DYNAMIC PROGRAMMING FOR REINFORCEMENT LEARNING

SAN DIEGO MACHINE LEARNING

JUNE 5, 2021

HOW TO PARTICIPATE

- One discussion leader, and everyone welcome to participate
- Majority of material comes from Reinforcement Learning by Sutton and Barto
- Options to approach the content:
  - Treat this as a standalone webinar
  - Read the book first, and come with questions and discussion items
  - Use this meetup as a primer and read the chapters afterward
- Ask questions
- Give feedback. Too fast or too slow? Want to see more of something or less of something else?
- Have fun!

AGENDA

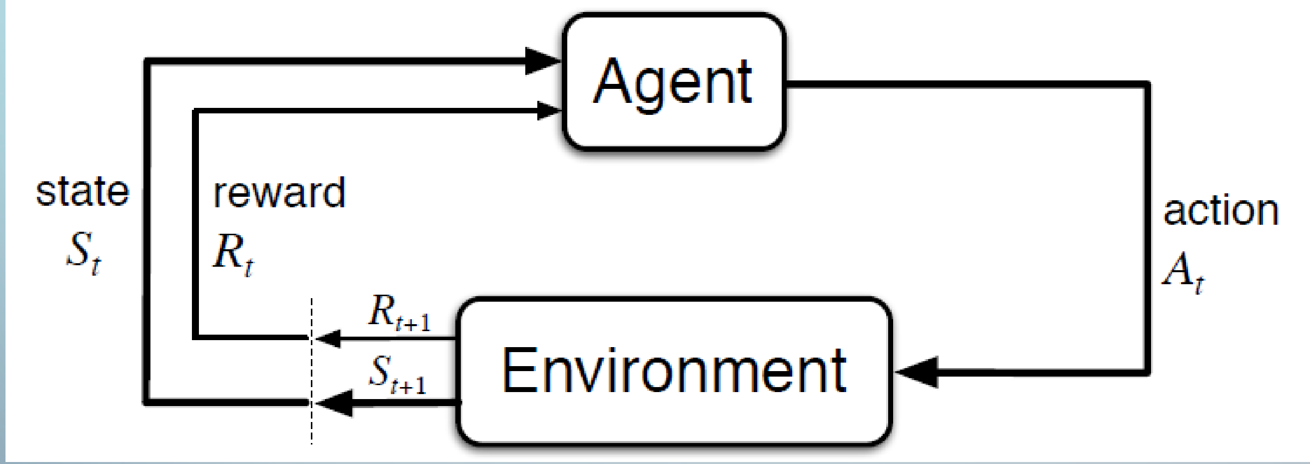
- Recap what reinforcement learning (RL) is
  - Elements and formulation as Markov decision processes (MDP)
  - Terminology and notation used in RL
  - The Bellman equations
- Introduce Dynamic Programing
  - Iterative policy evaluation
  - Policy improvement
  - Policy iteration
  - Extensions such as value iteration, general policy improvement

REINFORCEMENT LEARNING

- Reinforcement learning (RL) is about an *agent* learning from interacting with its uncertain *environment*
  - The agent interacts by choosing from a set of allowed *actions*
  - It gets feedback from a numeric *reward* signal
  - Goal is to maximize the *return*, which is the total rewards received
- Reinforcement learning is about exploring the environment and recording useful information for the future
- RL is sequential decision making; time is intrinsic

MARKOV DECISION PROCESSES

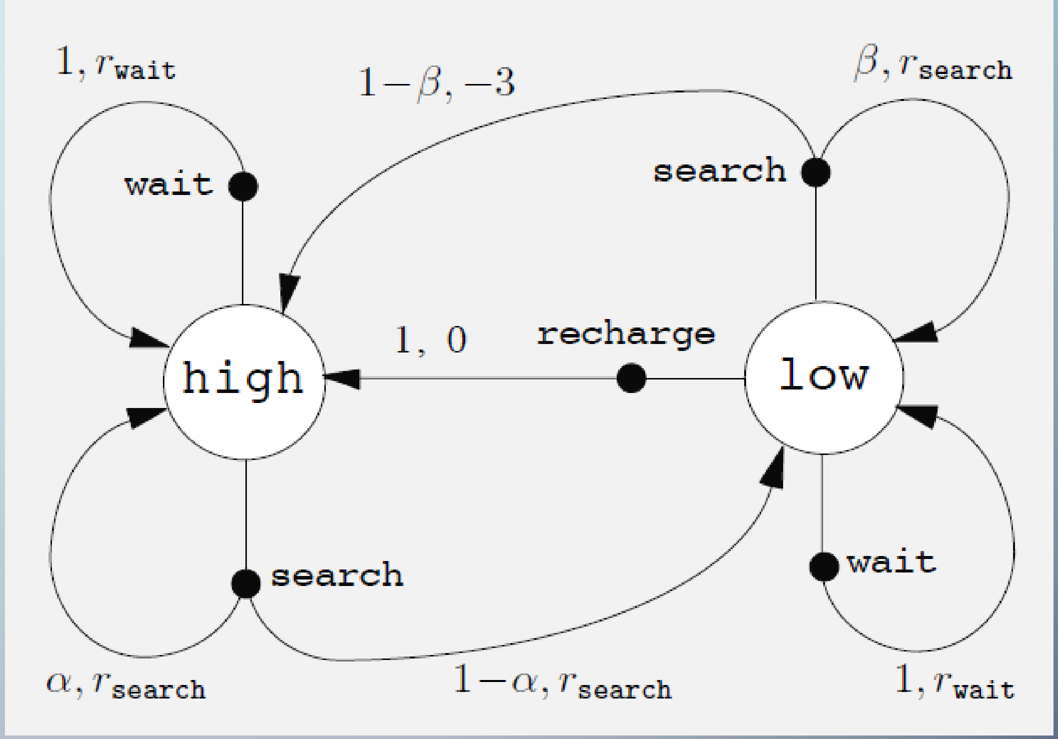
- Elements of the fully observable Markov Decision Process (MDP):
  - State - at each time step t, the environment is in some state S_t
  - Action - at each time step t, the agent chooses an action A_t
  - Reward - after taking the action, the agent is given a reward signal R_{t+1} and subsequently finds itself in a new state S_{t+1}



- In a *Markov* Decision Process, the transition at any given time t <u>only</u> depends on the state S_t and action chosen A_t

MDPS AS A GRAPH

- Sometimes it is easier to visualize a MDP as a directed graph
  - The states are nodes (big white circles)
  - The actions are edges leading from nodes (here with small black circles)
  - The rewards are values along directed edges that take you to a new state
- Here is the recycling robot from the book:



REINFORCEMENT LEARNING NOTATION

Letter	Used for
s	**S**tate
a	**A**ction
r	**R**eward
γ	Discount rate
G	Return – sum of all future rewards
p	Transition **p**robability
v	**V**alue function for states
q	Value function for state-action pairs
π	Policy (**π**ολιτική)
*	Optimal choices, e.g. π_*

BELLMAN EQUATION

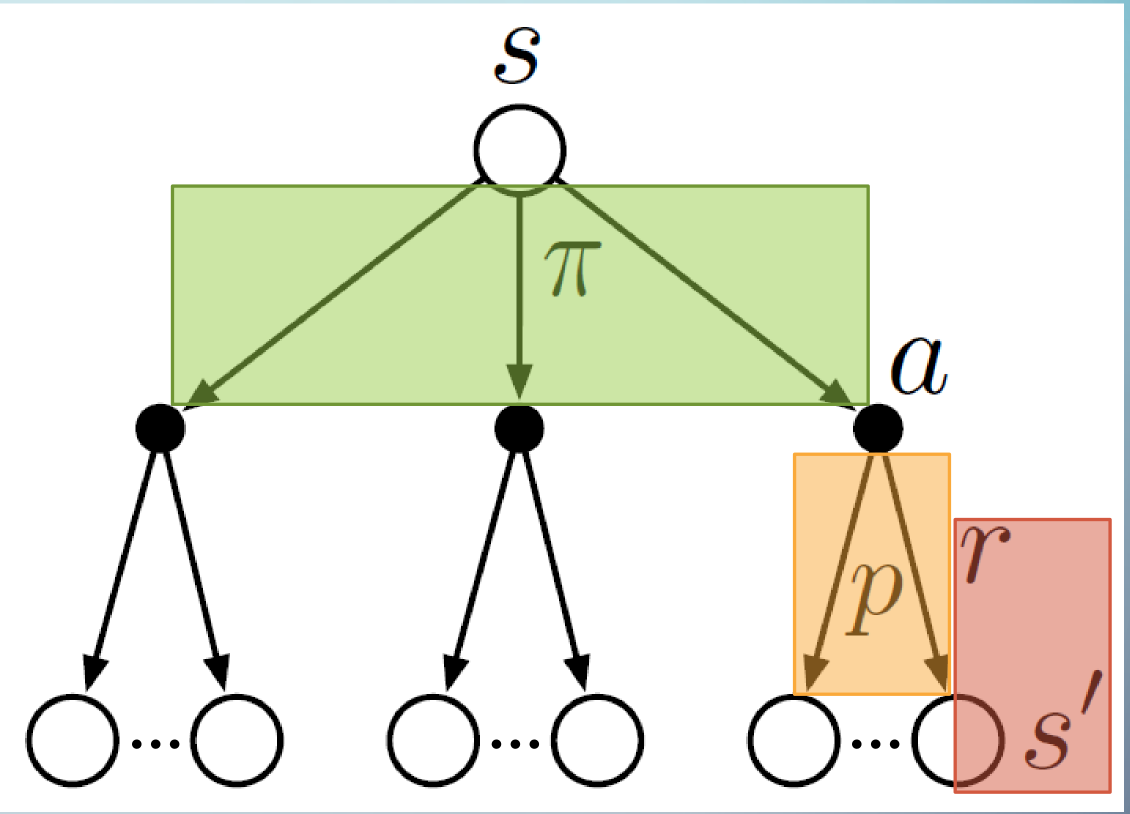
- The value function for state s under policy π is a sum of the rewards received and the value functions for each future state s' times the probability of winding up there
- Formally:

$$v_\pi(s) = \sum_a \underbrace{\pi(a,s)}_{\substack{\textbf{Probability you} \\ \textbf{take action } a}} \sum_{s',r} \underbrace{p(s',r|s,a)}_{\substack{\textbf{Probability you} \\ \textbf{get reward } r \\ \textbf{and end in state } s'}} [\underbrace{r+\gamma v_\pi(s')}_{\substack{\textbf{Reward plus} \\ \textbf{discounted value} \\ \textbf{of new state } s'}}]$$

BELLMAN EQUATION VISUALIZED

This is a *backup diagram* for $v_\pi(s)$. To compute it:

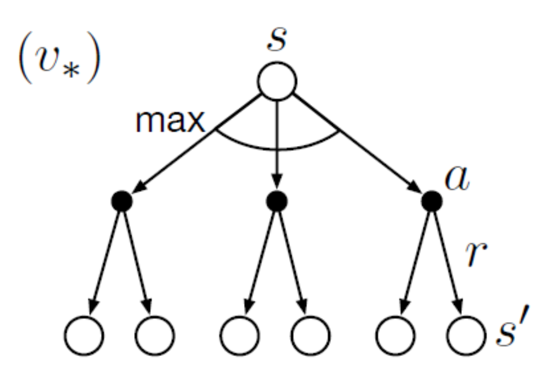
- We need to sum over each branch of π(), based on the probability of each action *a*
- And sum over of each branch of *p*(), based on probability we wind up in state *s'*
- The quantity we sum is the reward and the discounted value of possible state *s'*



$$v_\pi(s) = \sum_a \pi(a,s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

BELLMAN OPTIMALITY EQUATIONS

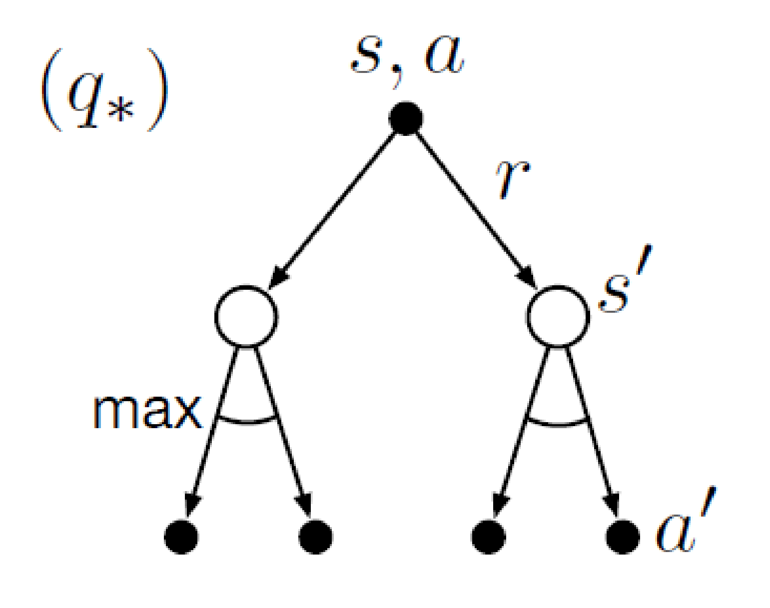
- The *Bellman optimality equation* says the optimal value for a state must be the same as the return from the best action
- We can rewrite it recursively



$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] && \text{(by (3.9))} \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] && (3.18) \\
&= \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma v_*(s')\big]. && (3.19)
\end{aligned}
$$

BELLMAN OPTIMALITY EQUATIONS

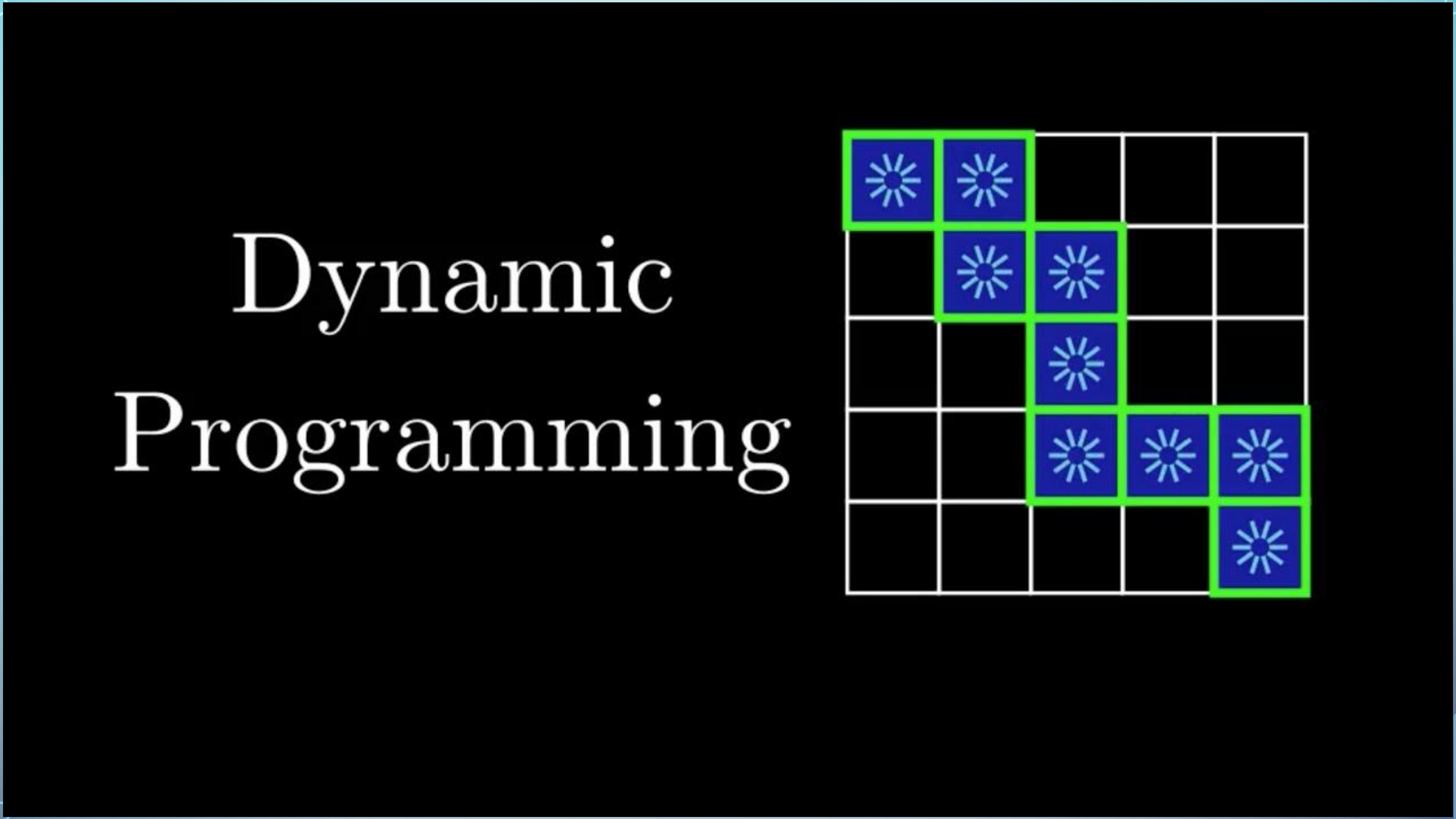
- The *Bellman optimality equation* for state-action pairs is very similar.
- The optimal value for a state-action pair must be the same as the return from the reward and best next action
- It also can be written recursively



$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big] \\
&= \sum_{s',r} p(s', r | s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big]. \qquad (3.20)
\end{aligned}
$$

REINFORCEMENT LEARNING CONTROL

- With this foundation, there's a lot we can tackle
  - Algorithms for learning
  - Dealing with memory and compute limitations
  - Getting models to converge quickly
- We also still have many challenges
  - Reward design – effectively communicating the real goal
  - Sparse rewards
  - Credit assignment – which actions in trajectory contributed
  - Exploration vs. exploitation


Dynamic
Programming

DYNAMIC PROGRAMMING

- Dynamic programming (DP) is a technique to compute optimal policies given a perfect model of the environment as a MDP
- Limited practical use because of the need for full knowledge of environment and expensive to compute
    - But it's a very useful theoretical building block
- DP builds on optimal solutions of subproblems where those subproblems overlap a lot
- For RL, we use DP to iteratively compute value functions. From there we can obtain optimal value functions and optimal policies

DP PREDICTION

- Here's the Bellman Equation again:

$$v_\pi(s) = \sum_a \pi(a, s) \sum_{s',r} p(s', r | s, a)[r + \gamma v_\pi(s')]$$

- If the environment dynamics are completely known (i.e. we know all values of the *p*() function), then for |S| states:
  - We can write |S| Bellman equations for each $v_\pi(s)$
  - Each of which is in up to |S| unknowns of $v_\pi(s')$
  - This is just a linear system of N equations in N unknowns and can be solved
  - Instead of going this route, DP iteratively approximates *v*()

DP PREDICTION

- Dynamic programming starts with an initial value for $v_\pi()$
  - Could be as simple as $v_\pi(s) = 0$ for every state s
- DP then tweaks the recursively expressed Bellman equation:

$$v_\pi(s) = \sum_a \pi(a,s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

into an update rule from iteration k to iteration k+1:

$$v_{k+1}(s) = \sum_a \pi(a,s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')]$$

DP PREDICTION

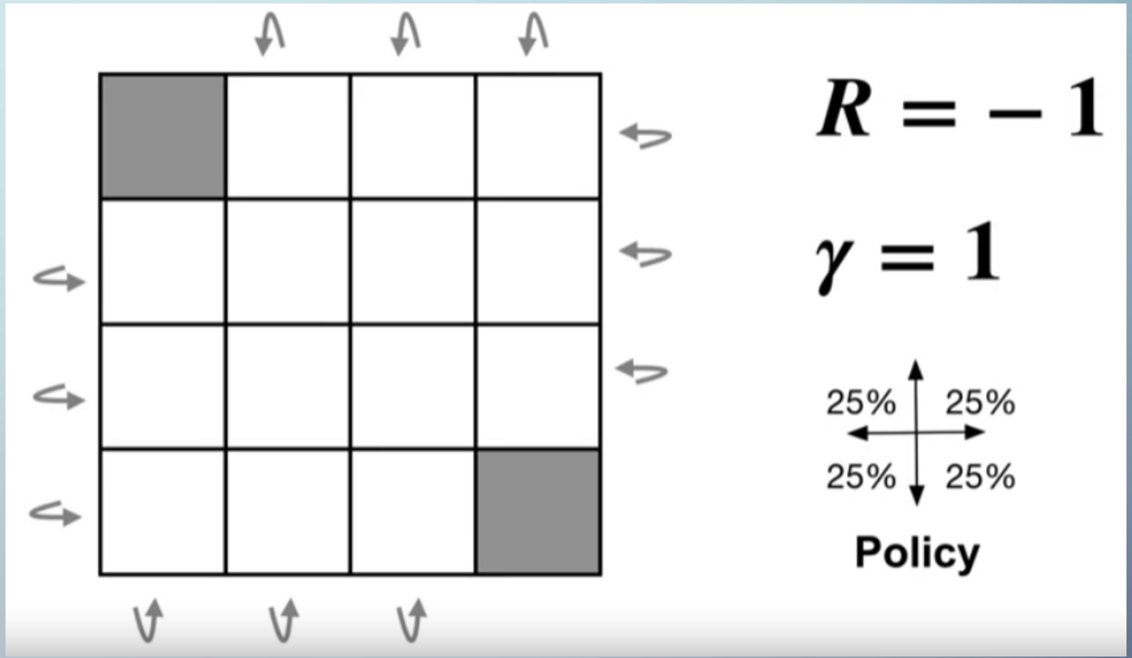
- The update rule simply says to calculate a new k+1 value for $v_\pi(s)$ for each state using all the estimates from the kth iteration

$$v_{k+1}(s) = \sum_{a} \pi(a,s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')]$$

- If we keep iterating until none of the state's estimated value is updated by more than some small number θ, then we will have converged very close to $v_\pi(s)$
- The above algorithm is called *iterative policy evaluation*
  - Given a policy π, iterative policy evaluation will calculate $v_\pi(s)$

ITERATIVE POLICY EVALUATION

- The Fundamentals of RL course on Coursera has a nice animation of iterative policy evaluation for a small grid world example
- https://www.coursera.org/learn/fundamentals-of-reinforcement-learning/lecture/ICAfp/iterative-policy-evaluation (time 3:20)
- The basic version of iterative policy evaluation uses *sweeps* where temporary variables hold all of the new values for each state, and then they are updated all at once



POLICY IMPROVEMENT

- In order to do dynamic programming, we said we fully knew the environment, so we fully know $p(s', r \mid s, a)$
- If we know *p*(), then for every action *a* we know what the reward *r* will be, and we can compute $r+\gamma v_\pi(s')$
- If for an action *a* that is not part of our policy π, $r+\gamma v_\pi(s') > v_\pi(s)$, then a new policy π' that takes action *a* from state *s* must be better than π
- We can be *greedy*, choosing all actions that are better in the above way

POLICY IMPROVEMENT

- Formally, choosing the best action looking one step ahead for every state gives us a new policy:

$$\pi'(s) = \underset{a}{\operatorname{argmax}} \sum_{s',r} p(s',r|s,a)[r + \gamma v_{\pi}(s')]$$

- This process of building a new policy based on the best options using the current value function is called *policy improvement*

POLICY ITERATION

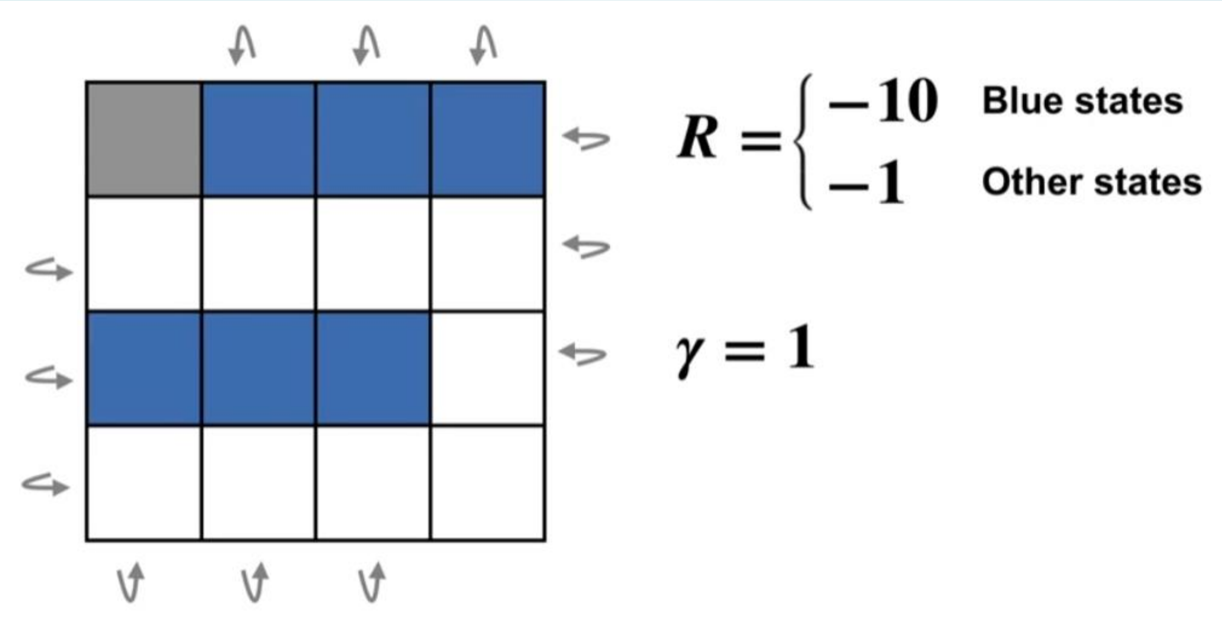
- After improving a policy, we can compute the new value function and improve the policy again
- The book shows a sequence like this:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_1 \xrightarrow{E} \ \dots \ \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

  - The arrows with E’s are full cycles of iterative policy evaluation
  - And the arrows with I’s are policy improvement
  - If a policy improvement doesn’t result in any changes, then we have converged on the optimal policy, and we can stop
- This alternating pattern of policy evaluation and improvement is called *policy iteration*

POLICY ITERATION

- The Fundamentals of RL course on Coursera also has a nice animation of policy iteration for a modified grid world example
- https://www.coursera.org/learn/fundamentals-of-reinforcement-learning/lecture/Xv32P/policy-iteration (4:25)



VALUE ITERATION

- In the policy iteration shown so far, policy evaluation involves a complete cycle of iterative policy evaluation
- We can shorten policy evaluation, and one special way is to do one single sweep of policy evaluation
- If we do this, the math works out to:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')]$$

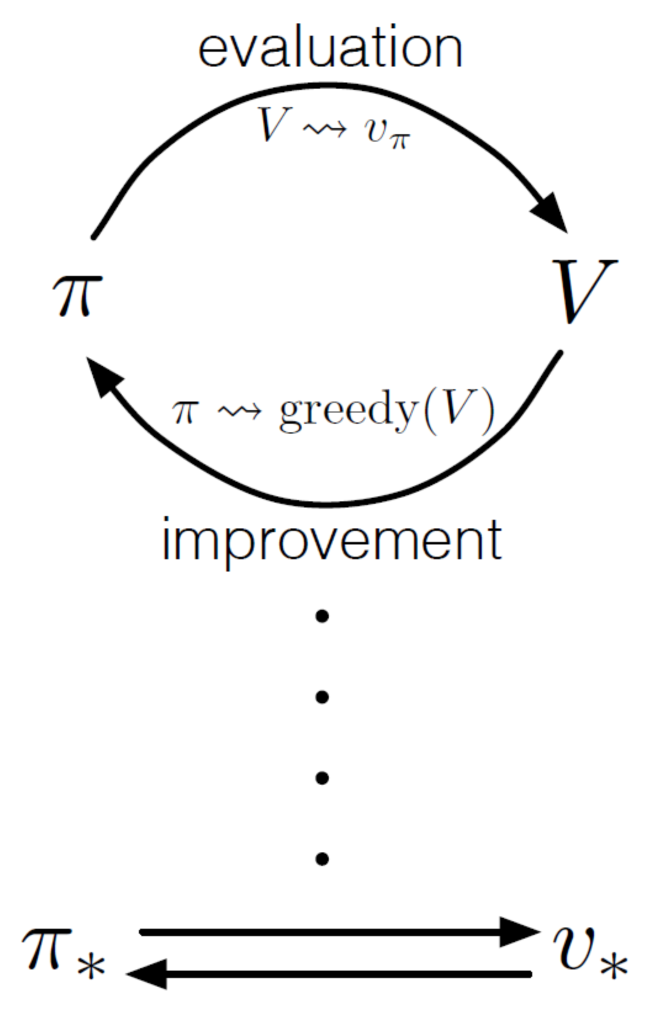
- And this is actually tweaking the Bellman optimality equation for states into an update rule

ASYNCHRONOUS DYNAMIC PROGRAMMING

- Another change we can make to the standard DP is to get rid of the full sweeps in policy evaluation
  - RL problems can have very many states, so a single sweep could be massive, e.g. backgammon has over 10^{20} states
  - In the extreme, we could evaluate just one state each time
- As long as we eventually update the values of all of the states over time, the asynchronous version will also converge to π_*
- Asynchronous versions can work in real time as the agent is experiencing the environment
- Interestingly, they also allow us to choose to focus on updating some states more than others, or tune the order of updates

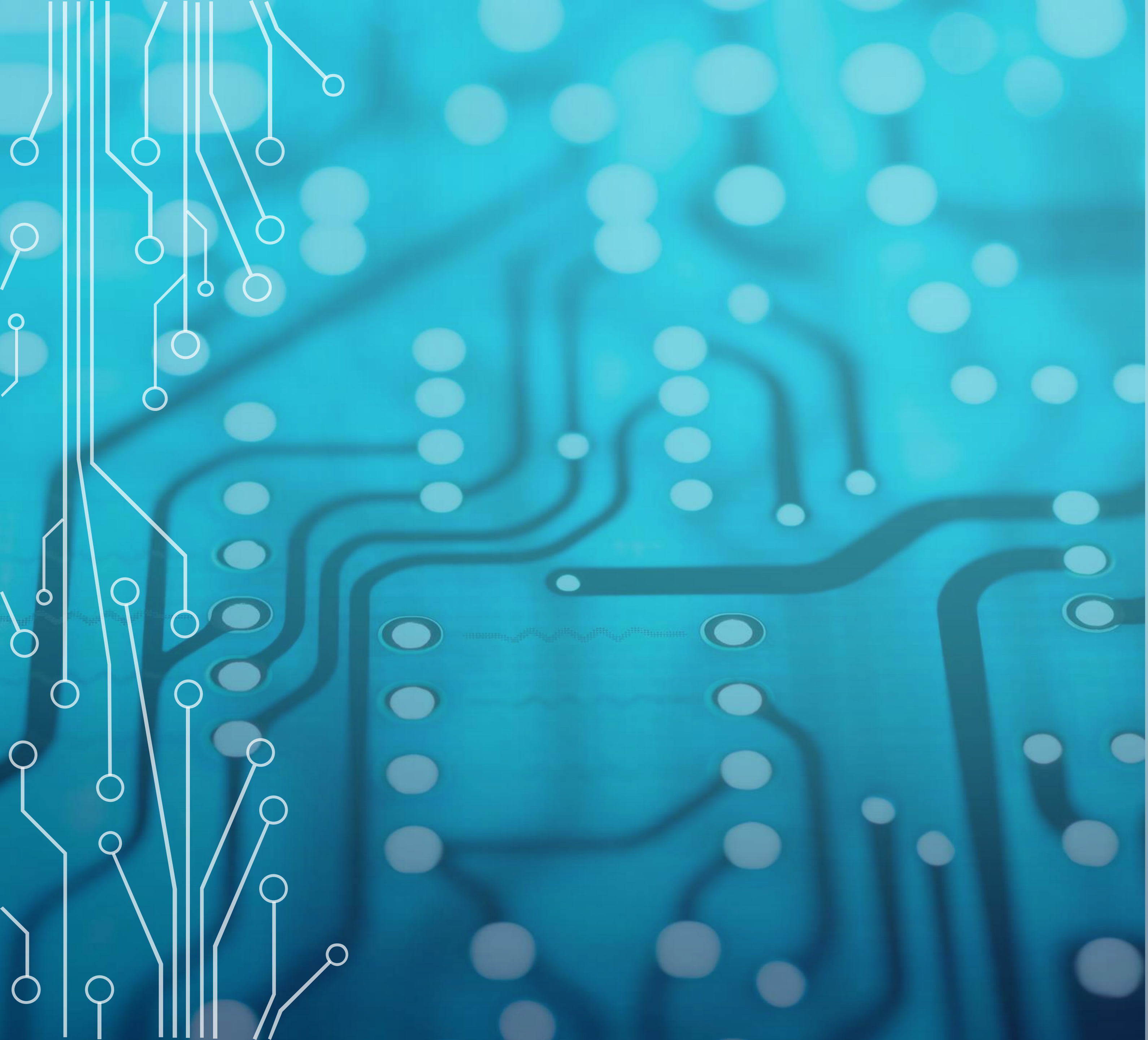
GENERALIZED POLICY ITERATION

- The term *generalized policy iteration* (GPI) refers to the general idea of letting policy evaluation and policy improvement processes interact
  - Doesn't matter how fully each evaluation or improvement step runs, or if they exactly alternate
- Note also that DP algorithms update values of states based on values of successor states. This behavior is called *bootstrapping*. Not all RL algorithms bootstrap.



RECAP

- Review what reinforcement learning (RL) is
  - Elements and formulation as Markov decision processes (MDP)
  - Terminology and notation used in RL
  - The Bellman equations
- Introduce Dynamic Programing
  - Iterative policy evaluation
  - Policy improvement
  - Policy iteration
  - Extensions such as value iteration, general policy improvement



QUESTIONS

&

DISCUSSION

NEXT SESSION

- We will finish the content of this topic, discussing Monte Carlo methods next week, Sat. June 12
- The following session will be about Temporal Difference Learning, on Sat. June 19
- This TD material is in chapter 6 of Sutton & Barto