CSE 573: Artificial Intelligence
Winter 2019

Hanna Hajishirzi

Reinforcement Learning

slides from
Dan Klein, Stuart Russell, Andrew Moore, Dan Weld, Pieter Abbeel, Luke Zettelmoyer

Announcements

- PS3 is due tonight.
- Quiz1 is graded.
- Project – Part I will be released tonight.
  - Groups of one or two
  - You can do your own project if relevant to this class.
- Survey
- Paper report
- Review sessions with TAs?

The Story So Far: MDPs and RL

Known MDP: Offline Solution

Goal	Technique
Compute V*, Q*, π*	Value / policy iteration
Evaluate a fixed policy π	Policy evaluation

Unknown MDP: Model-Based

Goal	Technique
Compute V*, Q*, π*	VI/PI on approx. MDP
Evaluate a fixed policy π	PE on approx. MDP

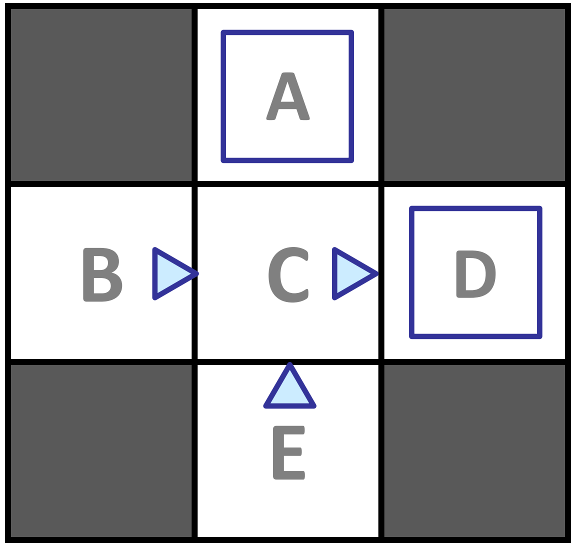
Unknown MDP: Model-Free

Goal	Technique
Compute V*, Q*, π*	Q-learning
Evaluate a fixed policy π	Value Learning

Reinforcement Learning - Neat property: Learn and Plan

Example: Model-Based Learning

Input Policy π



Assume: γ = 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
…

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

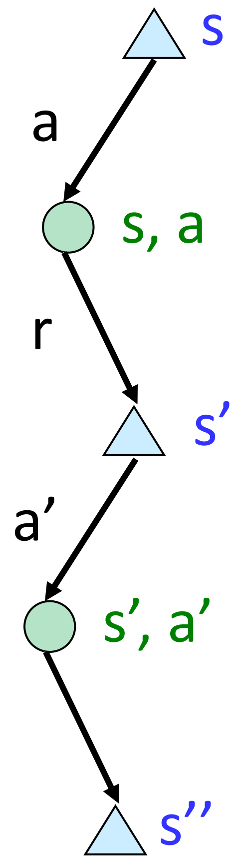
R(B, east, C) = -1
R(C, east, D) = -1
R(D, exit, x) = +10
…

Model-Free Learning

- Model-free (temporal difference) learning
  - Experience world through episodes

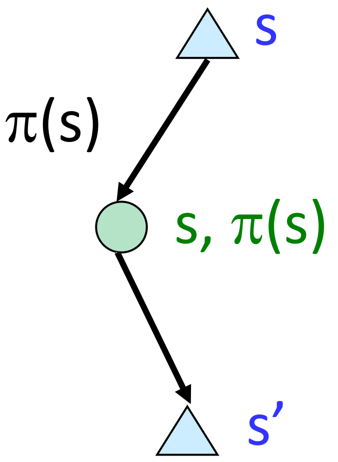
$$(s, a, r, s', a', r', s'', a'', r'', s'''' \ldots)$$

  - Update estimates each transition (s, a, r, s')
  - Over time, updates will mimic Bellman updates



Passive Reinforcement Learning: Temporal Difference Learning

- Big idea: learn from every experience!
  - Update V(s) each time we experience a transition (s, a, s', r)
  - Likely outcomes s' will contribute updates more often



- Temporal difference learning of values
  - Policy still fixed, still doing evaluation!
  - Move values toward value of whatever successor occurs: running average

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

Update to V(s): $$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha) sample$$

Same update: $$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$$

Active Reinforcement Learning

- Full reinforcement learning: optimal policies (like value iteration)
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - You choose the actions now
  - Goal: learn the optimal policy / values

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

Q-Learning

- We'd like to do Q-value updates to each Q-state:

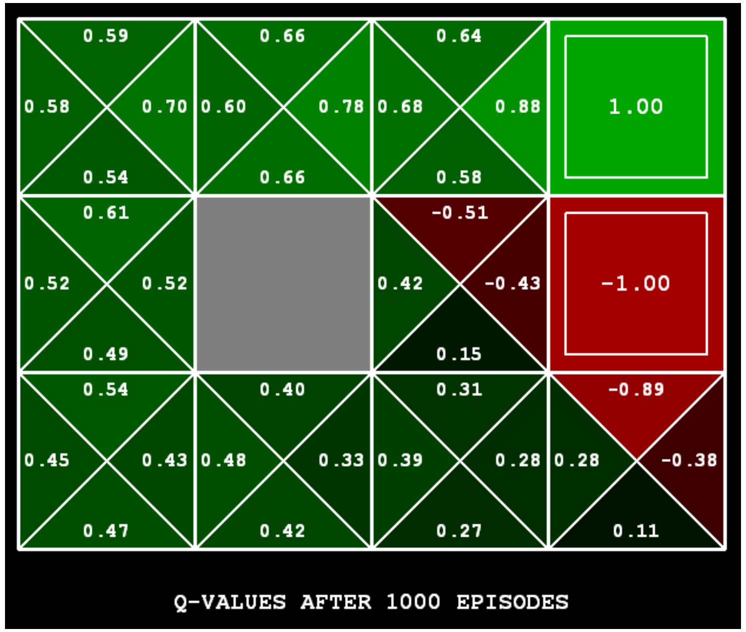
$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

  - But can't compute this update without knowing T, R

- Instead, compute average as we go
  - Receive a sample transition (s,a,r,s')
  - This sample suggests

$$Q(s,a) \approx r + \gamma \max_{a'} Q(s',a')$$

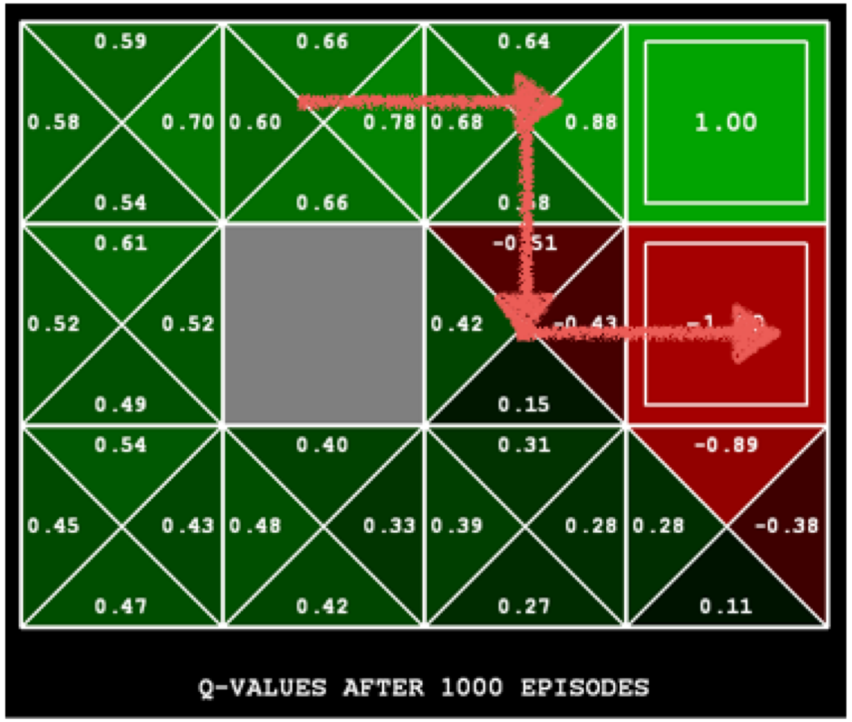
  - But we want to average over results from (s,a) (Why?)
  - So keep a running average



$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\left[r + \gamma \max_{a'} Q(s',a') \right]$$

Q-Learning Final Solution

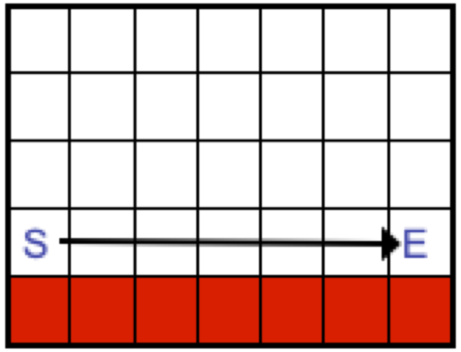
- Q-learning produces tables of q-values:

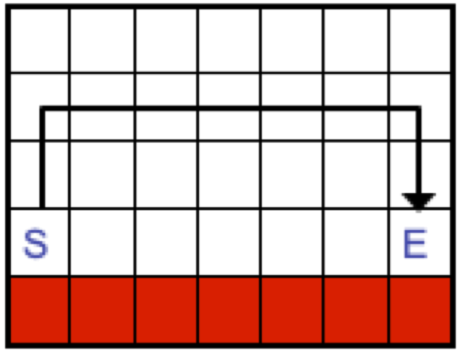


$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

Q-Learning Properties

- Amazing result: Q-learning converges to optimal policy -- even if you're acting suboptimally!
- This is called off-policy learning
- Caveats:
  - 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 (!)





(Tabular) Q-Learning

Algorithm:

Start with $Q_0(s,a)$ for all s, a.

Get initial state s

For k = 1, 2, … till convergence

Sample action a, get next state s’

If s’ is terminal:

$$\text{target} = R(s,a,s')$$

Sample new initial state s’

else:

$$\text{target} = R(s,a,s') + \gamma \max_{a'} Q_k(s',a')$$

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha\,[\text{target}]$$

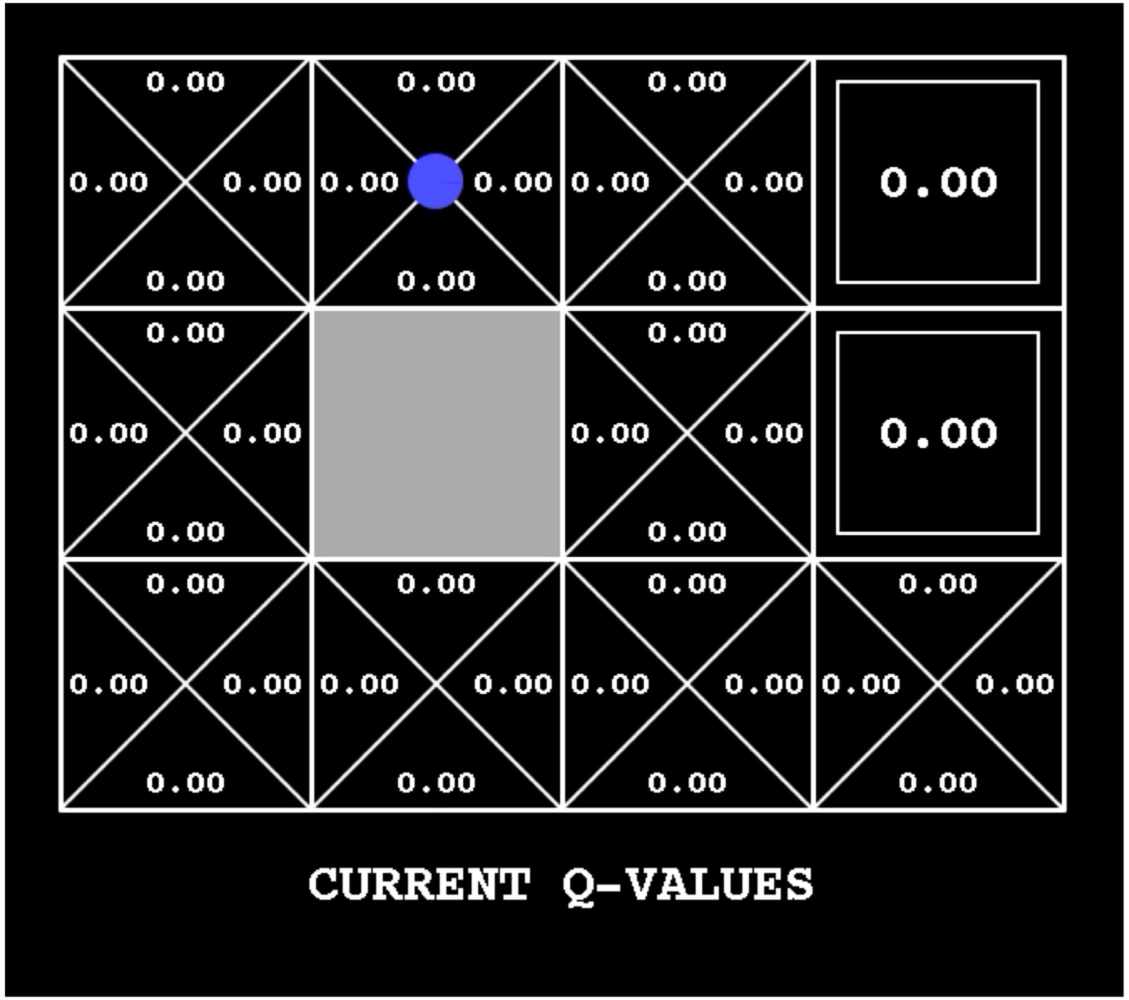
$$s \leftarrow s'$$

How to sample actions?

- Choose random actions?
- Choose action that maximizes $Q_k(s, a)$ (i.e. greedily)?
- ε-Greedy: choose random action with prob. ε, otherwise choose action greedily

How to Sample Actions (Explore)?

- Several schemes for forcing exploration
  - Simplest: random actions (ε-greedy)
    - Every time step, flip a coin
    - With (small) probability ε, act randomly
    - With (large) probability 1-ε, act on current policy
- Problems with random actions?
  - You do eventually explore the space, but keep thrashing around once learning is done
  - One solution: lower ε over time
  - Another solution: exploration functions



Q-Learn Epsilon Greedy

Exploration Functions

- When to explore?
  - Random actions: explore a fixed amount
  - Better idea: 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, e.g. $f(u,n) = u + k/n$

Regular Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} Q(s',a')$

Modified Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} f(Q(s',a'),N(s',a'))$

  - Note: this propagates the "bonus" back to states that lead to unknown states as well!

Regret

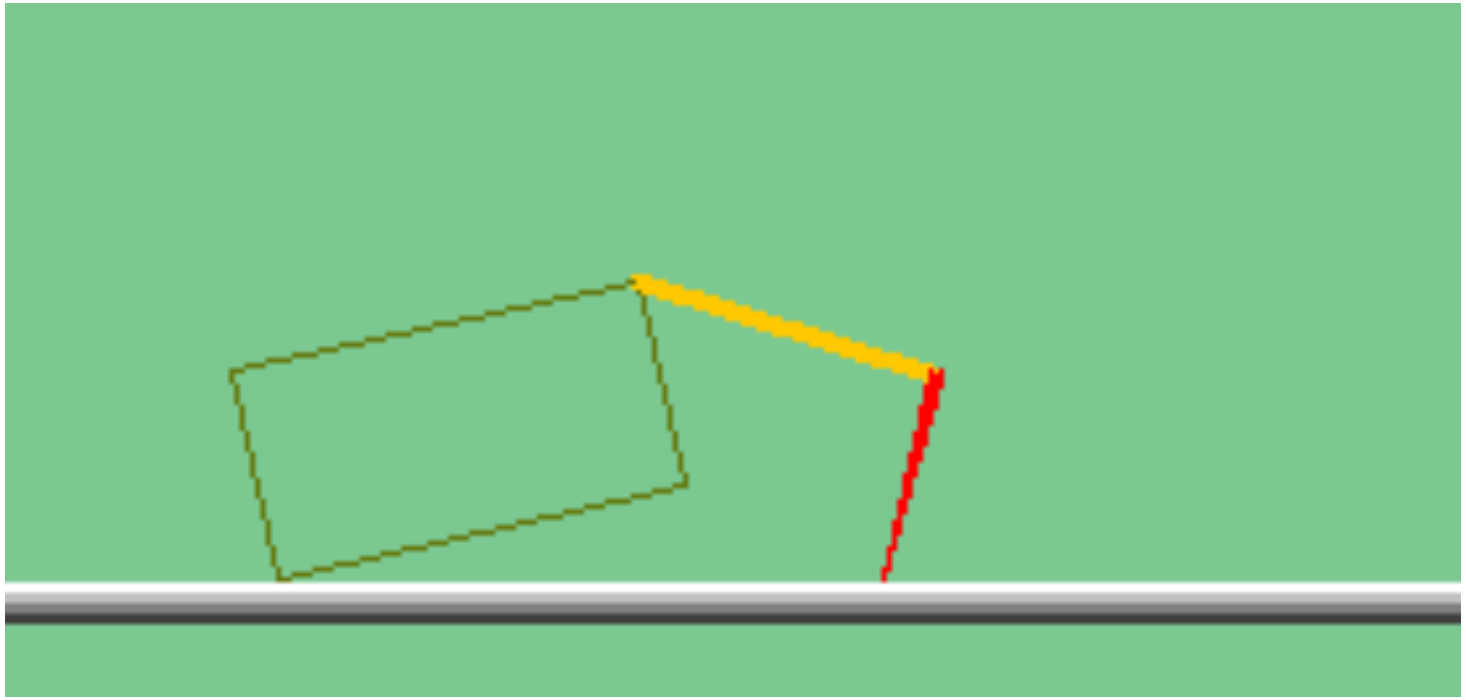
- Even if you learn the optimal policy, you still make mistakes along the way!

- Regret is a measure of your total mistake cost:
  - the difference between your (expected) rewards and optimal (expected) rewards

- Minimizing regret goes beyond learning to be optimal
  - it requires optimally learning to be optimal

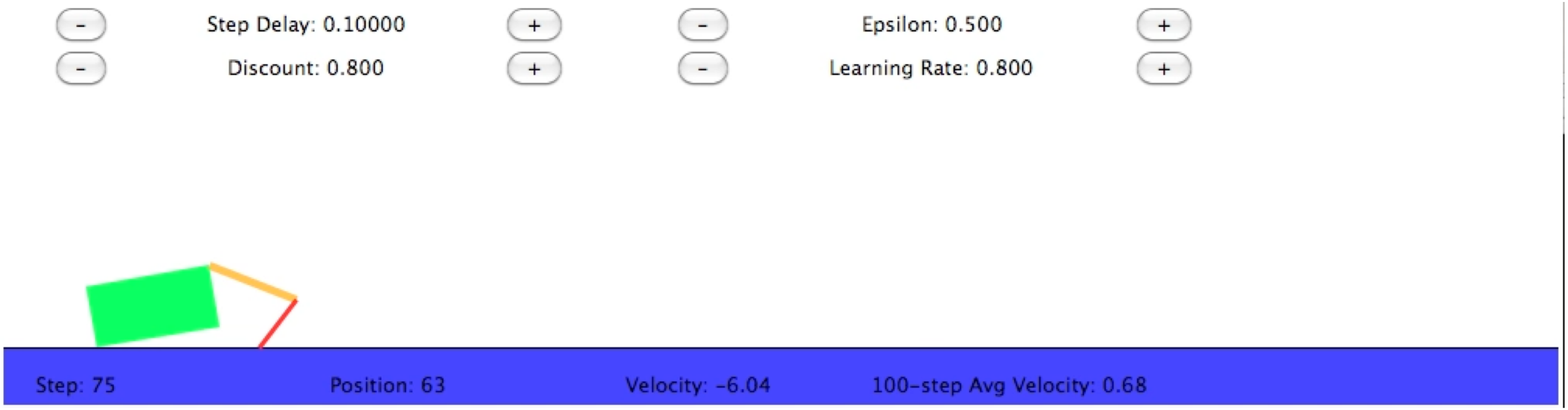
- Example: random exploration and exploration functions both end up optimal,
  - but random exploration has higher regret

The Crawler!



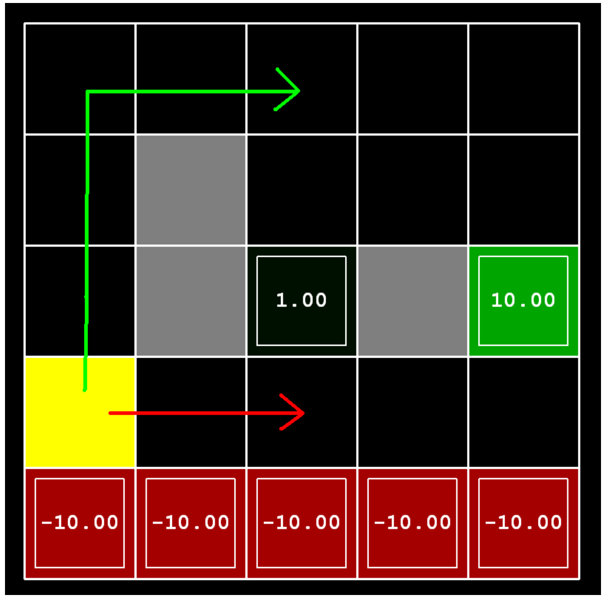
- **States: discretized value of 2d state: (arm angle, hand angle)**
- **Actions: Cartesian product of {arm up, arm down} and {hand up, hand down}**
- **Reward: speed in the forward direction**

[Demo: Crawler Bot (L10D1)] [You, in Project 3]

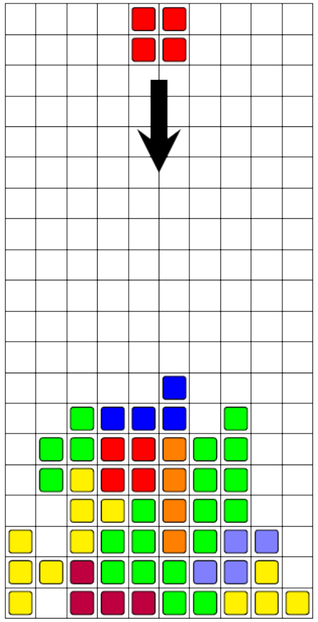

- Step Delay: 0.10000 +
- Epsilon: 0.500 +
- Discount: 0.800 +
- Learning Rate: 0.800 +
Step: 75
Position: 63
Velocity: -6.04
100-step Avg Velocity: 0.68

Can Tabular Methods Scale?

- Discrete environments



Gridworld
10^1

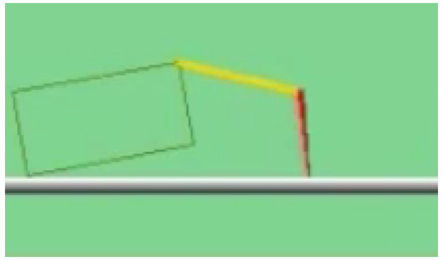


Tetris
10^60

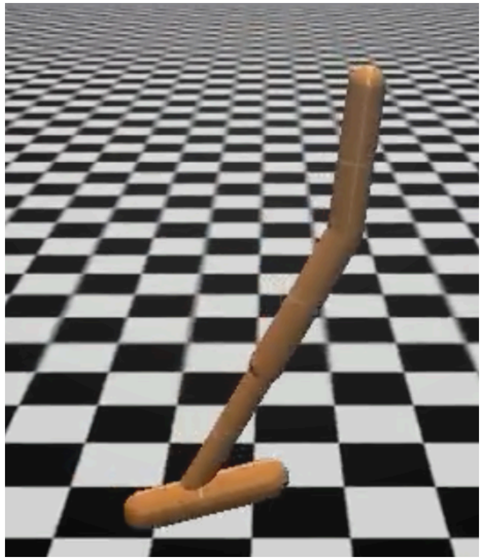
Atari
10^308 (ram) 10^16992 (pixels)

Can Tabular Methods Scale?

- Continuous environments (by crude discretization)



Crawler
10^2

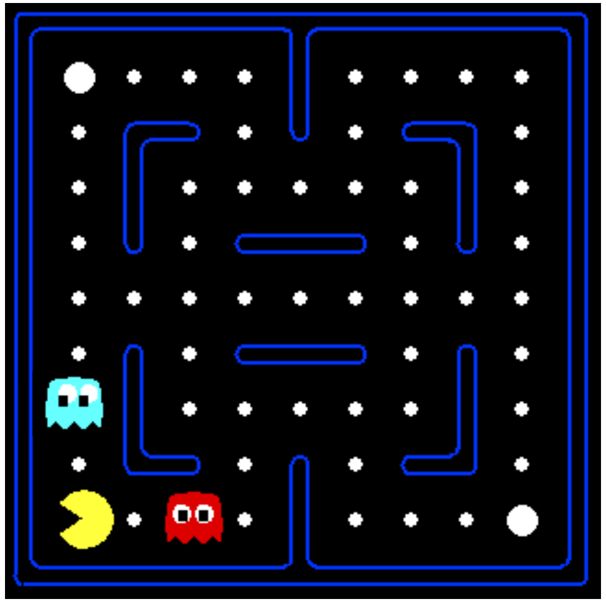


Hopper
10^10

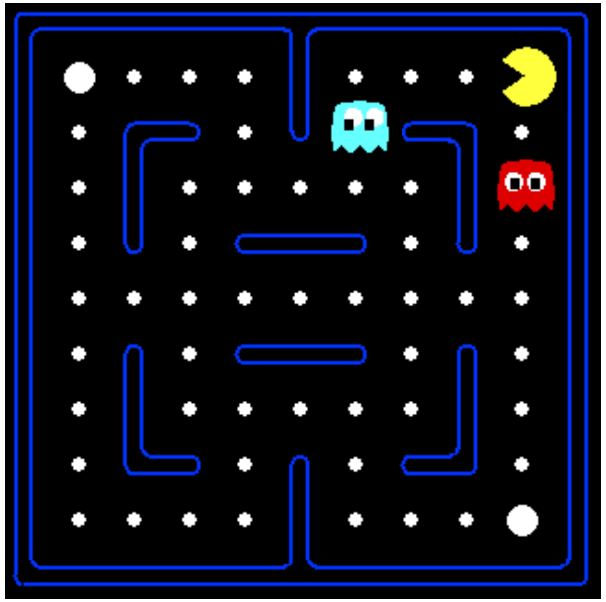
Humanoid
10^100

Example: Pacman

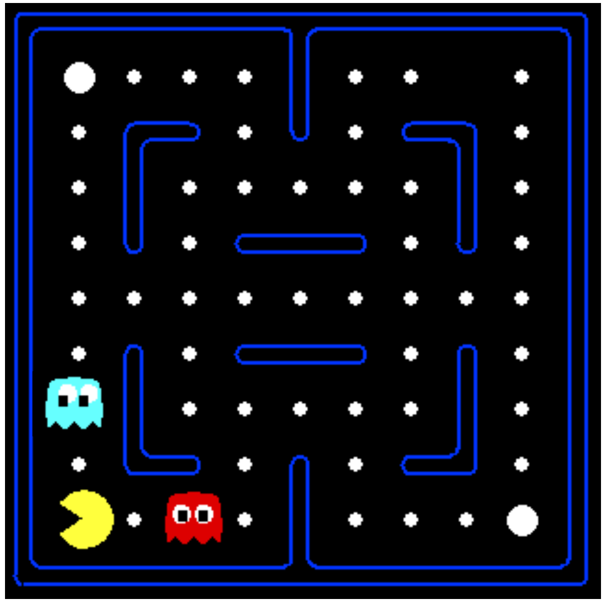
Let's say we discover through experience that this state is bad:



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



Or even this one!

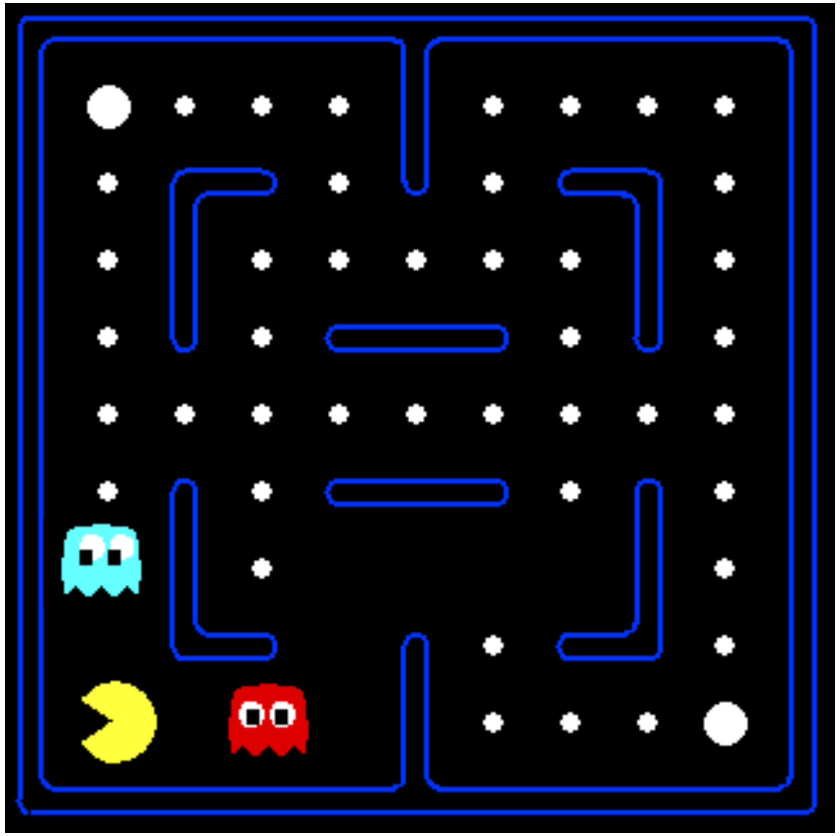


Generalizing Across States

- 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 (Approximate Q-Learning)
  - 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, and we'll see it over and over again

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.
  - 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:

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

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

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

- Disadvantage: states may share features but actually be very different in value!

Approximate Q-Learning

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

- Q-learning with linear Q-functions:

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

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

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

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

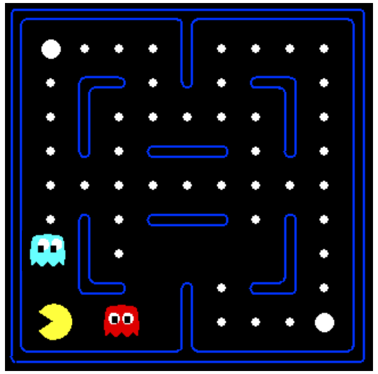
- Intuitive interpretation:
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state's features

- Formal justification: online least squares

Example: Q-Pacman

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

s

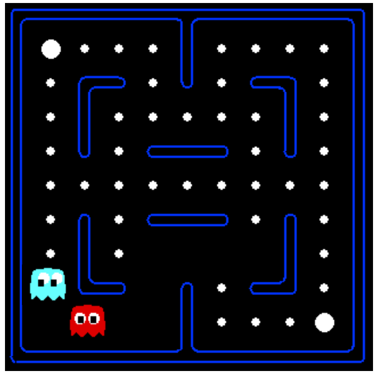


$f_{DOT}(s, \text{NORTH}) = 0.5$

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

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

s'



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

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

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

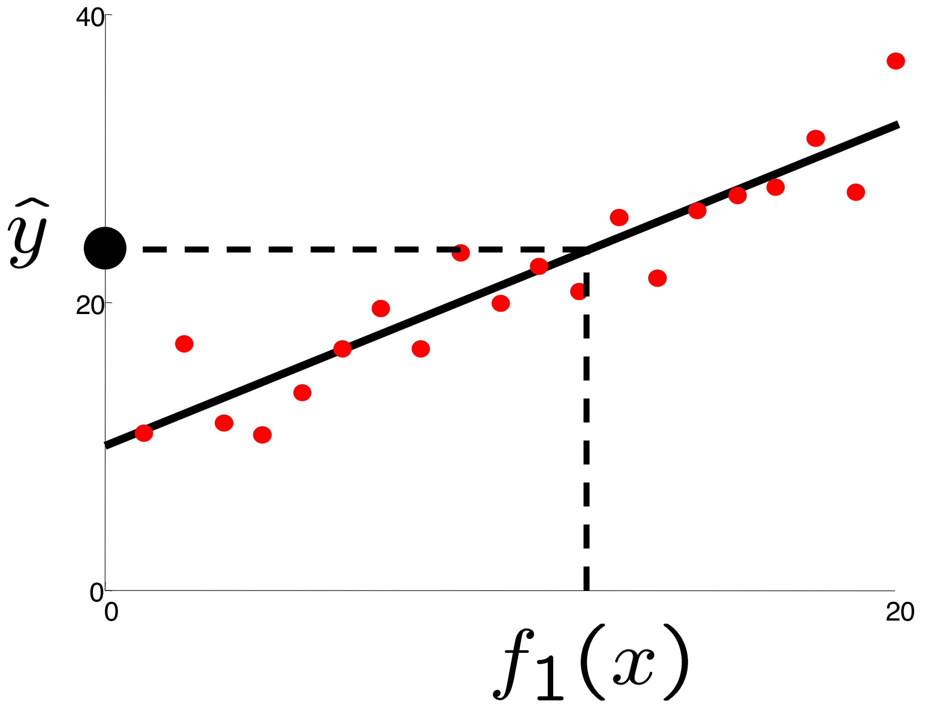
difference $= -501$

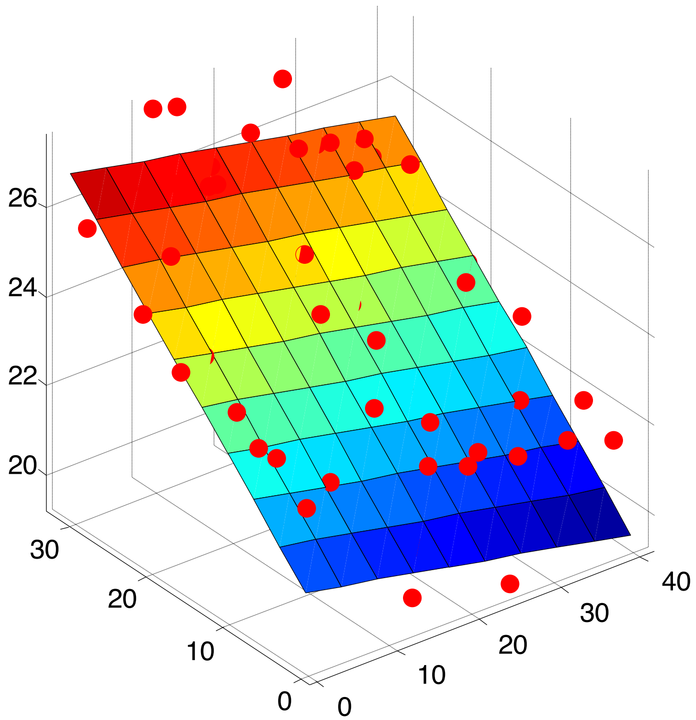
$w_{DOT} \leftarrow 4.0 + \alpha\,[-501]\,0.5$

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

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

Linear Approximation: Regression*





Prediction:

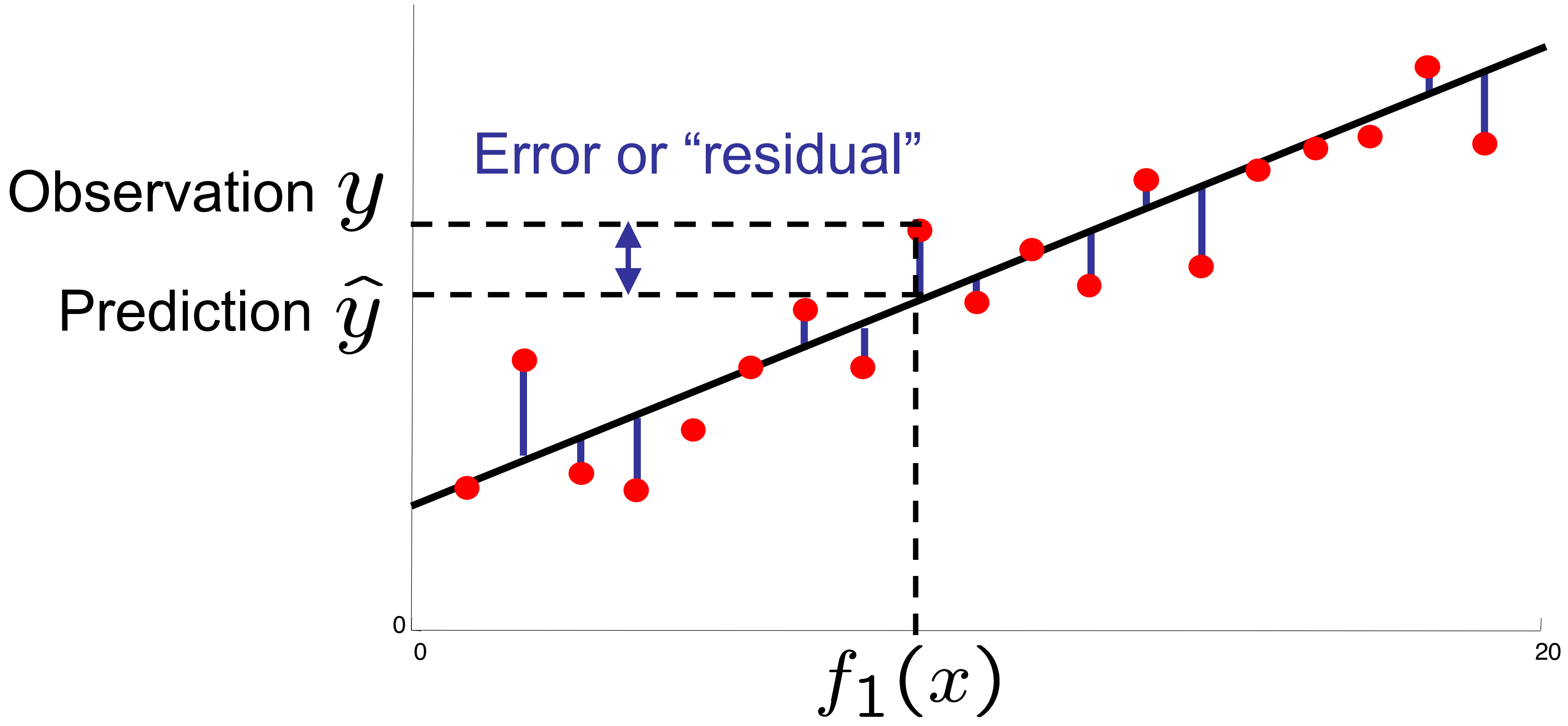
$$\hat{y} = w_0 + w_1 f_1(x)$$

Prediction:

$$\hat{y}_i = w_0 + w_1 f_1(x) + w_2 f_2(x)$$

Optimization: Least Squares*

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



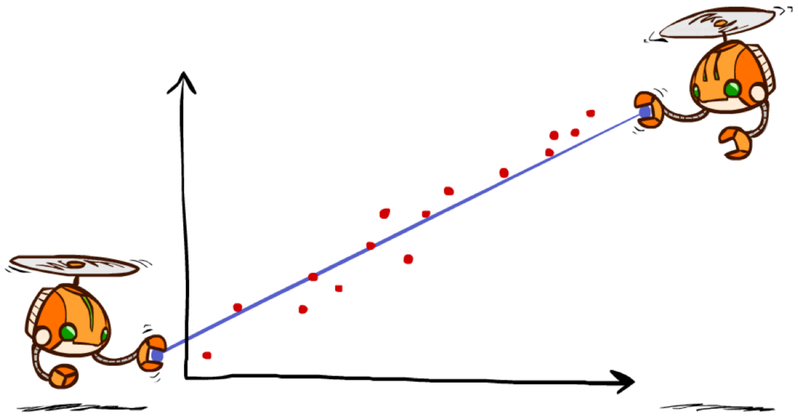
Minimizing Error*

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 Q-Learning

- Instead of a table, we have a parametrized Q function: $Q_\theta(s, a)$
  - Can be a linear function in features:

$$Q_\theta(s, a) = \theta_0 f_0(s, a) + \theta_1 f_1(s, a) + \cdots + \theta_n f_n(s, a)$$

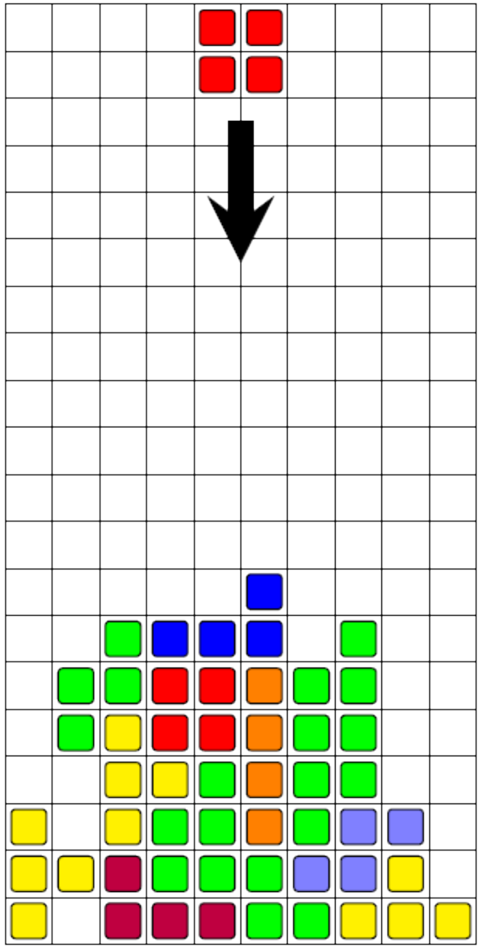
  - Or a complicated neural net

- Learning rule:
  - Remember: $\mathrm{target}(s') = R(s, a, s') + \gamma \max_{a'} Q_{\theta_k}(s', a')$
  - Update:

$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \left[\frac{1}{2} (Q_\theta(s, a) - \mathrm{target}(s'))^2 \right] \Bigg|_{\theta = \theta_k}$$

Engineered Approximate Example: Tetris

- state: naïve board configuration + shape of the falling piece ~10^{60} states!
- action: rotation and translation applied to the falling piece

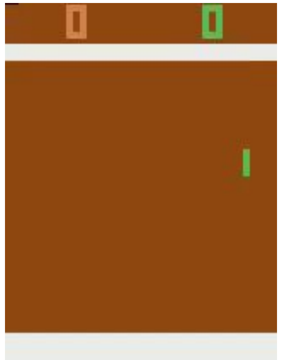


- 22 features aka basis functions ϕ_i
  - Ten basis functions, 0, *. . . , 9, mapping the state to the height h[k] of each* column.
  - Nine basis functions, 10, *. . . , 18, each mapping the state to the absolute difference* between heights of successive columns: |*h[k+1] − h[k]*|*, k = 1, . . . , 9.*
  - One basis function, 19, that maps state to the maximum column height: $\max_k$ *h[k]*
  - One basis function, 20, that maps state to the number of 'holes' in the board.
  - One basis function, 21, that is equal to 1 in every state.

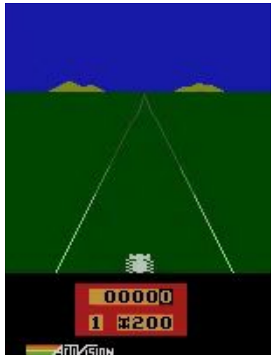
$$\hat{V}_\theta(s) = \sum_{i=0}^{21} \theta_i \phi_i(s) = \theta^\top \phi(s)$$

[Bertsekas & Ioffe, 1996 (TD); Bertsekas & Tsitsiklis 1996 (TD); Kakade 2002 (policy gradient); Farias & Van Roy, 2006 (approximate LP)]

Deep Reinforcement Learning

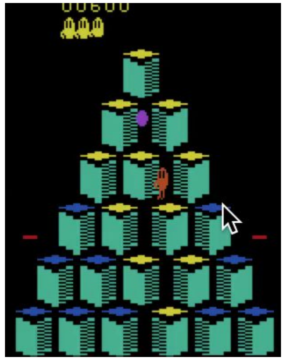


Pong



Enduro

Beamrider



Q*bert

- 49 ATARI 2600 games.
- From pixels to actions.
- The change in score is the reward.
- Same algorithm.
- Same function approximator, w/ 3M free parameters.
- Same hyperparameters.
- Roughly human-level performance on 29 out of 49 games.

Algorithm:

Start with $Q_0(s, a)$ for all s, a.

Get initial state s

For k = 1, 2, … till convergence

Sample action a, get next state s'

If s' is terminal:

$$\text{target} = R(s, a, s')$$

Sample new initial state s'

else:

$$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

Chasing a nonstationary target!

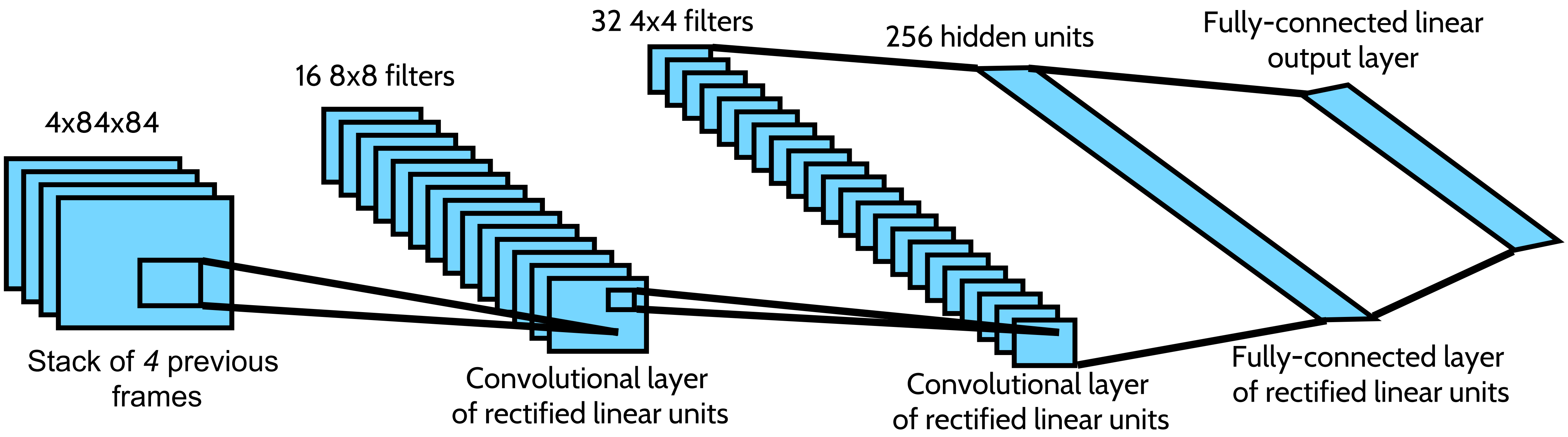
$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s' \sim P(s'|s,a)} \left[(Q_\theta(s, a) - \text{target}(s'))^2 \right] \Big|_{\theta=\theta_k}$$

$$s \leftarrow s'$$

Updates are correlated within a trajectory!

Atari Network Architecture

- Convolutional neural network architecture:
  - History of frames as input.
  - One output per action - expected reward for that action $Q(s, a)$.
  - Final results used a slightly bigger network (3 convolutional + 1 fully-connected hidden layers).



[Out of the scope of this class]

Policy Search

- Problem: often the feature-based policies that work well (win games, maximize utilities) aren't the ones that approximate V / Q best
  - E.g. your value functions from project 2 were probably horrible estimates of future rewards, but they still produced good decisions
  - Q-learning's priority: get Q-values close (modeling)
  - Action selection priority: get ordering of Q-values right (prediction)

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

- Policy search: start with an ok solution (e.g. Q-learning) then fine-tune by hill climbing on feature weights

Policy Search

- Simplest policy search:
  - Start with an initial linear value function or Q-function
  - Nudge each feature weight up and down and see if your policy is better than before

- Problems:
  - How do we tell the policy got better?
  - Need to run many sample episodes!
  - If there are a lot of features, this can be impractical

- Better methods exploit lookahead structure, sample wisely, change multiple parameters…

Why Policy Optimization?

- Often the policy can be simpler than Q or V
  - E.g., Robotic grasp
- V: doesn't prescribe actions
  - We need the dynamic model (+ compute 1 Bellman back-up)
- Q: need to be able to efficiently find the best action for every Q state
  - Challenge: What happens when actions are high-dimensional or continious

Policy Optimization

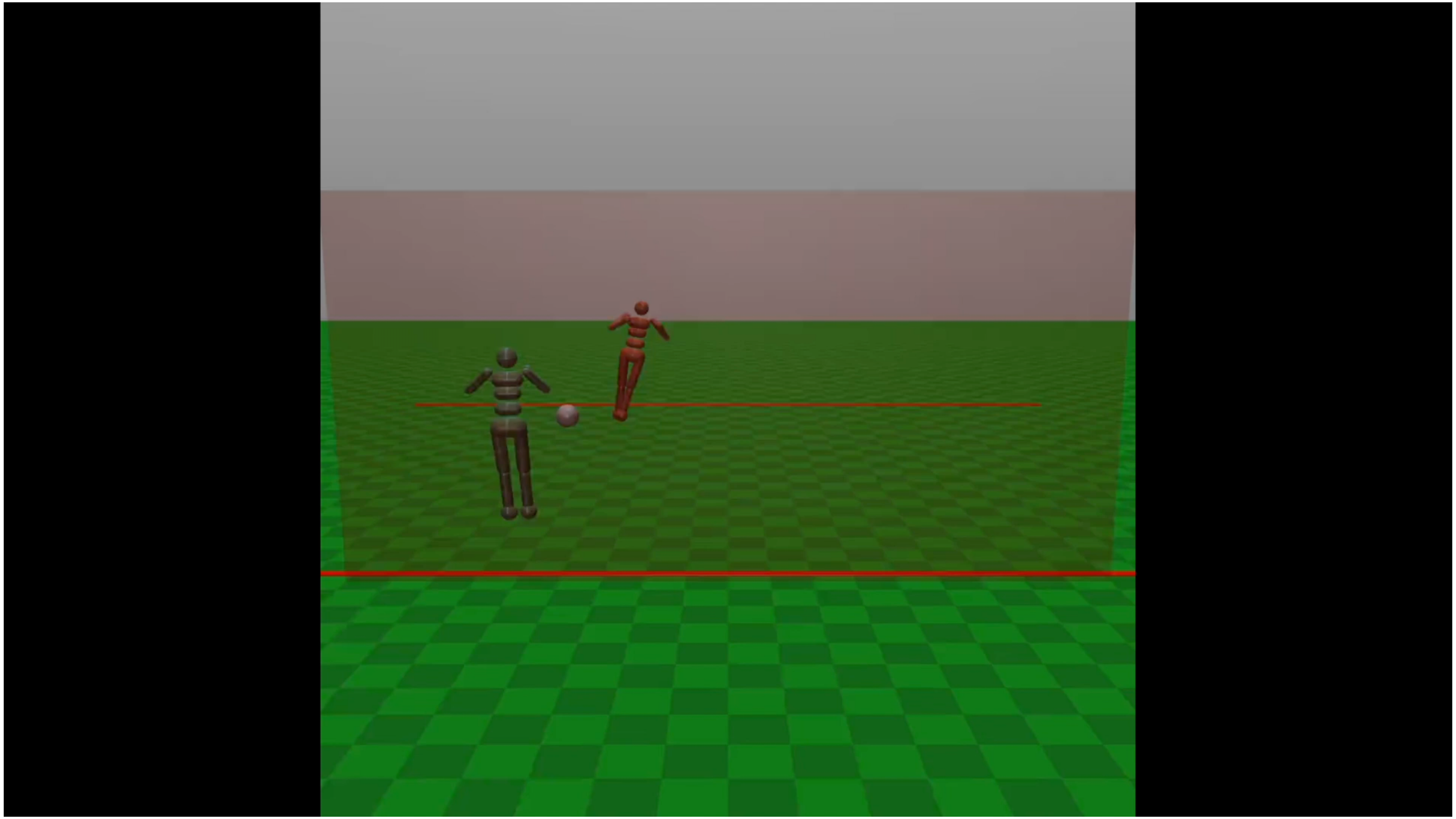
- Consider control policy parameterized by parameter vector θ

$$\max_{\theta} \quad \mathrm{E}[\sum_{t=0}^{H} R(s_t)|\pi_\theta]$$

- Stochastic policy class (smooths out the problem):

$\pi_\theta(u|s)$: probability of action u in state s

[Video: GAE]



	Policy Optimization	Dynamic Programming
Conceptually:	Optimize what you care about	Indirect, exploit the problem structure, self-consistency
Empirically:	More compatible with rich architectures (including recurrence) More versatile More compatible with auxiliary objectives	More compatible with exploration and off-policy learning More sample-efficient when they work

Example: Sidewinding

[Andrew Ng]

[Video: SNAKE – climbStep+sidewinding]