Chapter 21:
Reinforcement Learning

These slides are adopted from Berkeley course materials, reinforcement learning Stanford course materials, UBC Intelligent Systems materials, Reinforcement Learning Sutton and Barto book, and Russell and Norvig textbook.

Recall-Markov decision process (MDP)

A Markov decision process (MDP) is defined by:

- A set of states $s \in S$
- A set of actions (per state) A
- A transition model $P(s'|a,s)$
- A reward function $R(s,a,s')$ or $R(s)$

Still looking for a policy $\pi(s)$

An optimal policy is a policy that maximizes the expected total reward.

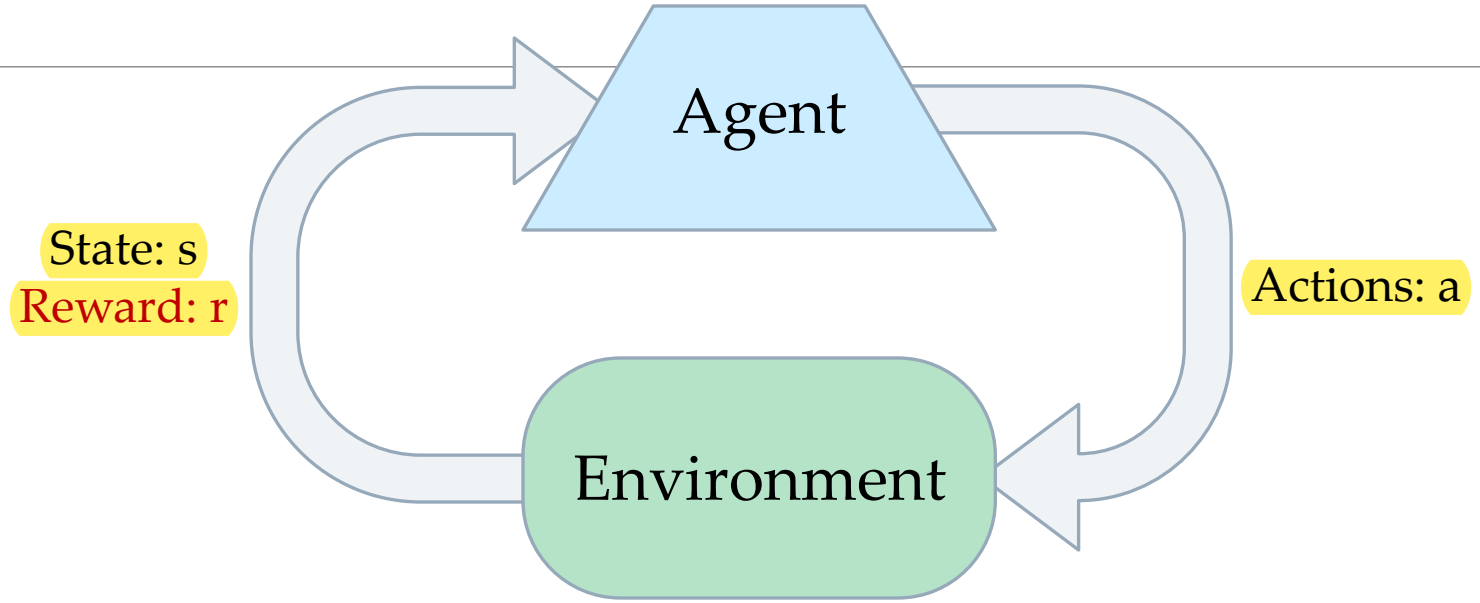
el reinforcement learning hya tefl byt3lm, fa el tefl da m3ndosh ay m3lomat, fa hwa byt3lm 3n tre2 eno ye3rf el nar btls3 wla la2, 3n tre2 eno yro7 y7ot eido feha.

w da khlahom ye2dro yekto algorithm 1 bs, w ydkhloh 3la al3ab ktera, w ygeb score 3alya fehom kolohom.

Reinforcement Learning

- The task of **reinforcement learning** is to use observed rewards to learn an optimal (or nearly optimal) policy for the environment.
- Whereas in Chapter 17 the agent has a complete model of the environment (Transition model and reward function).
- In this chapter, we still assume a Markov decision process, however the agent does not know both of **the transition model P(s' | s, a),** which specifies the probability of reaching state s from state s after doing action a; nor does it know **the reward function R(s),** which specifies the reward for each state.
  - In other words, the agent does not know which states are good or what the actions do.
  - Must actually try actions and states out to learn.

Reinforcement Learning



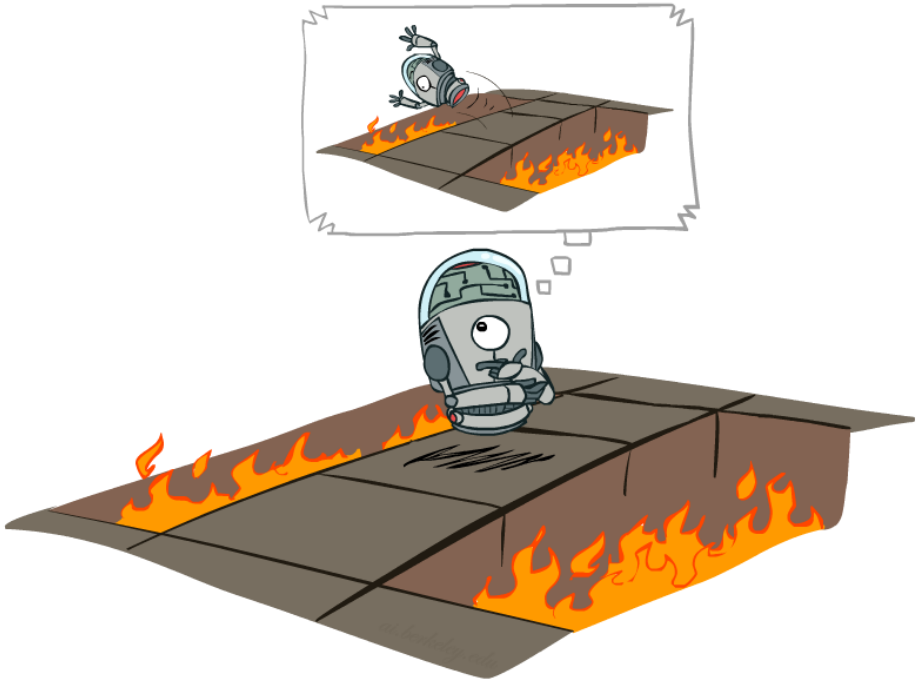
Basic idea:

- Receive feedback in the form of rewards
- Agent's utility is defined by the reward function
- Must (learn to) act so as to maximize expected rewards
- All learning is based on observed samples of outcomes!

Reinforcement Learning

- In many complex domains, reinforcement learning is the only feasible way to train program to perform at high levels.
- For example, in game playing, it is very hard for a human to provide accurate and consistent evaluations of large numbers of positions, which would be needed to train an evaluation function directly from examples.
- Instead, the program can be told when it has won or lost, and it can use this information to learn an evaluation function that gives reasonably accurate estimates of the probability of winning from any given position.

Offline (MDPs) vs. Online (RL)



Offline Solution

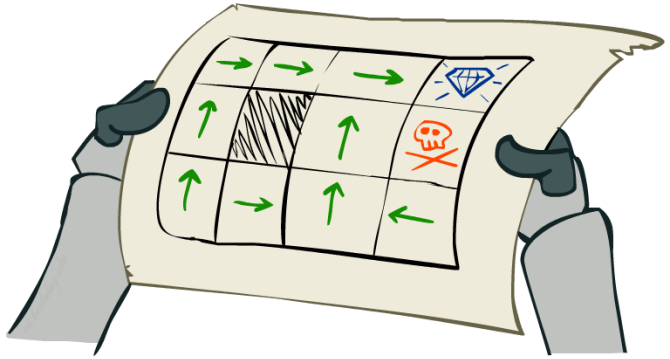
Online Learning

Passive Reinforcement Learning

- We assume a fully observable environment.
- In passive learning, the agent's policy π is fixed:

In state s, it always executes the action π(s).

- Its goal is simply to learn how good the policy is that is, to learn the utility function $U^{\pi}(s)$.



- The passive learning task is similar to the **policy evaluation** task in the **policy iteration** algorithm (Chapter 17).
- However, the passive learning agent does not know the **transition model** P(s' | s, a) nor does it know the **reward function** R(s).

Passive Reinforcement Learning

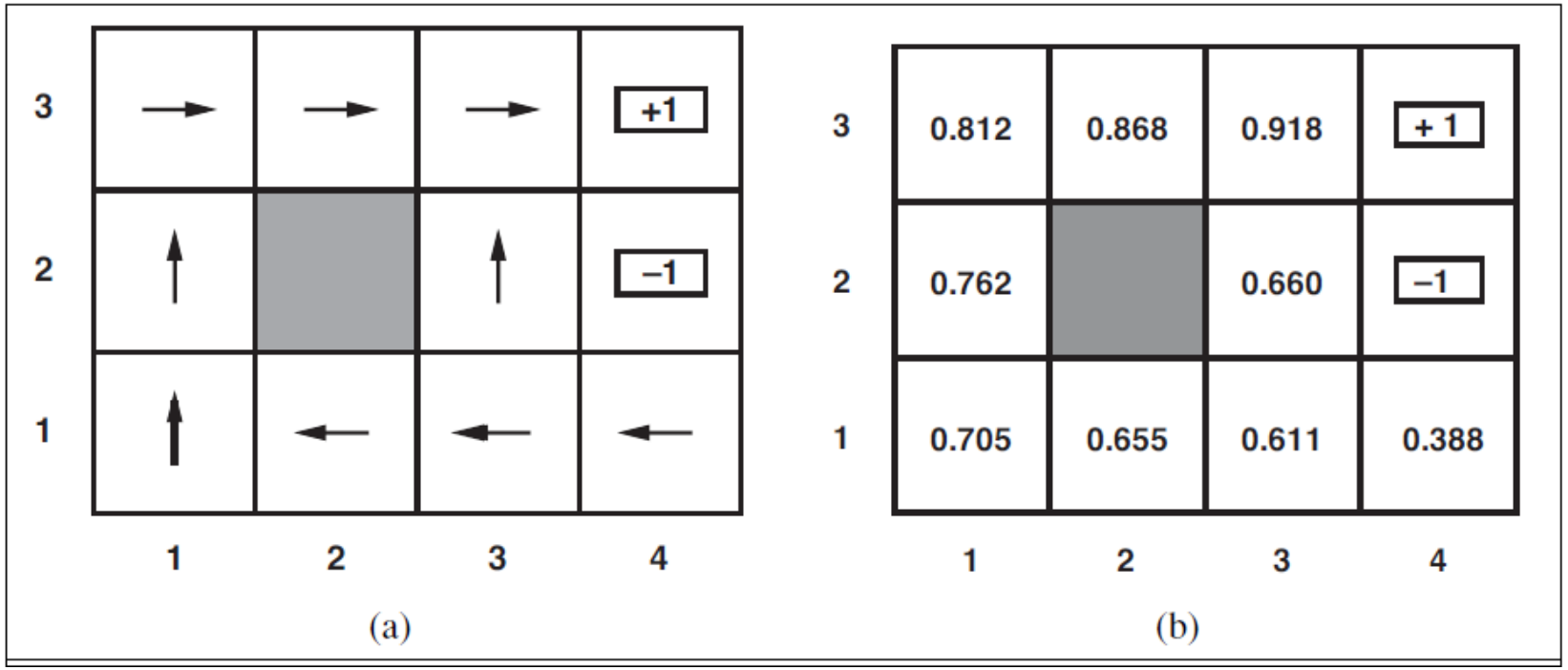
Simplified task: policy evaluation

- Input: a fixed policy $\pi(s)$
- You don't know the transition model $P(s'|s,a)$
- You don't know the rewards $R(s,a,s')$
- Goal: learn the utility values for the states.

In this case:

- No choice about what actions to take.
- Just execute the policy and learn from experience how good it is.
- This is NOT offline planning! You actually take actions in the world.

Passive Reinforcement Learning



(a) A policy π for the 4×3 world; this policy R(s)= − 0.04 in the nonterminal states and γ=1 (No discounting)

(b) The utilities of the states in the 4×3 world, given policy π.

Passive Reinforcement Learning- Example

- The agent executes a set of **trials** in the environment using its policy π.
- In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states, (4,2) or (4,3).
- The agent's percepts supply both the current state and the reward received in that state.

$$(1,1)_{\textbf{-.04}} \rightsquigarrow (1,2)_{\textbf{-.04}} \rightsquigarrow (1,3)_{\textbf{-.04}} \rightsquigarrow (1,2)_{\textbf{-.04}} \rightsquigarrow (1,3)_{\textbf{-.04}} \rightsquigarrow (2,3)_{\textbf{-.04}} \rightsquigarrow (3,3)_{\textbf{-.04}} \rightsquigarrow (4,3)_{\textbf{+1}}$$
$$(1,1)_{\textbf{-.04}} \rightsquigarrow (1,2)_{\textbf{-.04}} \rightsquigarrow (1,3)_{\textbf{-.04}} \rightsquigarrow (2,3)_{\textbf{-.04}} \rightsquigarrow (3,3)_{\textbf{-.04}} \rightsquigarrow (3,2)_{\textbf{-.04}} \rightsquigarrow (3,3)_{\textbf{-.04}} \rightsquigarrow (4,3)_{\textbf{+1}}$$
$$(1,1)_{\textbf{-.04}} \rightsquigarrow (2,1)_{\textbf{-.04}} \rightsquigarrow (3,1)_{\textbf{-.04}} \rightsquigarrow (3,2)_{\textbf{-.04}} \rightsquigarrow (4,2)_{\textbf{-1}} .$$

ehna hena bn3ml monte-carlo (random sampling) operations, w mnha bngeb el mean aw el expected value.

Passive Reinforcement Learning

- Given these trials, the agent learn the expected utility $U^{\pi}(s)$ associated with each nonterminal states.

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

- where R(s) is the reward for a state, S_t (a random variable) is the state reached at time t when executing policy π, and S_0 =s.

The utility of a state is the expected total reward from that state onward.

Direct Utility Estimation

Goal: Compute utilities for each state under π

Idea: Average together observed sample values

- Act according to π
- Every time you visit a state, write down what the sum of discounted rewards turned out to be
- Average those samples

Direct Utility Estimation

- Each trial provides a *sample* of the expected total reward for each state visited.

$$(1,1)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (2,3)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (4,3)_{+1}$$
$$(1,1)_{-.04} \rightsquigarrow (1,2)_{-.04} \rightsquigarrow (1,3)_{-.04} \rightsquigarrow (2,3)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (3,2)_{-.04} \rightsquigarrow (3,3)_{-.04} \rightsquigarrow (4,3)_{+1}$$
$$(1,1)_{-.04} \rightsquigarrow (2,1)_{-.04} \rightsquigarrow (3,1)_{-.04} \rightsquigarrow (3,2)_{-.04} \rightsquigarrow (4,2)_{-1} \ .$$

- For state (1,1), the first trial provides a sample total reward of:

  1-0.04*7=1-0.28=0.72

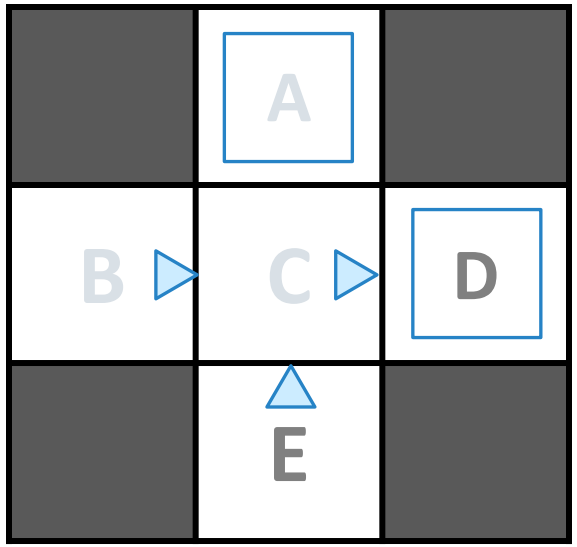
- Similarly, two samples for the state(1,2):
  - 1-0.04*6=1-0.24=0.76.
  - 1-0.04*4=1-0.16=0.84.

$$\frac{0.76 + 0.84}{2}$$

ehna bngrb kaza state, w m3 el tgroba el kter, htla2y en nfs el state tl3 leha kaza kema. fa bn3ml average baa 3lehom.

Example: Direct Utility Estimation

Input Policy π



Assume: $\gamma = 1$

Terminal states are: A and D

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

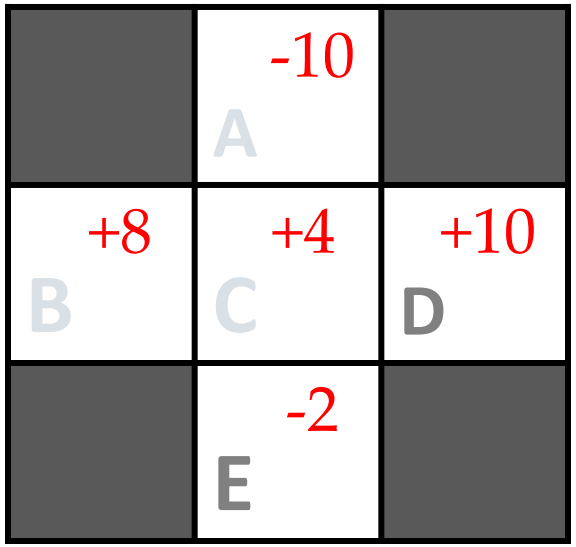
Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Output Values



If B and E both go to C under this policy, how can their values be different?

Direct Utility Estimation

- Thus, at the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly, just by keeping a running average for each state.

- In the limit of infinitely many trials, the sample average will converge to the true expectation of utility.

- It doesn't require any prior knowledge of the transition model or the reward function.

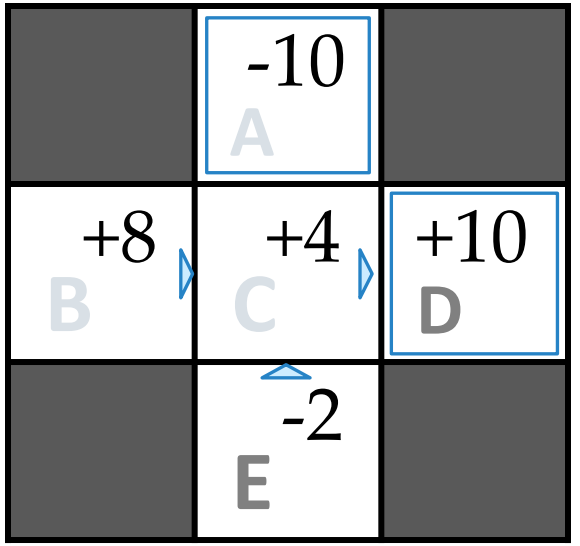
Problems with Direct Utility Estimation

- The utilities of states are not independent!
- The utility values obey the Bellman equations for a fixed policy:

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

- By ignoring the connections between states, direct utility estimation misses opportunities for learning.
- Thus, direct utility estimation often converges very slowly.

Output Values



If B and E both go to C under this policy, how can their values be different?

Model-Based Learning

Model-Based Idea:

- Learn an approximate model based on experiences
- Solve for utilities as if the learned model were correct

Step 1: Learn empirical MDP model

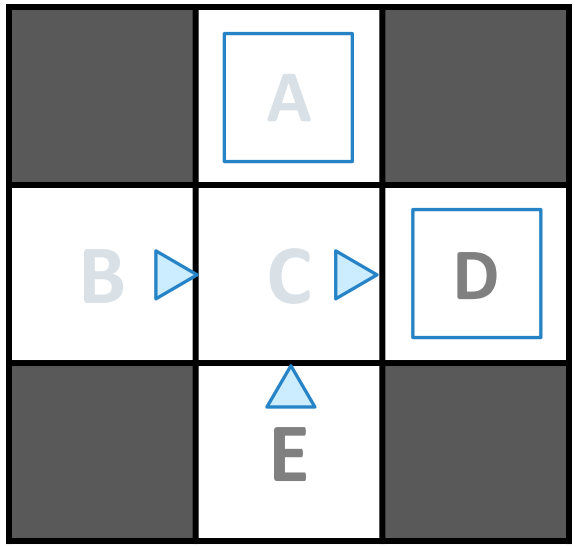
- Count outcomes s' for each s, a
- Normalize to give an estimate of P(s'|s,a)
- Discover each **R(s)** when we experience (s, a, s')

Step 2: Solve the learned MDP

- For example, use the policy evaluation algorithm.

Example: Model-Based Learning

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Learned Model

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

P(C|B, east) = 1.00
P(D|C, east) = 0.75
P(A|C, east) = 0.25
…

$\hat{R}(s,a,s')$

R(B, east, C) = -1
R(C, east, D) = -1
R(D, exit, x) = +10
…

Adaptive Dynamic Programming (ADP)

- An **adaptive dynamic programming** (or ADP) agent is an example of model-based passive reinforcement learning.
- The ADP agent is limited only by its ability to learn the transition model.

function PASSIVE-ADP-AGENT($percept$) **returns** an action
  inputs: $percept$, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    mdp, an MDP with model P, rewards R, discount γ
    U, a table of utilities, initially empty
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    $N_{s'|sa}$, a table of outcome frequencies given state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s' is new **then** $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$ and $N_{s'|sa}[s', s, a]$
    for each t such that $N_{s'|sa}[t, s, a]$ is nonzero **do**
      $P(t \mid s, a) \leftarrow N_{s'|sa}[t, s, a] \,/\, N_{sa}[s, a]$
  $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
  if s'.TERMINAL? **then** $s, a \leftarrow$ null **else** $s, a \leftarrow s', \pi[s']$
  return a

3ebo en kol iteration feha 7sabat ktera.

lakn 3dd el iterations elly m7tagha ntegt eny bst5dm DP fa hya 3ddhom msh kter.

Average Through Time

- Suppose we have a sequence of values (your sample data):
  $v_1, v_2, .., v_k$
- And want a running approximation of their expected value e.g., given sequence of grades, estimate expected value of next grade
- A reasonable **estimate** is the average of the first *k* values:

$$A_k = \frac{v_1 + v_2 + ... + v_k}{k}$$

Average Through Time

$$A_k = \frac{v_1 + v_2 + ... + v_k}{k}$$

$$kA_k = v_1 + v_2 + ... + v_{k-1} + v_k \quad (1)$$ and equivalently for k-1

$$(k-1)A_{k-1} = v_1 + v_2 + ... + v_{k-1} \quad (2)$$ which replaced in the equation above gives

$kA_k = (k-1)A_{k-1} + v_k$ Substitute from (2) in (1) and divide by k we get :

$$A_k = (1 - \frac{1}{k})A_{k-1} + \frac{v_k}{k}$$

and if we set $\alpha_k = 1/k$

$$A_k = (1 - \alpha_k)A_{k-1} + \alpha_k v_k$$

$$A_k = A_{k-1} + \alpha_k (v_k - A_{k-1})$$

New Estimate

Previous Estimate

New value

el formula de bnst5dmha fe 7agat ktera gedan zy el gradient descent msln,
el fekra kolaha en el value l gdeeda hya el value el adema + rate mdrob feh el value el adema b3d ama 3mlt 3leha operation mo3ayana. el science da gamel awi wlahy, bs mehtag w2t afhmo.

Temporal Difference Learning

- Temporal difference uses the observed transitions to adjust the utilities of the observed states so that they agree with the Bellman constraint equations.
- Update U(s) each time we experience a transition (s, a, s', r)
- Likely outcomes s' will contribute updates more often.
- Policy is still fixed, still doing evaluation!

Temporal Difference Learning

- Move utilities toward value of whatever successor occurs: running average

Sample of U(s): $sample = R(s) + \gamma U^{\pi}(s')$

- Using the running average equation:

$$A_k = (1-\alpha_k)A_{k-1} + \alpha_k v_k$$

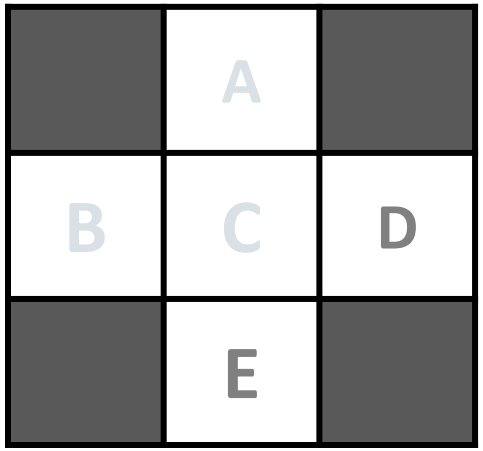
$$U^{\pi}(s) = (1-\alpha)U^{\pi}(s) + (\alpha)\,\text{sample}$$

$$U^{\pi}(s) = U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s))$$

- Because this update rule uses the difference in utilities between successive states (the new utility value versus the previous estimate of it) , it is often called the **temporal-difference.**

Example: Temporal Difference Learning

States

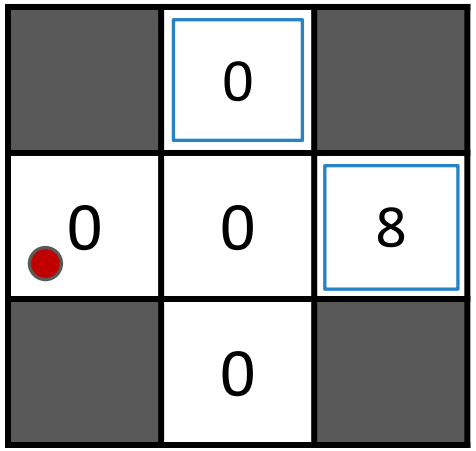


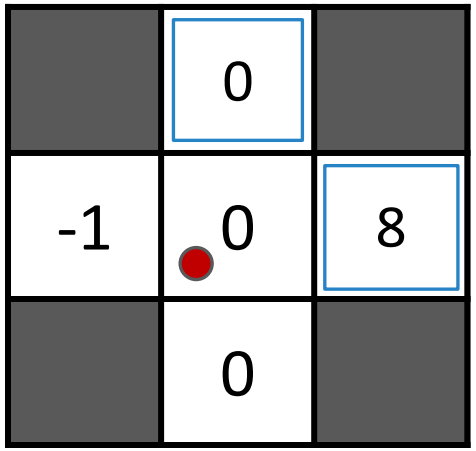
Assume: $\gamma = 1$, $\alpha = 1/2$

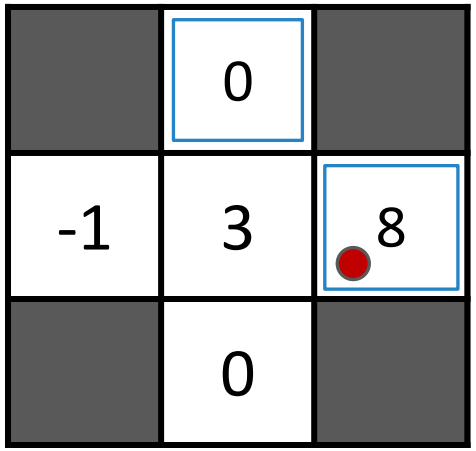
Observed Transitions

B, east, C, -2

C, east, D, -2







$$U^{\pi}(s) = U^{\pi}(s) + \alpha(sample - U^{\pi}(s))$$

Temporal Difference Learning

- The temporal difference (TD) update involves only the observed successor whereas the actual equilibrium conditions involve all possible next states.

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

- However, the temporal difference converges since the observed samples are actually drawn from the transition model P(s'|s,a).
- If we change α from a fixed parameter to a function that decreases as the number of times a state has been visited increases, then $U^{\pi}(s)$ itself will converge to the true value.

$$U^{\pi}(s) = U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s))$$

Problems with TD Learning

- TD learning is a model-free way to do policy evaluation, mimicking Bellman updates with running sample averages
- However, we cannot turn values into a (new) policy since the transition model is not known!

$$\pi(\mathrm{s}) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s'|s,a)U(s')$$

Active Reinforcement Learning

- A passive learning agent has a fixed policy and applies it.
- An active agent must decide what actions to take.

- We will consider:
  - Q-learning
  - Sarsa
  - Active ADP

Q-Learning

- **Q-learning** learns an action-utility representation instead of learning utilities.
- Q(s, a) denotes the utility value of doing action a in state s.
- Recall the Bellman equation:

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

- Q-values are directly related to utility values as follows:

$$U(s) = \max_{a} Q(s, a)$$

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s' \mid s, a) \max_{a'} Q(s', a') .$$

Q-Learning

Q-Learning: sample-based Q-value iteration

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

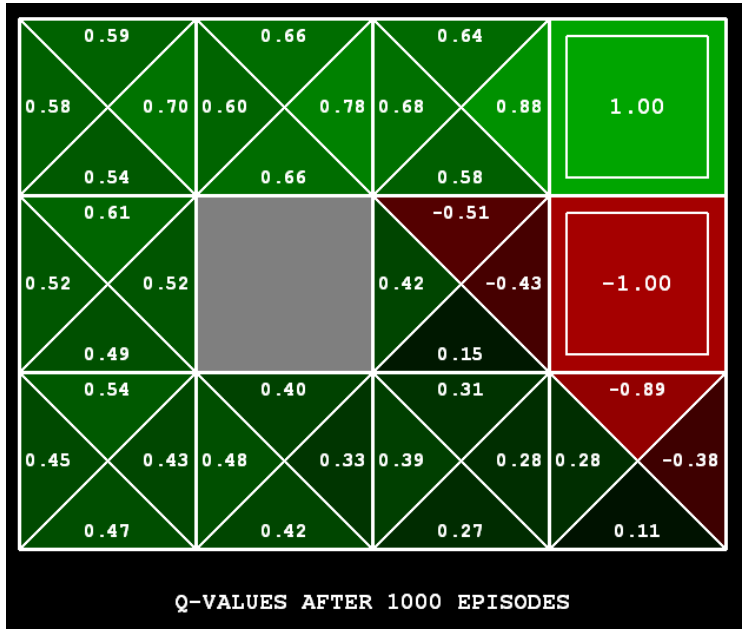
Learn Q(s,a) values as you go

- Receive a sample (s,a,s',r)
- Consider your old estimate: $Q(s,a)$
- Consider your new sample estimate:

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

- Incorporate the new estimate into a running average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$



0D3)]

Q-Learning

- A Temporal Difference agent that learns a Q-function does not need a transition model P(s'| s, a), neither for learning nor for action selection.
- Thus, Q-learning is a **model-free** method.

- Qlearning calculates Q values, then how the agent should act?

Exploration vs. Exploitation

- Q-learning computes a Q-function Q(s,a) that allows the agent to see, for every state, which is the action with the highest discounted expected reward.
- Given a Q-function the agent can :
  - **Exploit** the knowledge accumulated so far, and choose the action that maximizes Q(s,a) in a given state (***greedy behavior***).

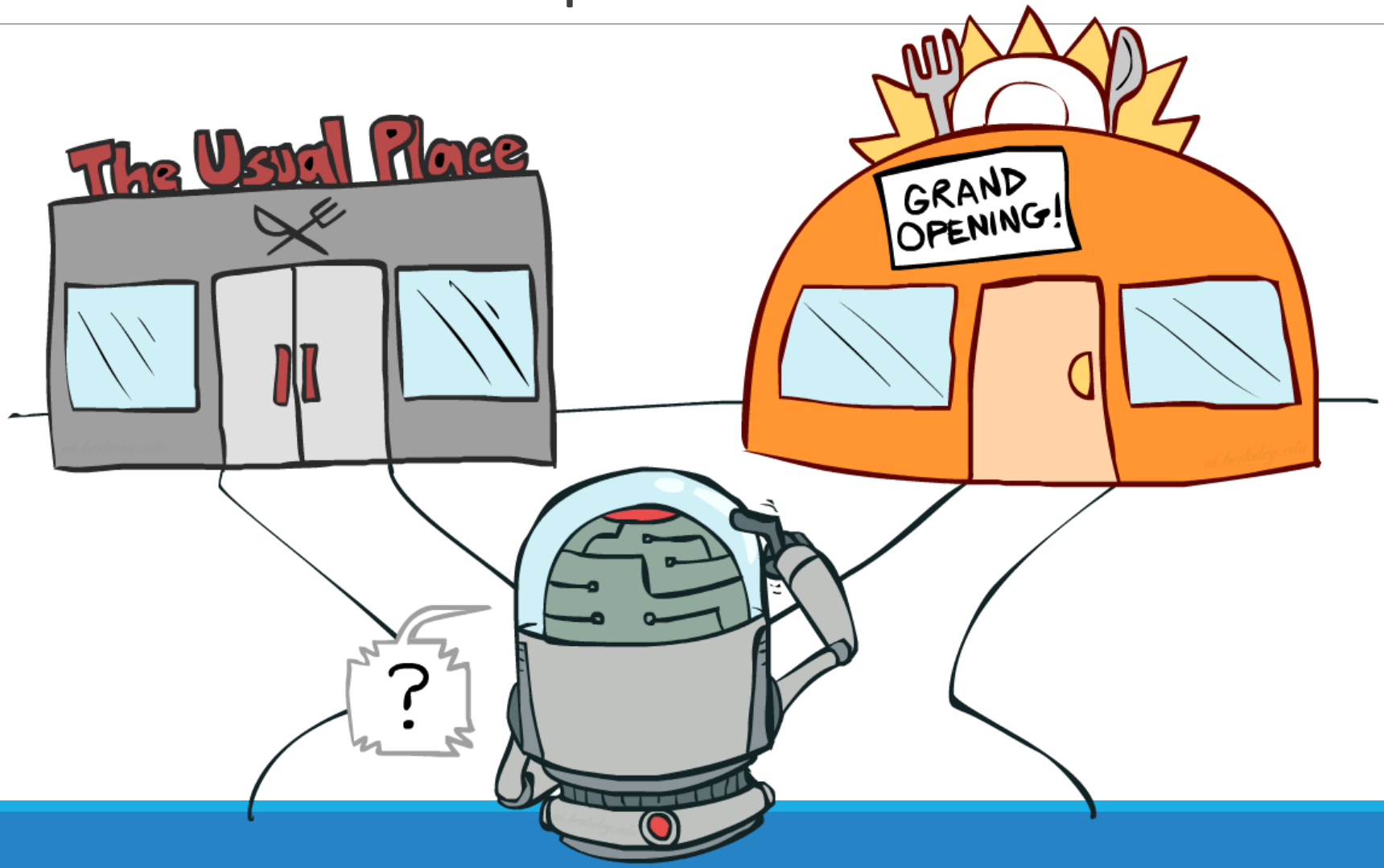
$$\pi(\mathrm{s}) = \underset{a \in A(s)}{\operatorname{argmax}}\ Q(s,a)$$

  - **Explore** new actions, hoping to improve its estimate of the optimal Q-function, i.e. (do not choose the action suggested by the current Q(s,a))

Exploration vs. Exploitation

- Actions not only provide rewards, but also they contribute to learning the true model by affecting the percepts that are received.
- By improving the model, the agent will receive greater rewards on the long term.
- There is a tradeoff between **exploitation** and **exploration.**
- Exploration (Take actions that have not been tried much in a state to improve the agent's learning model (Qvalues)).
- Exploitation(Take actions whose estimate will yield high discounted expected reward).

Exploration vs. Exploitation



Exploration

- The learned model is not the same as the true environment; what is optimal in the learned model can therefore be suboptimal in the true environment.
- Unfortunately, the agent does not know what the true environment is, so it cannot compute the optimal action for the true environment.
- Thus, the agent needs to explore other actions!

Exploration vs. Exploitation

- When to explore and when the exploit?
  - Never exploring may lead to being stuck in a suboptimal course of actions
  - Exploring too much is a waste of the knowledge accumulated via experience

- Must find the right compromise!

Exploration Strategies

- It is very hard to obtain an *optimal* exploration method. However, it is possible to come up with a *reasonable* scheme that will eventually lead to optimal behavior by the agent.
- Technically, any such scheme needs to be **greedy in the limit of infinite exploration (GLIE).**
- A **GLIE** scheme must do the following:
  1. **Must try each action in each state an unbounded number of times** to avoid having a finite probability that an optimal action is missed because of an unusually bad series of outcomes.
  2. **Must eventually become greedy**, so that the agent's actions become optimal with respect to the learned (and hence the true) model.
- We will look at two exploration strategies
  - ε-greedy
  - Exploration functions.

Epsilon (ε)-greedy

- ε-greedy takes random actions as follows:
  - Every time step:
  - With (small) probability ε, act randomly.
  - With (large) probability 1-ε, choose the best action according to the learned Q values.

- Problems with random actions?
  - You do eventually explore the space, but you do not exploit what you have learned!
  - One solution: lower ε over time, for example (ε=1/t)
  - Another solution: exploration functions

Exploration Functions

When to explore?

- Explore areas whose badness is not (yet) established, eventually stop exploring.

Exploration function

- Takes a value estimate u and a visit count N, and returns an optimistic utility
- Note: this propagates the “bonus” back to states that lead to unknown states as well!

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s' \mid s,a)U^+(s'),\ N(s,a)\right)$$

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

where R^+ is an optimistic estimate of the best possible reward obtainable in any state and N_e is a fixed parameter. This will have the effect of making the agent try each action–state pair at least N_e times.

Q-learning

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
        $S \leftarrow S'$
    until S is terminal

Q-Learning Properties

Q-learning converges to optimal policy -- even if you're acting suboptimally!

This is called off-policy learning

- You have to explore enough.
- You have to eventually make the learning rate small enough.
- … but not decrease it too quickly.
- Basically, in the limit, it doesn't matter how you select actions (!)

State-Action-Reward-State-Action (Sarsa)

Sarsa is another temporal difference active reinforcement learning algorithm.

- Sarsa Q update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma\, Q(s',a') - Q(s,a))$$

where a is the action *actually taken* in state s.

No max operator.
You evaluate Qvalue of the action you actually take!

- The rule is applied at the end of each (s, a, r, s', a'), so it is named (SARSA).
- SARSA waits until an action is actually taken and backs up the Q-value for that action.
- Q-learning update:
- Q-learning backs up the *best* Q-value from the state reached in the observed transition.

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

Sarsa

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize S
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Loop for each step of episode:
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ε-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma Q(S', A') - Q(S, A)\right]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until S is terminal

Qlearning vs. SARSA

- Q-learning uses the best Q-value, it pays no attention to the actual policy being followed.
- Q learning is an **off-policy** learning algorithm.
- However, SARSA is an **on-policy** algorithm.
- Q-learning is more flexible than SARSA, as a Q-learning agent can learn how to behave well

even when guided by a random or adversarial exploration policy.

- On the other hand, SARSA is more realistic.
- For a greedy agent that always takes the action with best Q-value, the two algorithms

are identical, however, when exploration is happening, they differ significantly.

Active Model-based Reinforcement learning

- We can turn the passive model-based reinforcement learning (ADP) into an active one as follows:

1. Estimate the MDP model parameters given observed transitions and rewards.
2. Use the estimated MDP to compute estimate of optimal values and policy using value iteration or policy iteration algorithms
3. Apply any GILE exploration strategy (greedy or exploration function) since the learned model is not the true model!
   - At each time step:
     - Either act randomly according to the exploration strategy (e.g. ε-greedy)
     - Or exploit using the learned utility values from the model

$$\text{A}= \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s'|s,a)U(s')$$

Model-based versus Model free Reinforcement Learning

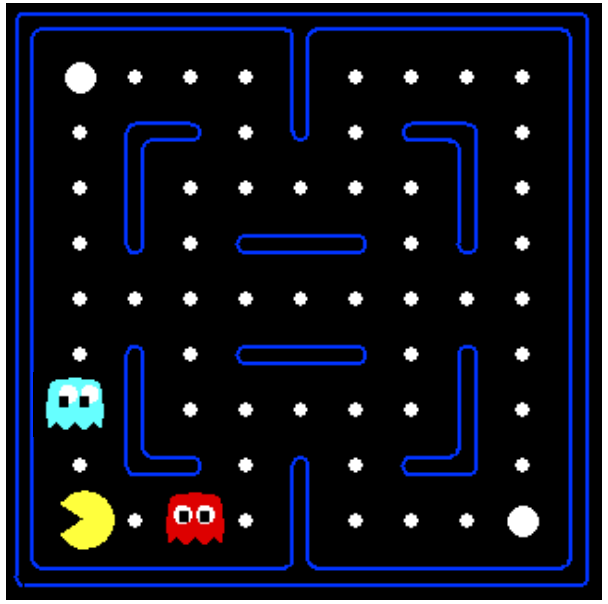
- Is it better to learn a model and a utility function or an action value function with no model?
  - This is still an open-question
- Model-based approaches require less data to learn well, but they can be computationally more expensive.
- Q-learning takes longer time to converge because it does not enforce consistency among Q-values via the model, but they are less computationally expensive.

Generalizing Across States

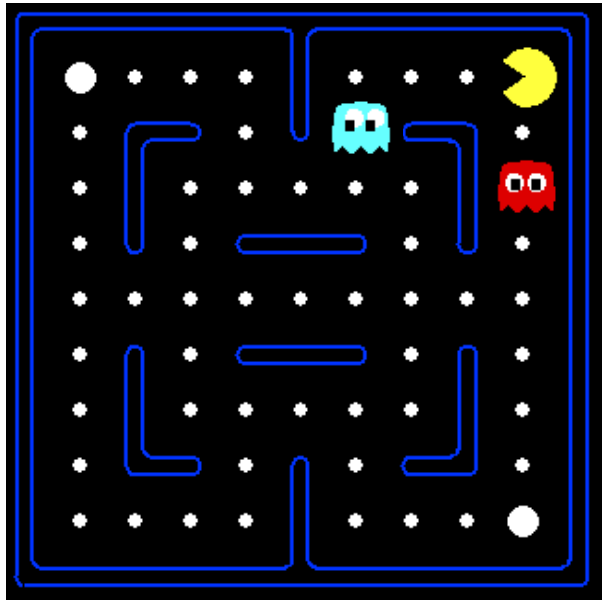
- Basic Q-Learning keeps a table of all q-values for all states/states-actions pairs.
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training, for Backgammon and chess, the state space if of order
  - 10^{20} and 10^{40}.
  - Too many states to hold the q-tables in memory
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar situations.
  - This is a fundamental idea in machine learning!

Example: Pacman

Let's say we discover through experience that this state is bad:

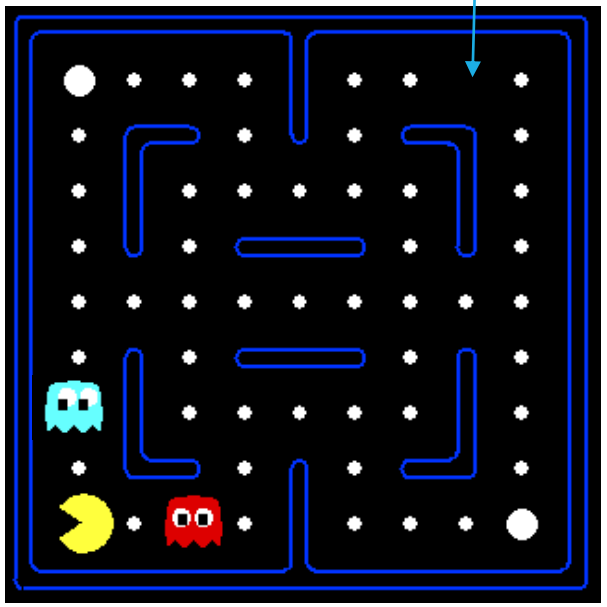


In naïve q-learning, we know nothing about this state:



Or even this one!

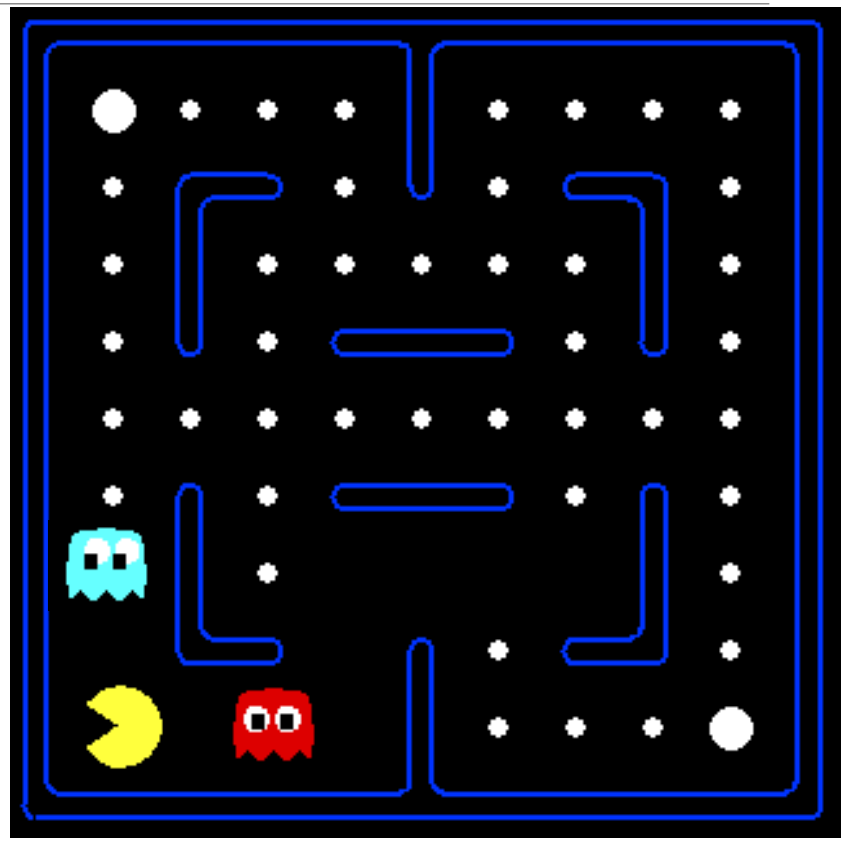
Only one dot has changed!



Feature-Based Representations

Solution: describe a state using a vector of features (properties)

- Features are functions from states to real numbers (often 0/1) that capture important properties of the state
- Example features:
  - Distance to closest ghost
  - Distance to closest dot
  - Number of ghosts
  - 1 / (dist to dot)^2
  - Is Pacman in a tunnel? (0/1)
  - …… etc.
  - Is it the exact state on this slide?
- Can also describe a q-state (s, a) with features (e.g. action moves closer to food)



Linear Value Functions

- Using a feature representation, we can write a q function (or value function) for any state using a few weights:

$$U(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

- Our experience is summed up in a few powerful numbers!

- However, some states may share features but actually be very different in value!

Approximate Q-Learning

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

transition $= (s, a, r, s')$

Q-learning with linear Q-functions:

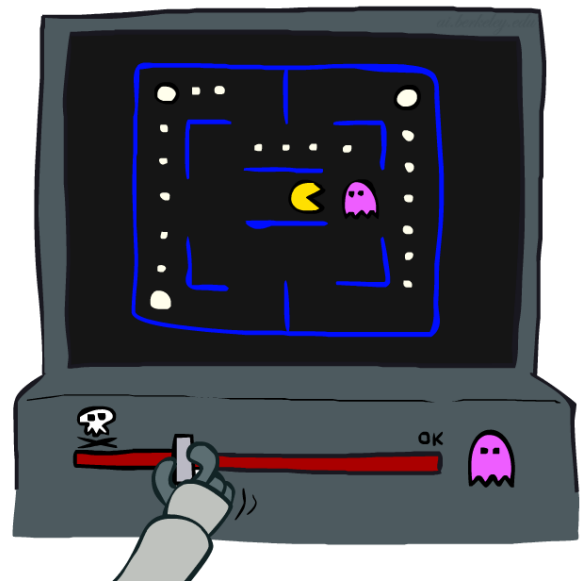
$$\text{difference} = \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s,a)$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha \,[\text{difference}]$$

$$w_i \leftarrow w_i + \alpha \,[\text{difference}]\, f_i(s,a)$$

Exact Q's

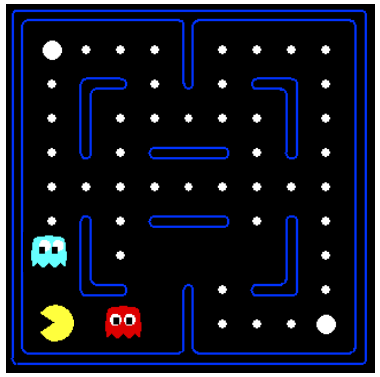
Approximate Q's



Example: Q-Pacman

$$Q(s,a) = 4.0 f_{DOT}(s,a) - 1.0 f_{GST}(s,a)$$

s



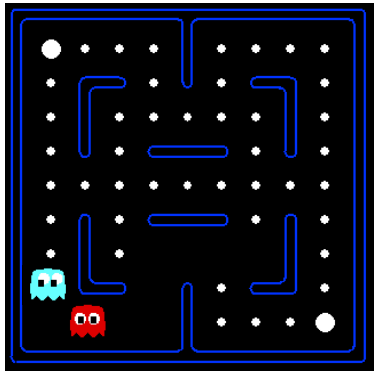
$f_{DOT}(s, \text{NORTH}) = 0.5$

$f_{GST}(s, \text{NORTH}) = 1.0$

$Q(s, \text{NORTH}) = +1$

$a = \text{NORTH}$
$r = -500$

s'



$Q(s', \cdot) = 0$

$$r + \gamma \max_{a'} Q(s', a') = -500 + 0$$

difference $= -501$

$$w_{DOT} \leftarrow 4.0 + \alpha\,[-501]\,0.5$$
$$w_{GST} \leftarrow -1.0 + \alpha\,[-501]\,1.0$$

$$Q(s,a) = 3.0 f_{DOT}(s,a) - 3.0 f_{GST}(s,a)$$

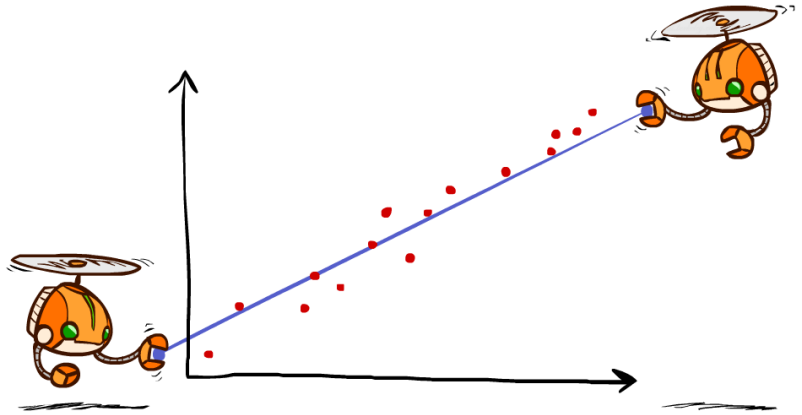
Minimizing Least Squares Error

Imagine we had only one point x, with features f(x), target value y, and weights w:

$$\text{error}(w) = \frac{1}{2}\left(y - \sum_k w_k f_k(x)\right)^2$$

$$\frac{\partial\ \text{error}(w)}{\partial w_m} = -\left(y - \sum_k w_k f_k(x)\right) f_m(x)$$

$$w_m \leftarrow w_m + \alpha\left(y - \sum_k w_k f_k(x)\right) f_m(x)$$



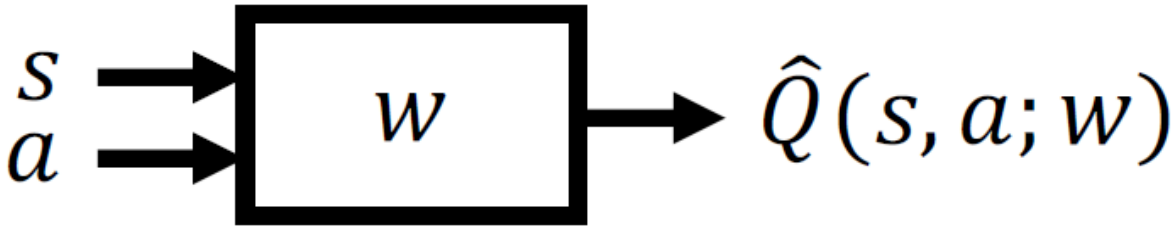
Approximate q update explained:

$$w_m \leftarrow w_m + \alpha\left[r + \gamma \max_a Q(s', a') - Q(s, a)\right] f_m(s, a)$$

"target" "prediction"

Approximate Qlearning

- There are many possible function approximators, the most popular ones are:
  - Linear combinations of features
  - Neural networks



- The representation is viewed as approximate because it might not be the case that the *true* utility function or Q-function can be represented in the chosen form.

Function Approximation

Advantages:

- Function approximation makes it practical to represent utility functions for very large state spaces.
- Function approximation allows the learning agent to generalize from states it has visited to states it has not visited.

Disadvantage:

- There could fail to be any function in the chosen hypothesis space (function representation) that approximates the true utility function sufficiently well.

Policy Search

- Policy search considers *parameterized* representations of the **policy π** that have far fewer parameters θ than there are states in the state space.
- However, the policy is a *discontinuous* function of the parameters when the actions are discrete.
- Thus, gradient-based search would be hard to be applied.
- Consequently, policy search methods often use a **stochastic policy** representation $\pi_\theta(s, a)$, which specifies the *probability* of selecting action a in state s.

$$\pi_\theta(s, a) = \mathbb{P}[a|s; \theta]$$

Policy Search

- The goal is to find a policy that maximizes the policy value $\rho(\theta)$. (optimization problem)
- The **policy value** $\rho(\theta)$ is defined as the expected reward-to-go (utility) when $\pi\theta$ is executed.

- This optimization problem can be solved using:
  - Gradient based methods if $\rho(\theta)$ is differentiable.
  - Local Search methods such as: hill climbing

Policy Search

- One popular representation for the policy is the **softmax function.**
- The softmax-based policy becomes nearly deterministic if one action is much better than the others.
- The softmax function gives a differentiable function of θ; hence, the value of the policy ρ(θ) is a differentiable function of θ.

$$\pi_\theta(s,a) = e^{\hat{Q}_\theta(s,a)} / \sum_{a'} e^{\hat{Q}_\theta(s,a')}$$

Applications of Reinforcement Learning

- Games
- Robotics
- Self-driving cars

Summary

- Reinforcement Learning
- Passive (Direct utility estimation, ADP, TD) vs. Active reinforcement learning (Active ADP, Qlearning, Sarsa)
- Model based (ADP) vs. Model free methods (TD, Qlearning, Sarsa)
- Exploration vs. Exploitation
- On-Policy (Sarsa) versus Off-Policy (Q-learning) learning.
- Function approximation
- Policy Search