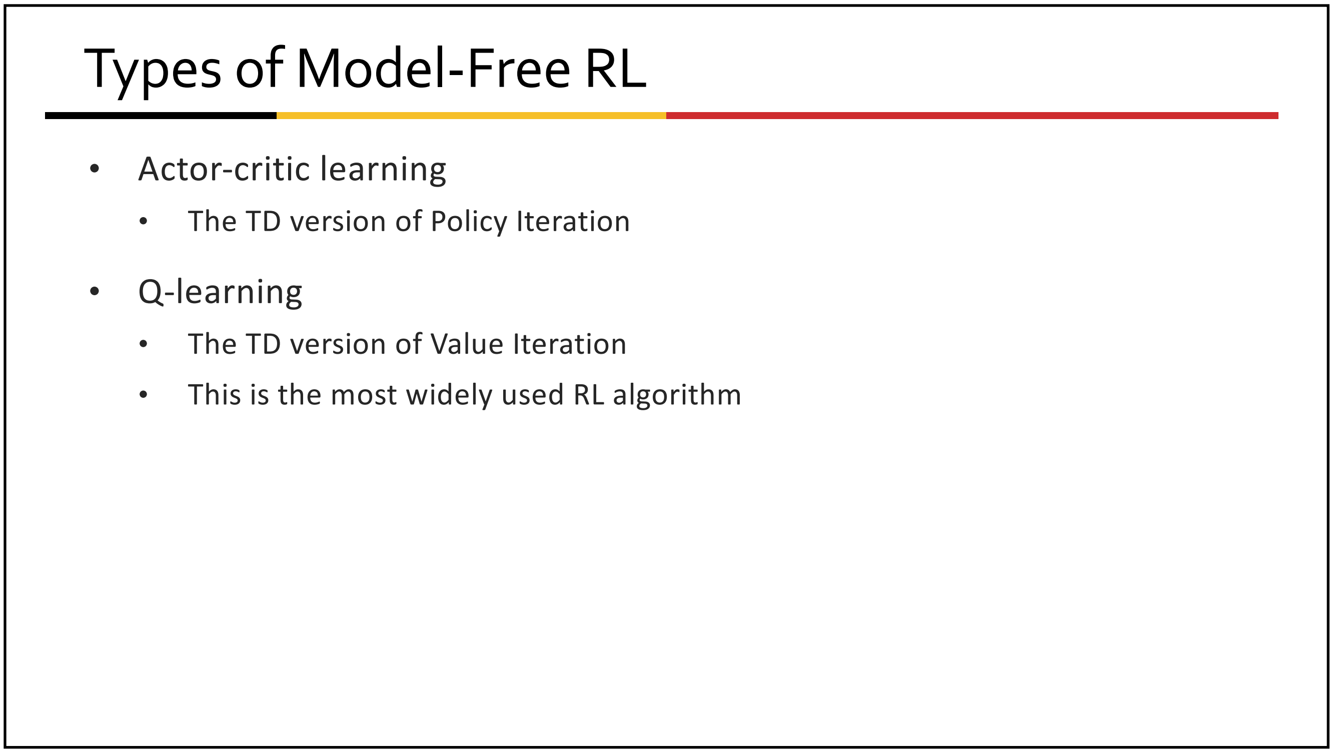
91



Types of Reinforcement Learning

- Search-based: evolution directly on a policy
  - E.g. genetic algorithms
- Model-based: build a model of the environment
  - Then you can use dynamic programming
  - Memory-intensive learning method
- Model-free: learn a policy without any model
  - Temporal difference methods (TD)
  - Requires limited episodic memory (though more helps)

92



Types of Model-Free RL

- Actor-critic learning
  - The TD version of Policy Iteration
- Q-learning
  - The TD version of Value Iteration
  - This is the most widely used RL algorithm

93

Q-Learning: Definitions

- Current state: s
- Current action: a
- Transition function: $\delta(s, a) = s'$
- Reward function: $r(s, a) \epsilon R$
- Policy $\pi(s) = a$

Markov property: this is independent of previous states given current state

In classification we'd have examples ***(s, π(s))*** to learn from

- $Q(s, a) \approx$ value of taking action a from state s

94

The Q-function

- **Q(s, a)** estimates the discounted cumulative reward
  - Starting in state s
  - Taking action a
  - Following the current policy thereafter
- Suppose we have the optimal Q-function
  - What's the optimal policy in state s?
  - The action $argmax_b Q(s, b)$
- But we don't have the optimal Q-function at first
  - Let's act as if we do
  - And updates it after each step so it's closer to optimal
  - Eventually it will be optimal!

95

Q-Learning: Updates

- The basic update equation

$$Q(s,a) \longleftarrow r(s,a) + \max_b Q(s',b)$$

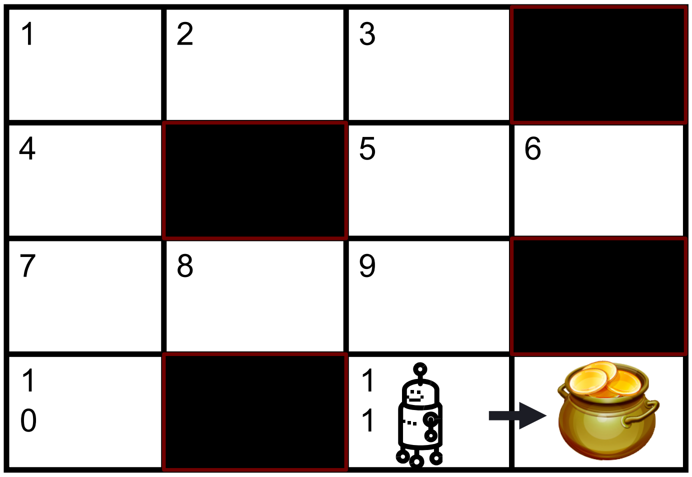
- With a discount factor to give later rewards less impact

$$Q(s,a) \longleftarrow r(s,a) + \gamma \max_b Q(s',b)$$

- With a learning rate for non-deterministic worlds

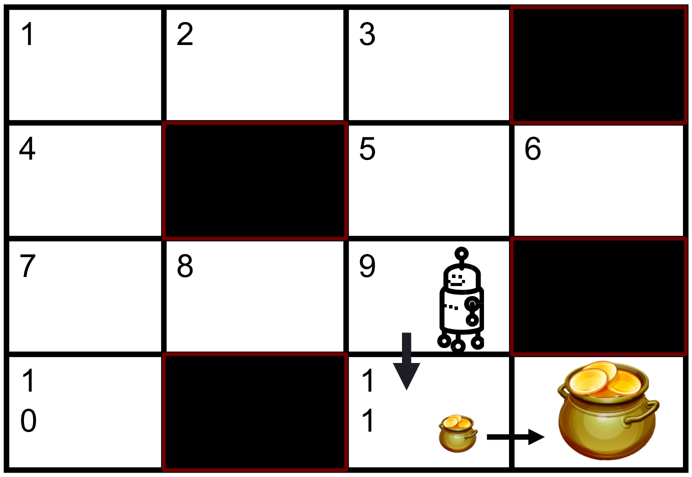
$$Q(s,a) \longleftarrow [1-\alpha]Q(s,a) + \alpha[r(s,a) + \gamma \max_b Q(s',b)]$$

Q-Learning: Update Example



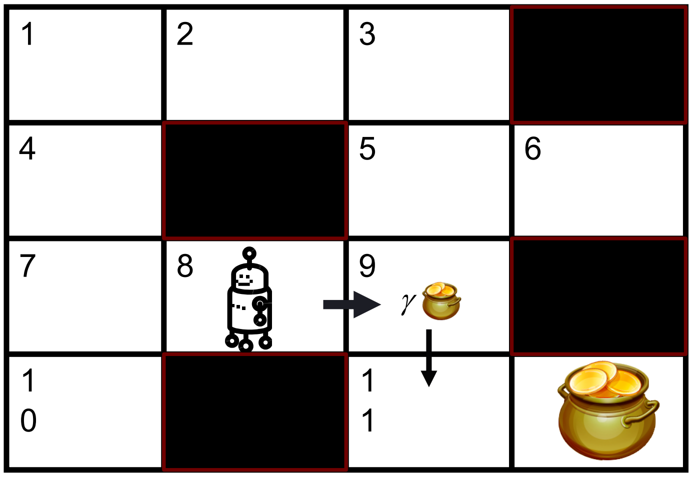
$$Q(s_{11}, a_{\rightarrow}) =$$

Q-Learning: Update Example



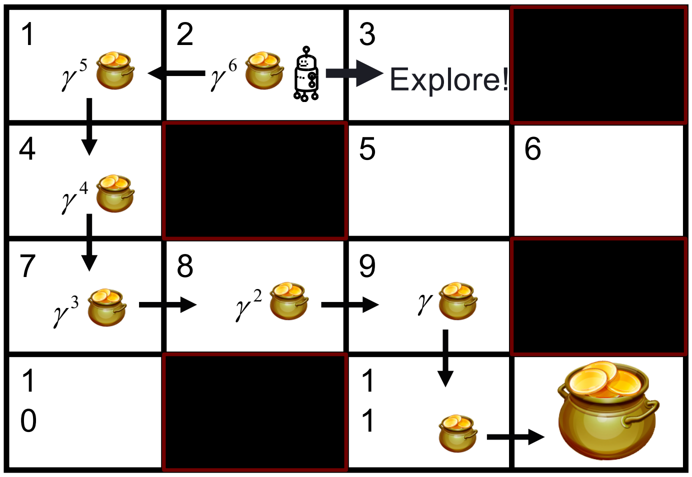
$Q(s_9, a_\downarrow) = 0 + \gamma$ 💰

Q-Learning: Update Example



$Q(s_8, a_\rightarrow) = 0 + \gamma^2$ 💰

The Need for Exploration



$$\arg\max Q(s_2, a) = \leftarrow$$

$$best = \rightarrow$$

100

RL Summary 1:

- **Reinforcement learning systems**
  - Learn **series** of actions or decisions, rather than a single decision
  - Based on feedback given at the end of the series
- A reinforcement learner has
  - A goal
  - Carries out trial-and-error search
  - Finds the best paths toward that goal

101

Exploration/Exploitation

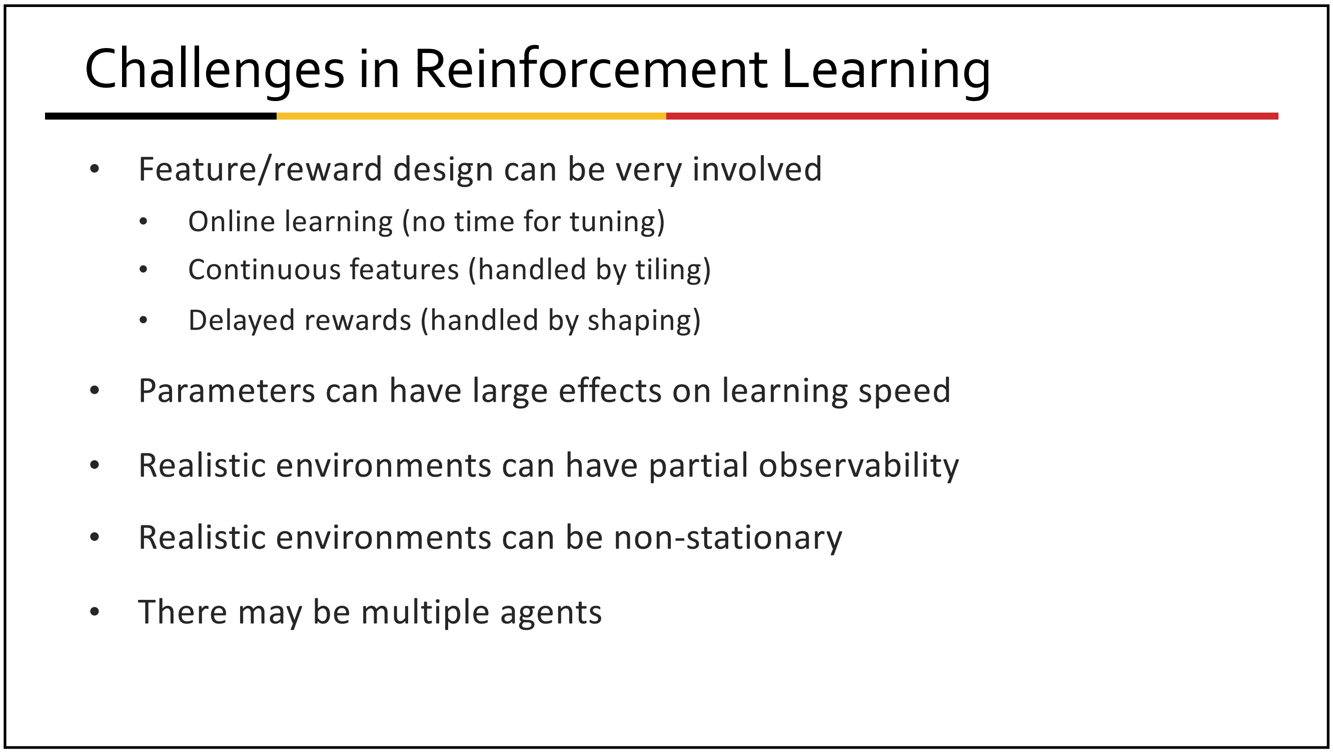
- Can’t always choose the action with highest Q-value
  - The Q-function is initially unreliable
  - Need to explore until it is optimal
- Most common method: ε-greedy
  - Take a random action in a small fraction of steps (ε)
  - Decay ε over time
- There is some work on optimizing exploration
  - Kearns & Singh, ML 1998
  - But people usually use this simple method

102

Q-Learning: Convergence

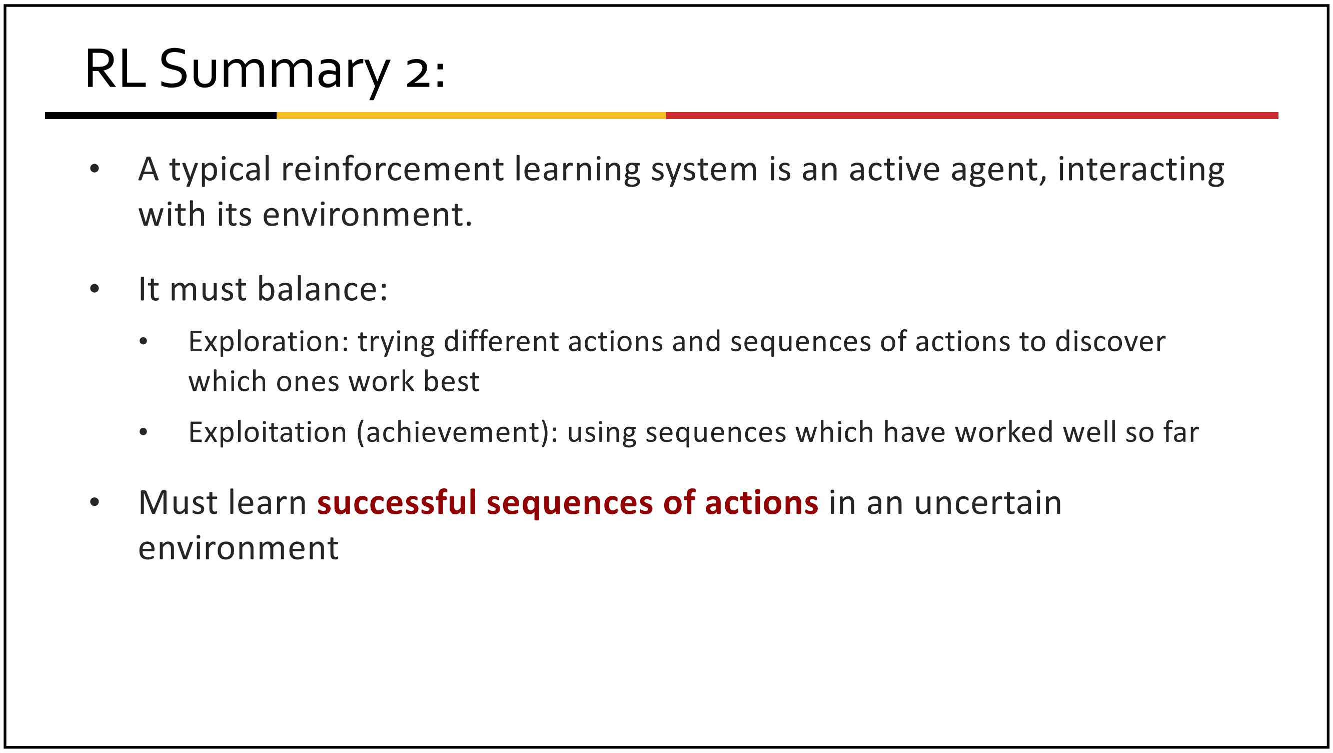
- Under certain conditions, Q-learning will converge to the correct Q-function
  - The environment model doesn’t change
  - States and actions are finite
  - Rewards are bounded
  - Learning rate decays with visits to state-action pairs
  - Exploration method would guarantee infinite visits to every state-action pair over an infinite training period

103



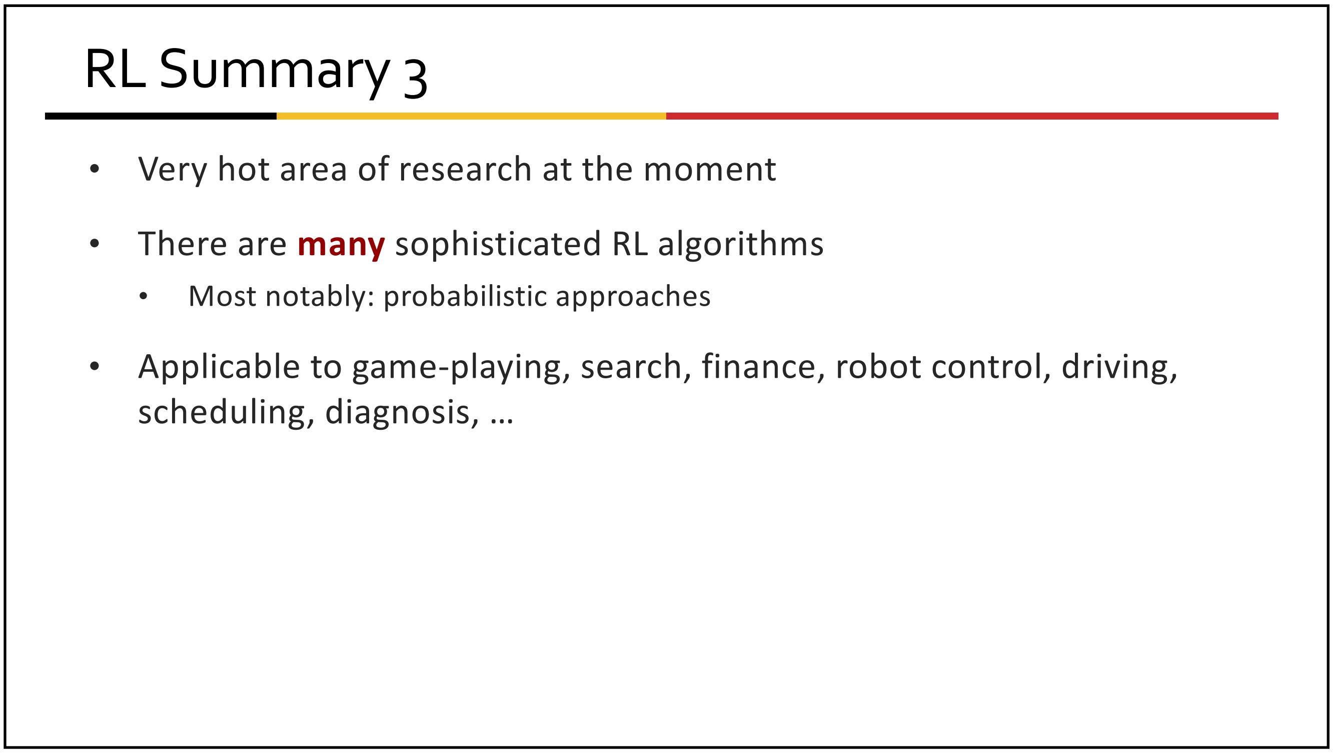
Challenges in Reinforcement Learning

- Feature/reward design can be very involved
  - Online learning (no time for tuning)
  - Continuous features (handled by tiling)
  - Delayed rewards (handled by shaping)
- Parameters can have large effects on learning speed
- Realistic environments can have partial observability
- Realistic environments can be non-stationary
- There may be multiple agents



RL Summary 2:

- A typical reinforcement learning system is an active agent, interacting with its environment.
- It must balance:
  - Exploration: trying different actions and sequences of actions to discover which ones work best
  - Exploitation (achievement): using sequences which have worked well so far
- Must learn **successful sequences of actions** in an uncertain environment



RL Summary 3

- Very hot area of research at the moment
- There are **many** sophisticated RL algorithms
  - Most notably: probabilistic approaches
- Applicable to game-playing, search, finance, robot control, driving, scheduling, diagnosis, …