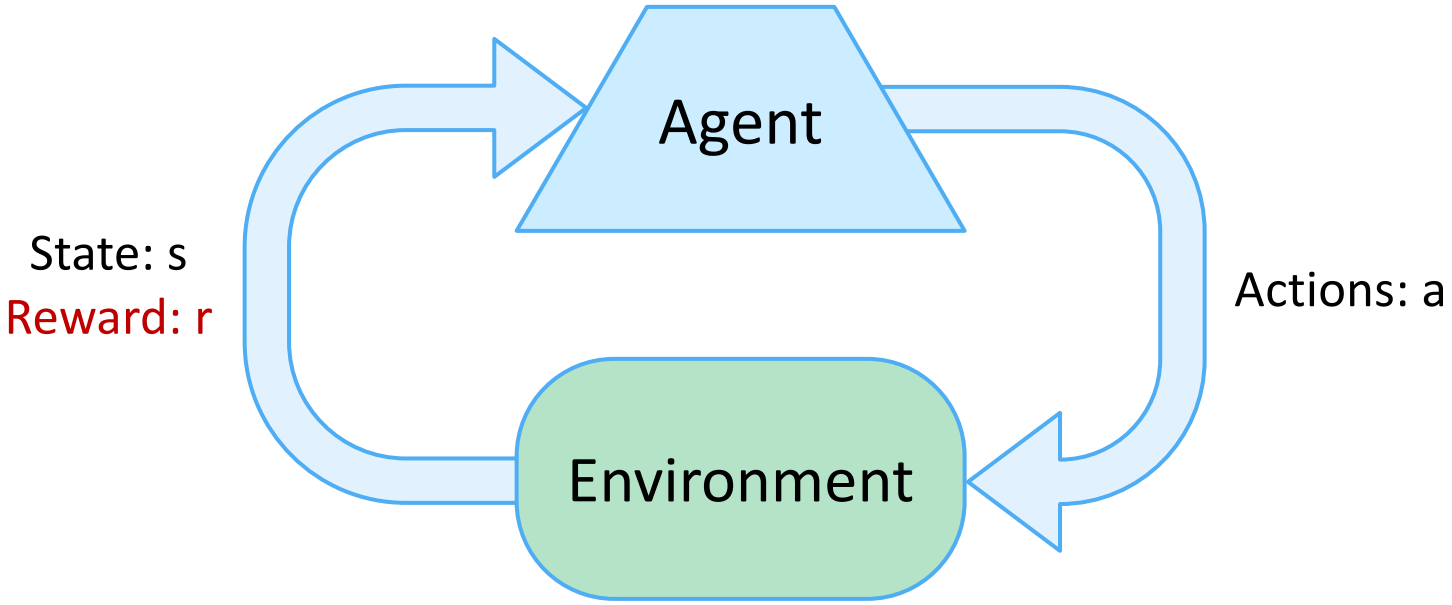
Reinforcement learning

Devika Subramanian

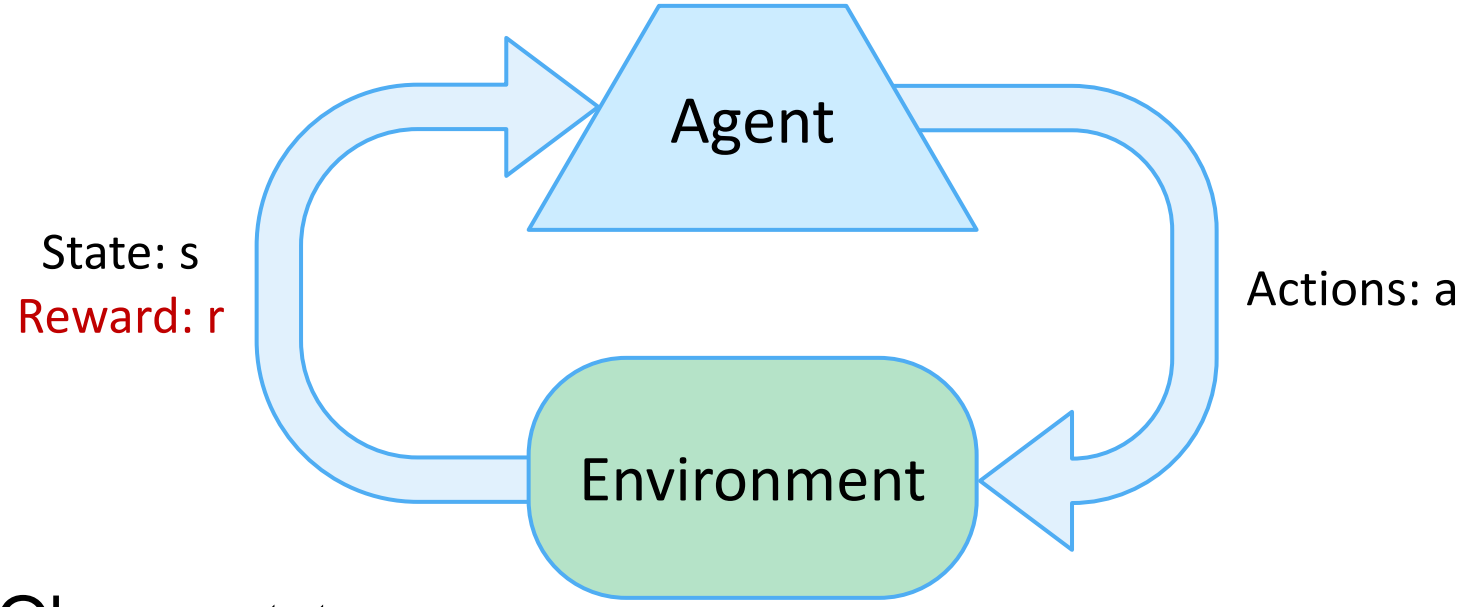
Derived from P. Abbeel and D. Klein lectures

The 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!

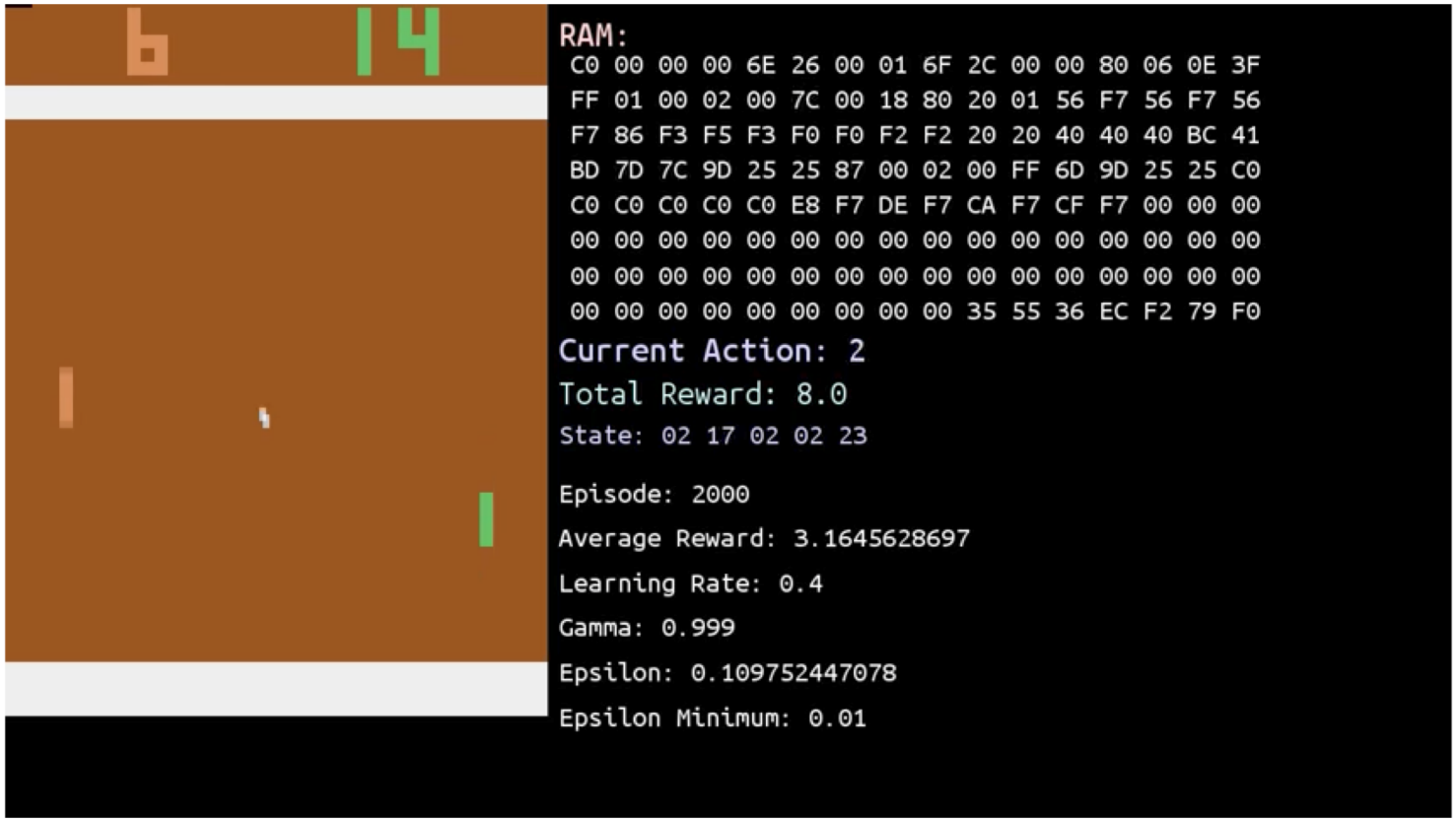
Reinforcement learning



- Observe state s
- Select action a = $\pi(s)$ [$\max_a$ Q(s,a)]
- Execute action a, environment transitions to state s', receive reward r
- Update policy π [or V(s) or Q(s,a)]

Example: learning to play Pong

- https://www.youtube.com/watch?v=PSQt5KGv7Vk

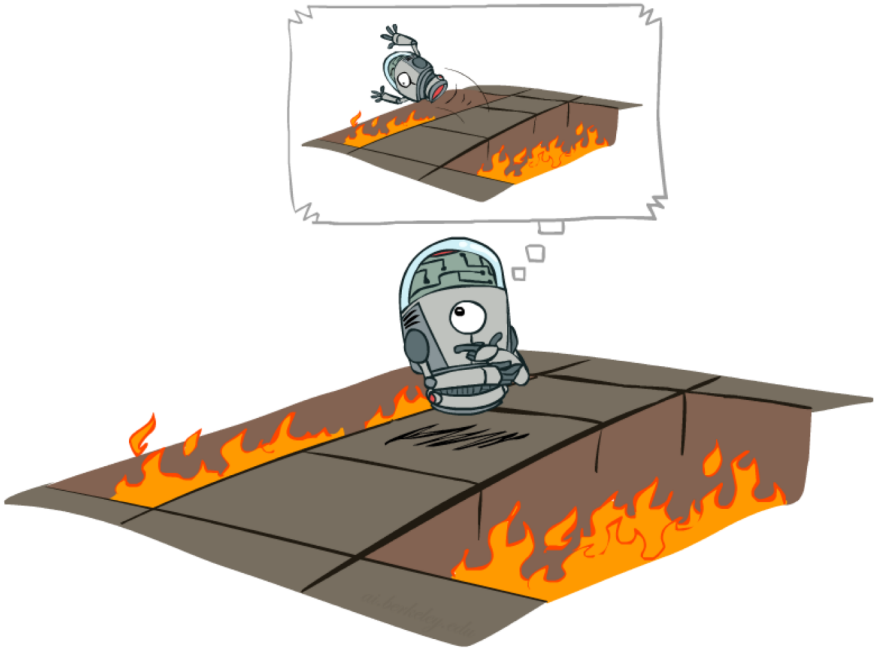


Markov decision processes and RL

- Recall Markov decision process (MDP):
  - A set of states $s \in S$ (of the environment)
  - A set of actions (per state) A (of the agent)
  - A model T(s,a,s') (state transition model)
  - A reward function R(s,a,s')
- Find a policy $\pi(s)$ to optimize long run expected sum of rewards

- New twist: we do not know T or R
  - i.e. we don't know which states are good or what the actions do
  - Must actually try actions and states out to learn

Offline (MDPs) and online (RL)



Offline Solution

Online Learning

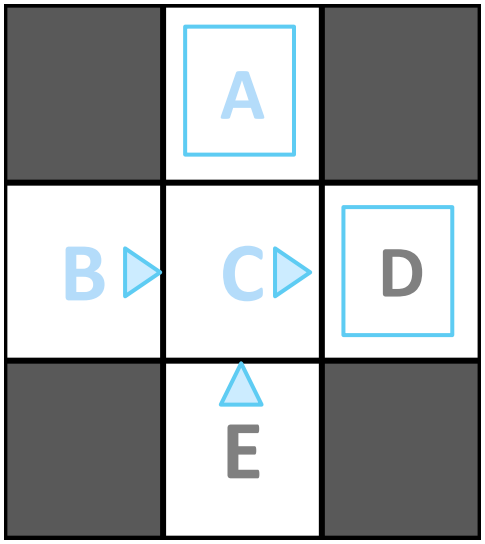
Model-based RL

- Learn an approximate model for T and R based on sequence of (s,a,s',r) tuples from each episodic interaction with the environment.
  - Count outcomes s' for each (s,a)
  - Normalize to get an estimate of T(s,a,s')
  - Maintain a running average of r associated with each (s,a,s') encountered to get an estimate of R(s,a,s')
- Solve for optimal π using the learned estimates for T and R.
- Two phase approach
  - Training phase to get T and R estimates
  - Solve using value or policy iteration

Example: model based RL

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

$\hat{T}(s, a, s')$

T(B, east, C) = 1.00
T(C, east, D) = 0.75
T(C, east, A) = 0.25
...

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

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

Model-free learning

- Passive reinforcement learning
  - Input: a fixed policy π
  - Goal: learn V_π(s)

- Learner has no choice on actions.
- Just executes the given policy and learns long run value of each state s for that policy.
- Not offline planning, learner interacts in the world to learn V_π

LMS algorithm

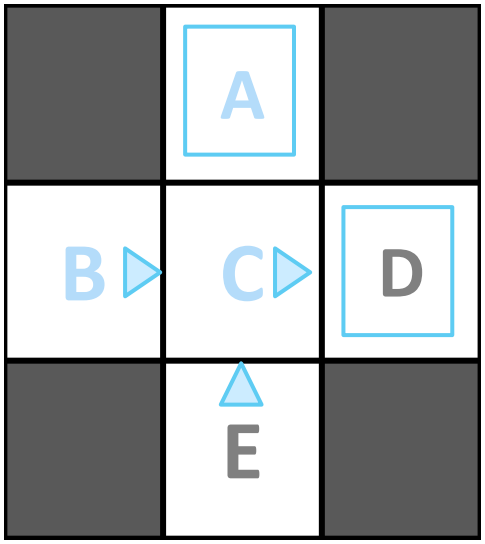
- Gather sequences of episodic interactions from the world according to given policy π
- LMS algorithm: For each sequence, for each state s in sequence, maintain a running average of the discounted sum of rewards of the sequence from s to end of sequence.

Example: LMS

Observed Episodes (Training)

Input Policy

π



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

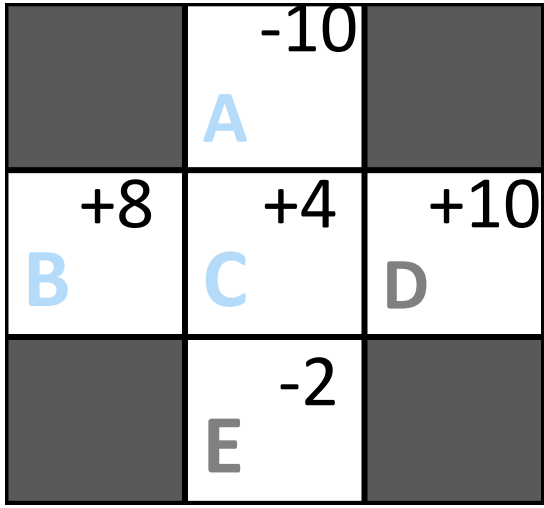
Episode 3

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

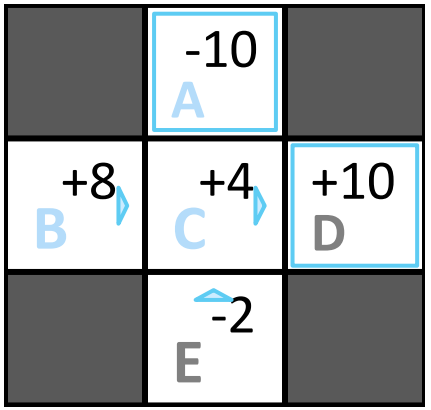


Assume: $\gamma = 1$

Problems with LMS

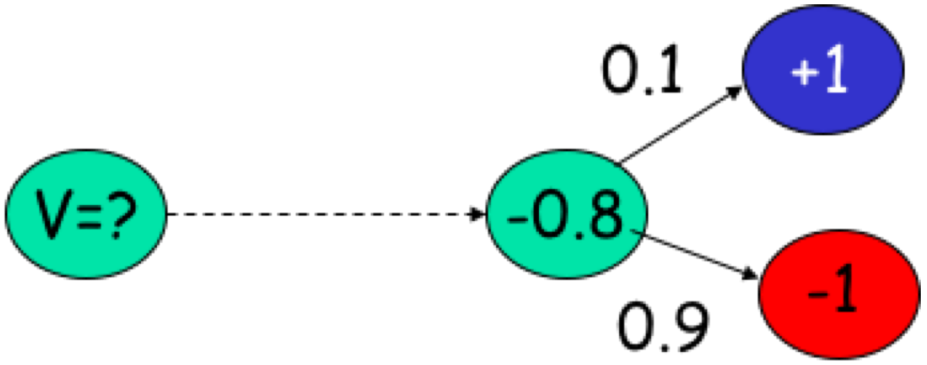
- What is good about LMS
  - Easy to understand
  - Does not require estimation of T and R
  - Eventually computes correct V_pi(s) values using sample transitions
- What is bad about LMS
  - Very inefficient: learns each state separately
  - Fails to take advantage of state connections
  - Takes a very long time to converge

Output Values



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

Problem with LMS



- If a new state is reached for the first time and the sequence ends up in the state with reward + 1, LMS will assign V of this state to be + 1. It will not take the transition into the state with utility –0.8 into account and will need to see many many examples before realizing that the utility of the new state is also –0.8

A fix to LMS using Bellman updates

- In each round, update V using a one-step-look-ahead over V

$$V_0^\pi(s) = 0$$

$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$

- This approach takes advantage of the transitions between states; but we need to know T and R
- How can we update V without knowing T and R?

Temporal difference learning

- Learn from every experience (s,a,s',r)
  - Update V(s) not at end of a sequence, but on every transition
  - Likely outcomes s' will contribute updates more often
- Move V(s) to r + (discounted) value of observed successor state s': i.e., $r + \gamma V^{\pi}(\text{s'})$

- $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha(r + \gamma V^{\pi}(\text{s'}))$

Exponential moving average

- The interpolation update of $V^{\pi}(s)$ using learning rate α constructs an exponential moving average of $V^{\pi}(s)$.

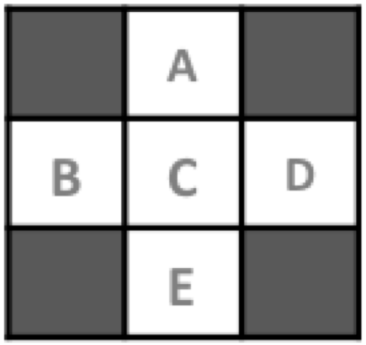
- Makes recent samples more important, and forgets about distant past values.

$$\bar{x}_n = (1-\alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n$$

$$\bar{x}_n = \frac{x_n + (1-\alpha) \cdot x_{n-1} + (1-\alpha)^2 \cdot x_{n-2} + \dots}{1 + (1-\alpha) + (1-\alpha)^2 + \dots}$$

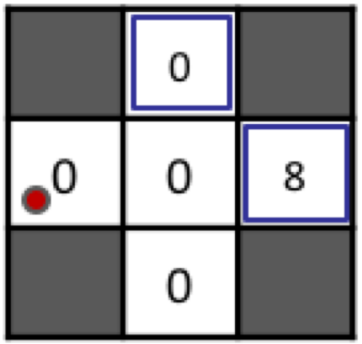
Example: temporal difference learning

States

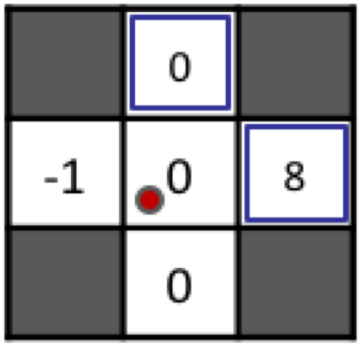


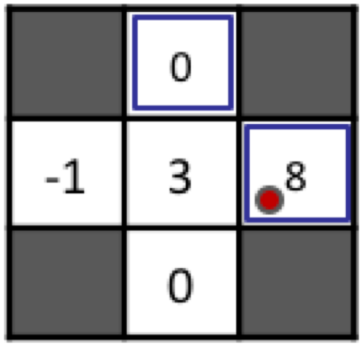
Observed Transitions

B, east, C, -2



C, east, D, -2





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

$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^{\pi}(s')\right]$$

Problems with TD learning

- TD value leaning is a model-free way to do policy evaluation, mimicking Bellman updates with running sample averages.
- However, if we want to turn values into a (new) policy, we're sunk:

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

$$Q(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V(s') \right]$$

- Idea: directly learn Q(s,a) rather than V(s); this makes action selection model-free too!

Active reinforcement learning

- Full reinforcement learning: optimal policies (like value iteration)
  - No knowledge of transitions T(s,a,s')
  - No knowledge of rewards R(s,a,s')
  - Agent chooses actions now (as opposed to the fixed policy in passive reinforcement learning)
  - Goal: learn the optimal policy / values

- In this case:
  - Learner makes choices!
  - Fundamental tradeoff: exploration vs. exploitation
  - This is NOT offline planning! The agent actually takes actions in the world and finds out what happens…

Q-learning (TD learning of Q(s,a) values)

- Q(s,a) = 0 for all states s and actions a from state s
- Update Q(s,a) as agent interacts with world
  - Obtain tuple (s,a,s',r) after choosing a in state s
  - New estimate of Q(s,a) would be $r + \gamma \max_{a'} Q(s', a')$
  - Incorporate new estimate into running average
  - $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$
  - https://www.youtube.com/watch?v=AMnW-OsOcI8

Active reinforcement learning

- Initialize Q(s,a) = 0 for all s,a
- Loop forever
  - Observe state s
  - Select action $a = \max_a Q(s, a)$
  - Execute action a, environment transitions to state s', receive reward r
  - Update Q(s,a)
    - $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$

Properties of Q-learning

- Q-learning converges to the optimal policy, even if agent acts sub-optimally. This is called off-policy learning.
- Caveats
  - You have to explore all of the state space; visiting each state a minimum number of times to get a robust estimate of Q(s,a)
  - You have to anneal the learning rate α over time.

Summary: MDPs and RL

- Known MDP
  - Solve for optimal V*, Q* and pi* offline by value or policy iteration
  - Execute pi* for ever
- Unknown MDP: Model-based solution
  - Estimate T and R by exploring state space randomly.
  - Solve for optimal policy pi* by using estimated T and R.
  - Execute pi* for ever
- Unknown MDP: Model-free passive learning
  - Directly estimate V(s) or Q(s,a) by following a fixed policy pi.
  - Use TD learning to speed up convergence.
  - Choice of pi immaterial to (eventual) convergence to optimal V* and Q*
- Unknown MDP: Model-free active learning
  - Directly estimate Q(s,a) and choose action according to Q.
  - Use TD learning to speed up convergence
  - Eventual convergence to optimal Q*

Exploration vs Exploitation

- How do we choose actions during the learning phase? Do we go with actions that are known to work well from past experience (exploit) or do we try out new actions hoping that they may yield bigger rewards (explore)?
- Lots of theory in the area called "bandit problems" in statistics.

Exploration vs Exploitation

- Several schemes for forcing exploration
  - Random actions (epsilon-greedy)
    - Every time step, flip a coin
    - With small probability epsilon, act randomly
    - With probability (1-epsilon), pick best action dictated by current Q
  - Problems with epsilon-greedy policy
    - Eventually agent explores the space, but thrashes around once learning has converged
    - A solution: lower epsilon over time

Generalizing across states

- Basic Q-Learning keeps a table of all Q values
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - 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

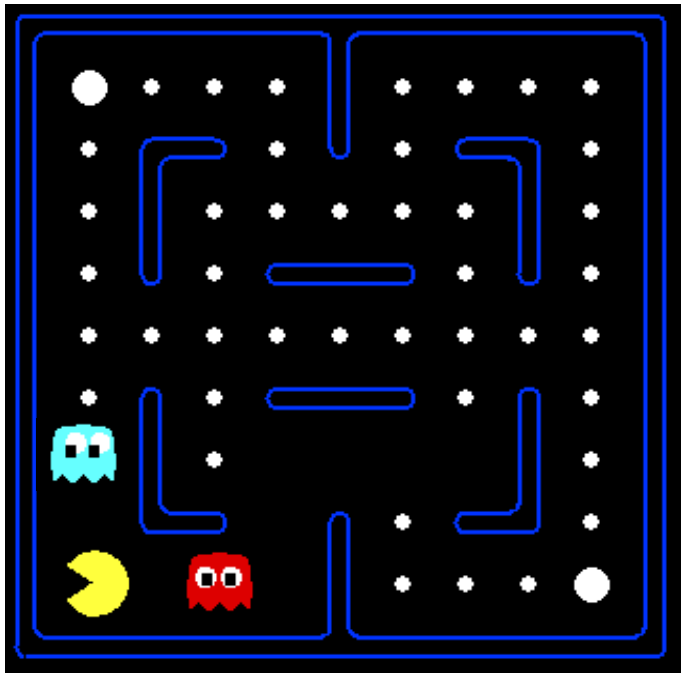
Example: why we need generalization

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

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

Feature-based representations

- Solution: describe a state using a vector of features
  - 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.
  - Can also describe Q(s,a) with these features



Linear Value functions

- Using a feature representation, we can write the Q function (or value function) for any state, action pair (resp. state) as a weighted sum:

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

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

- Advantage: our experience is summed up in a few numbers w.

- Disadvantage: states may share features but actually be very different in value! (good representations matter!)

Approximate Q learning

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

- Q-learning with linear Q-functions:

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

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

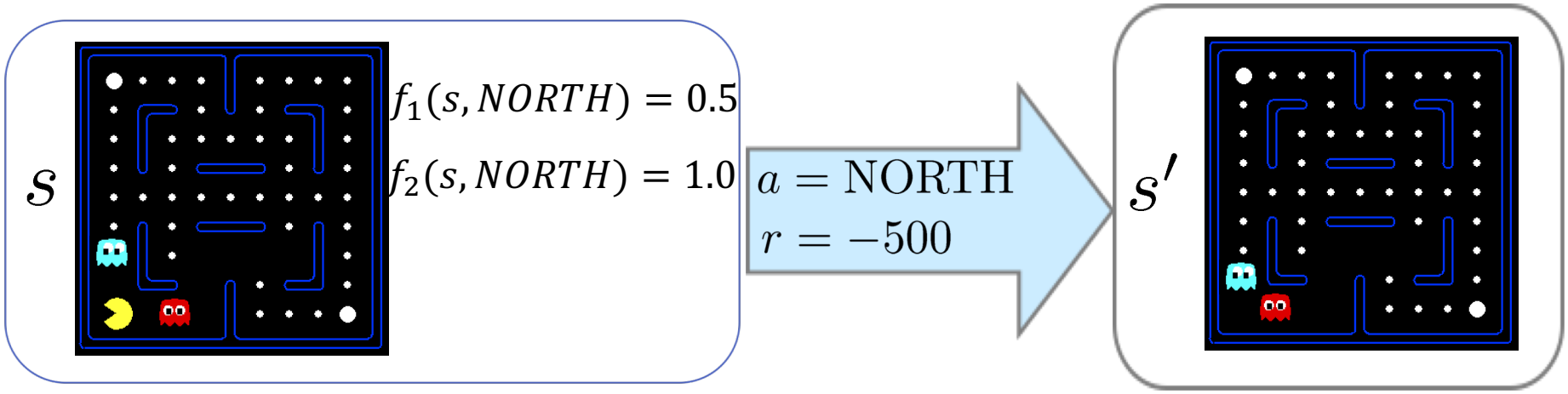
  $Q(s,a) \leftarrow Q(s,a) + \alpha$ [difference] Exact Q representation

  $w_i \leftarrow w_i + \alpha$ [difference] $f_i(s,a)$ Approx Q representation (least squares update)

- Intuitive interpretation:
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the features that were on in that state: downgrade all states with that state's features
- Formal justification: online least squares

Example: Q learning in Pacman

$$Q(s,a) = 4.0f_1(s,a) - 1.0f_2(s,a)$$



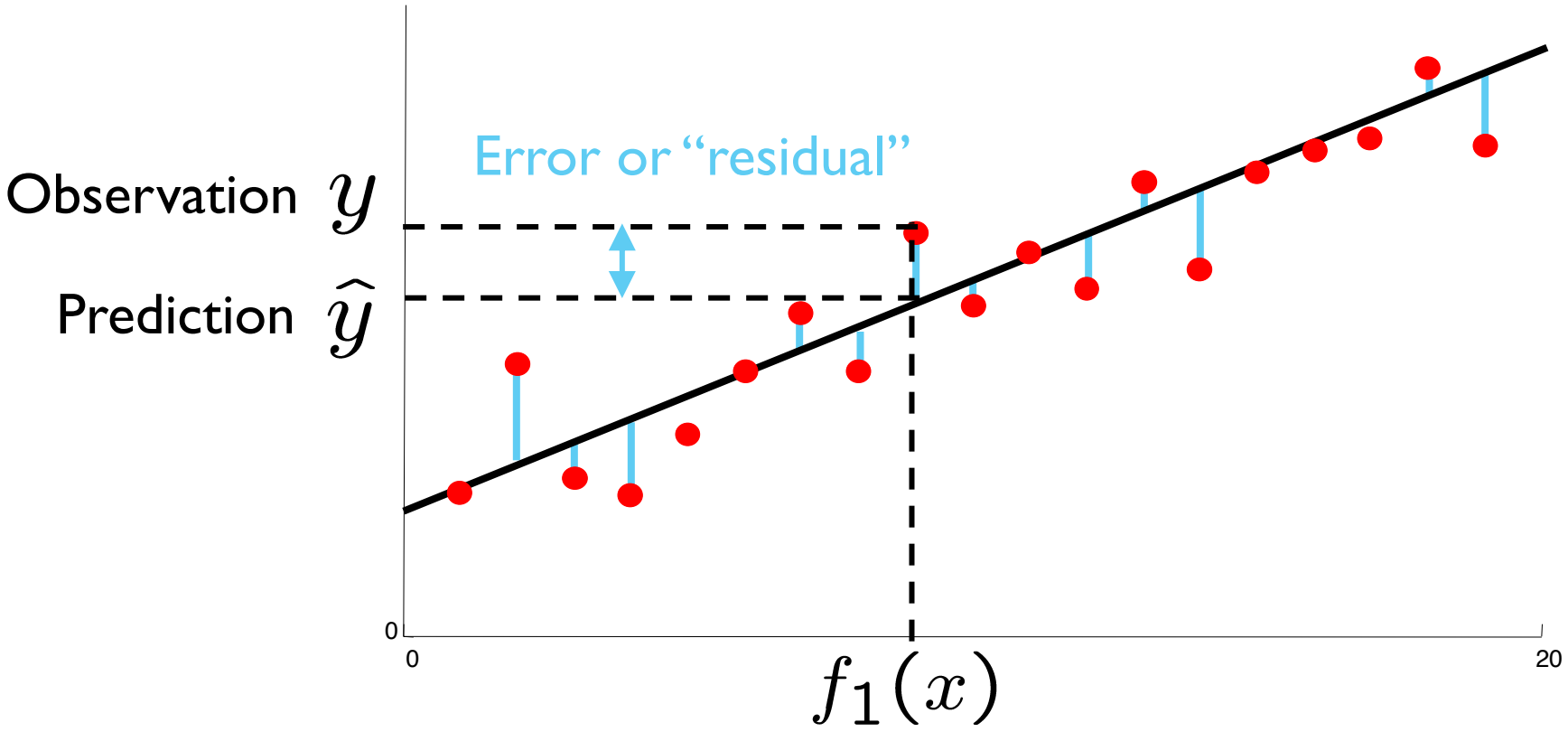
$$Q(s, NORTH) = 1.0 \qquad r + \gamma \max_{a'} Q(s', a') = -500 + 0$$

$$w_1 = 4.0 + \propto (-501)0.5 \qquad w_2 = -1.0 + \propto (-501)1.0$$

$$Q(s,a) = 3.0f_1(s,a) - 3.0f_2(s,a)$$ Updated equation

Approximate Q learning update

$$\text{total error} = \sum_i (y_i - \hat{y}_i)^2 = \sum_i \left(y_i - \sum_k w_k f_k(x_i) \right)^2$$



Error minimization

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"

Other extensions

- Represent Q(s,a) by a neural network so that more complex functions on the basic features can be represented.
- Use backpropagation to update weights, instead of the least squares update on weights shown before.

TD Gammon

- A reinforcement learner that learned to play backgammon using TD learning with function approximation .
- 2 layer neural network to represent V(s). The state of the board is captured by a vector of length 198. It includes (number of pieces on board, off board, whose turn it is, and for each cell whether there are 1, 2, 3 or more white (resp. black) pieces). There are 80 hidden units and 4 outputs. The outputs are the probability that white wins, probability that white gammons, probability that black wins, and probability that black gammons.

TD Gammon (contd.)

- Network is trained to minimize the error between actual payoff in game and payoff computed from the 4 outputs.
- Compression achieved by an implicit representation of V that allows learner to generalize from states it has visited from states that it has not. It examines only 10^{44} out of the 10^{120} possible states.
- After 1,500,000 training games with 80 hidden units, TD Gammon lost 1 point in 40 games.

Why does TD Gammon work?

- Incomplete understanding of TD-Gammon's success. The specific input encoding and choice of neural net parameters are crucial.
- Method does not scale to Go and Chess.

Policy search

- Problem: often the feature-based policies that work well (win games, maximize utilities) aren't the ones that approximate V / Q best
  - Q-learning's priority: get Q-values close (modeling)
  - Action selection priority: get ordering of Q-values right (prediction)
  - This distinction between modeling and prediction comes up a lot in AI.

- Solution: learn policies that maximize rewards, not the values that predict them

- Policy search: start with a policy (i.e., one found by Q-learning), then fine-tune policy directly by hill climbing on feature weights.

AlphaGo as described by DeepMind

- http://icml.cc/2016/tutorials/AlphaGo-tutorial-slides.pdf
- Alpha Go uses RL only as a component, look ahead search by generating game tree is crucial for AlphaGo, and exploring the state space in a smarter way.