CPSC 4420/6420

Artificial Intelligence

10 – Function Approximation

September 22, 2020

Announcements

- Project 2 is due on 9/24
- Quiz 4 will be assigned after class
  - Deadline is next Tuesday

Lecture 10

Slide Credits:
Stuart Russell
Pieter Abbeel
Dan Klein
Ioannis Karamouzas

(Tabular) Q-Learning

- Q-Learning: sample-based Q-value iteration

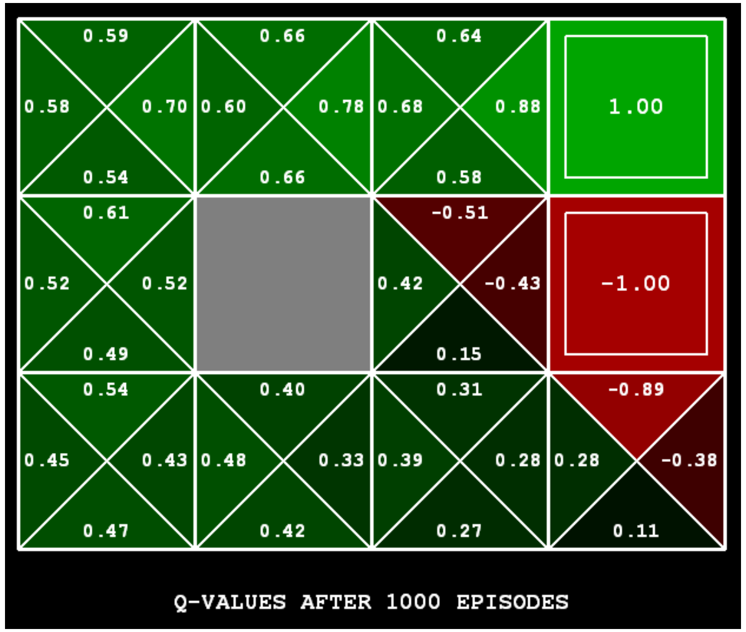
$$Q_{k+1}(s,a) \leftarrow \sum_{s'} P(s'|s,a)\left(R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right)$$

- Learn Q-values as you go
  - Receive a sample (s, a, r, s')
  - Consider your previous estimate: $\hat{Q}(s,a)$
  - Consider your new sample estimate

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

  - Incorporate the new estimate into a running average

$$\hat{Q}(s,a) \leftarrow (1-\alpha)\hat{Q}(s,a) + (\alpha)\,[\,\textit{target}\,]$$



(Tabular) Q-Learning

Initialize $Q(s, a)$ for all s, a

Repeat (for each episode):

Get initial state s

Repeat (for each step of episode):

Sample action a from s, observe reward r and next state s'

If s' is terminal:

$$target = r$$

else:

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

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

$$s \leftarrow s'$$

until s is terminal

until convergence

Epsilon-greedy policy

- With (small) probability ε, act randomly
- With (large) probability 1-ε, act on current policy
- How to set ε in practice?
  - Use a fixed ε
  - Start with a large ε (e.g, ε=1), decrease it over time to a small positive number (e.g., ε =0.1)

Q-learning properties

- Q-learning converges to optimal policy, even if you're acting suboptimally!
- This is called off-policy learning
  - On policy: Learn the values of the policy used to generate the data
  - Off policy: Learn the value of another policy
- 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!

The story so far

- Given an MDP find the optimal policy π*

$$\pi^* = \text{argmax}_\pi \, E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi\right]$$

- Exact methods
  - Value Iteration
  - Policy Iteration

<u>Limitations</u>

- Update equations require access to dynamics model and reward function
- Iteration over / storage for all states and actions: requires small, discrete state-action space

Approximate Q-Learning

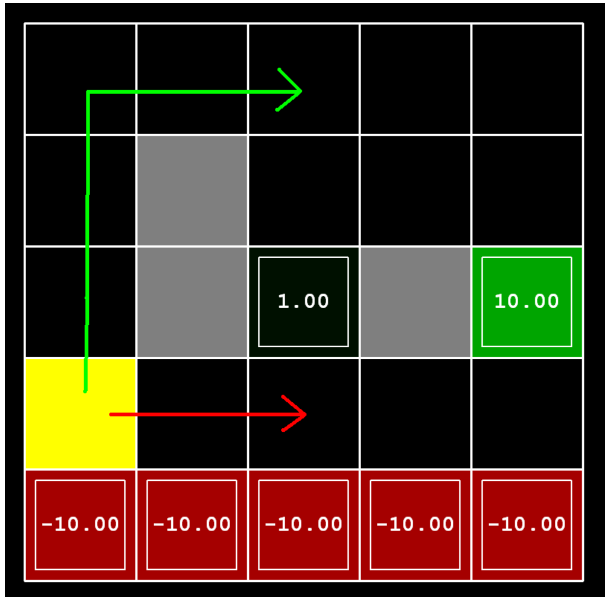
Generalizing across states

- Basic (tabular) Q-Learning keeps a table of all q-values

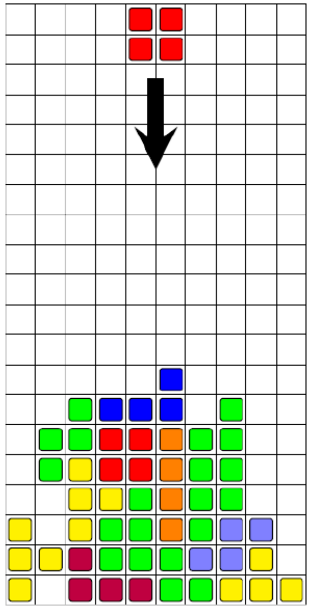
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the q-tables in memory

Can Q-learning scale?

- Discrete environments



Gridworld
10^2

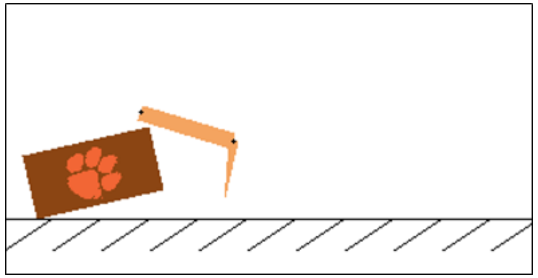


Tetris
10^60

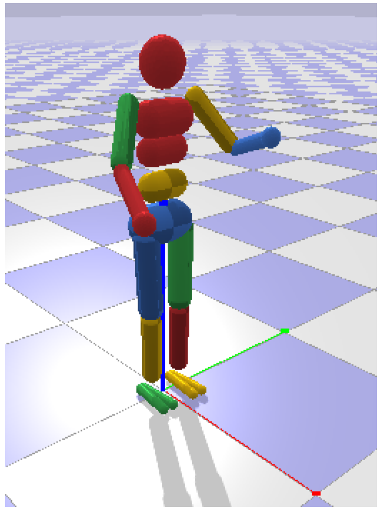
Atari
10^308

Can Q-learning scale?

- Continuous environments (crude discretization)



Crawler
10^2



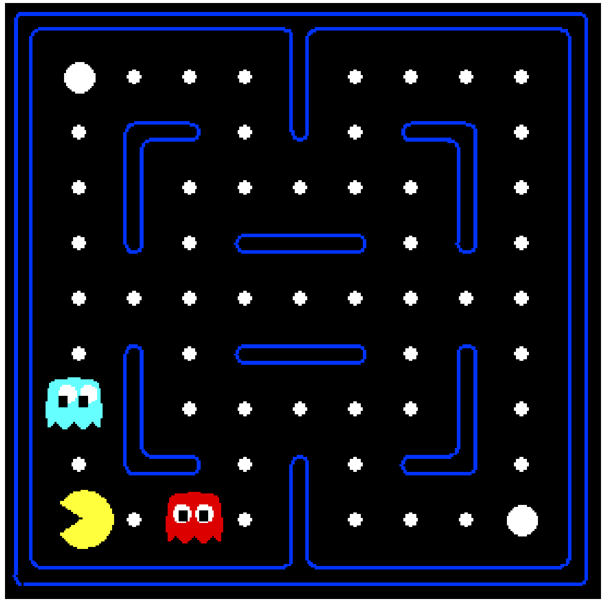
Humanoid
10^100

Generalizing across states

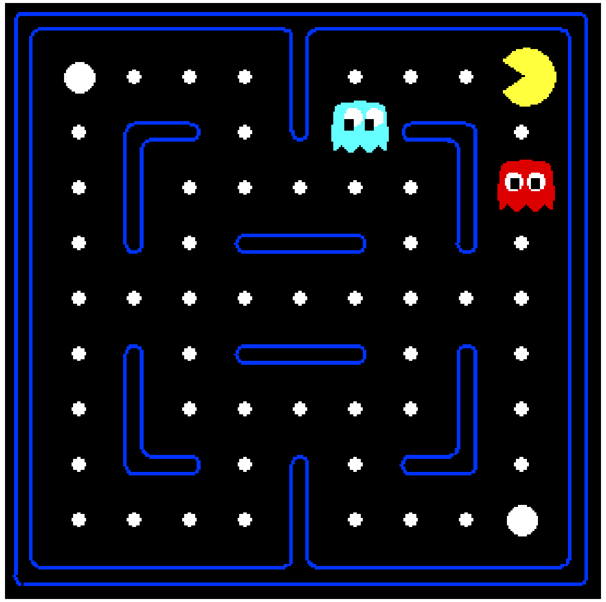
- Basic (tabular) Q-Learning keeps a table of all q-values
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the q-tables in memory
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar situations

Example

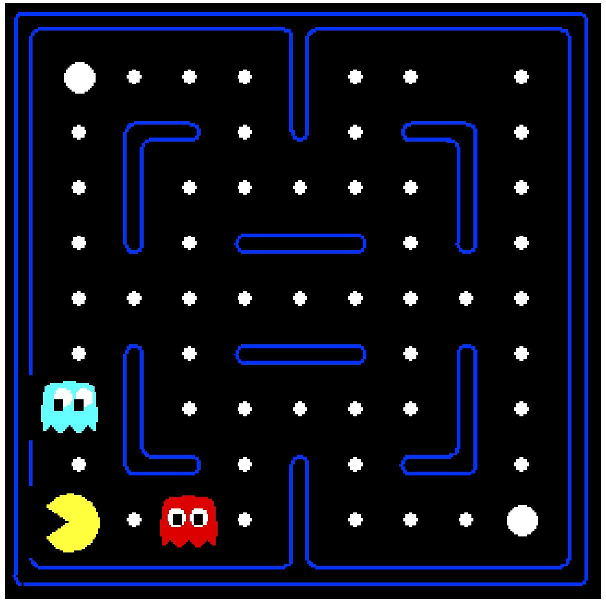
Let's say we discover through experience that this state is bad:



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

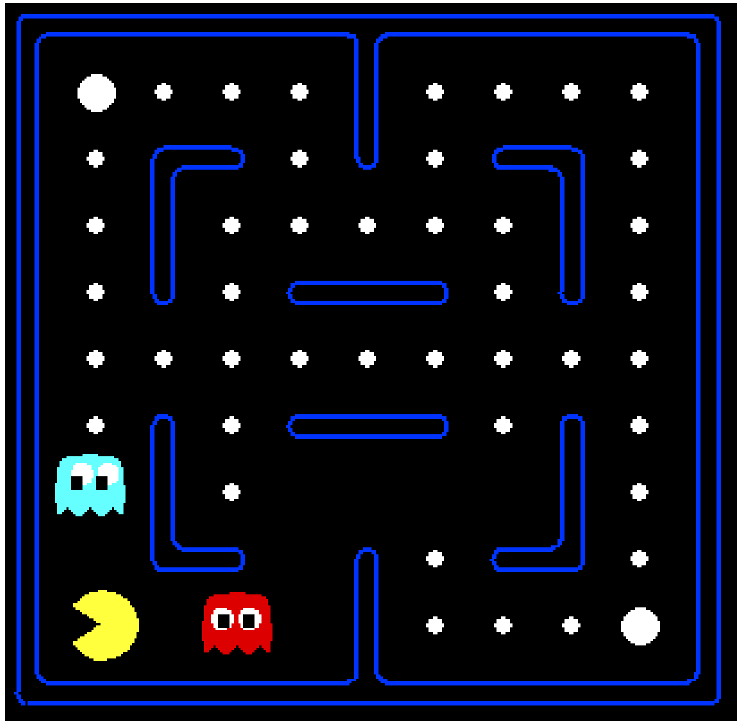


Or even this one!



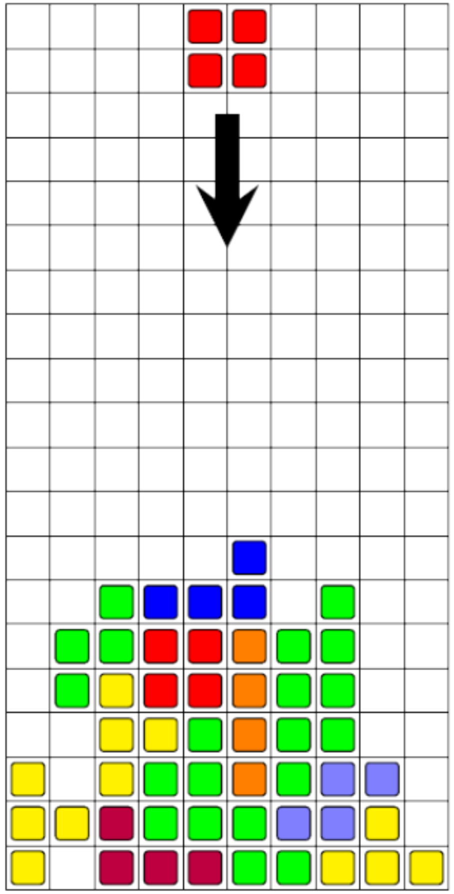
Feature-based representations

- Solution: describe a state using a vector of features
  - Features are functions from states to real numbers that capture important states' properties
  - Example features:
    - Distance to closest ghost
    - Distance to closest dot
    - Number of ghosts
    - Is Pacman in dead-end?
    - ……
  - Can also describe a q-state (s, a) with features (e.g. action moves closer to food)



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 f_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



Linear Q-value functions

- Instead of storing table, we can write a *parametrized* Q (or V) function for any state using a few weights: $Q_w(s,a)$
- Simple approach is to use a linear function in features f_i

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

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

Linear Q-value functions

- Instead of storing table, we can write a *parametrized* Q (or V) function for any state using a few weights: $Q_w(s,a)$
- Simple approach is to use a linear function in features f_i

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

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

- $\boldsymbol{w}$ is a free parameter vector to be chosen from its domain W
  - +: Representation size from $|S \times A|$ down to $|W|$ → less parameters to estimate
  - -: Less expressiveness

Approximate Q-learning

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

- Q-learning with linear functions
  - Receive a sample transition (s, a, r, s')
  - Consider the sample estimate

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

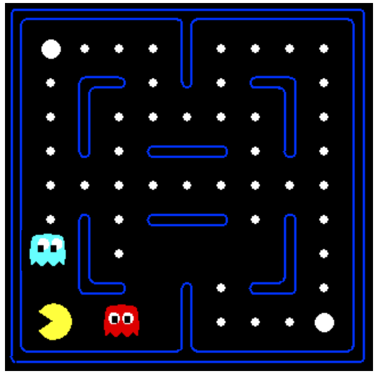
  - Update each weight w_k

$$w_k \leftarrow w_k + \alpha[\,\textit{target} - Q_w(s,a)\,] f_k(s,a)$$

- Interpretation
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the features that were on

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

s



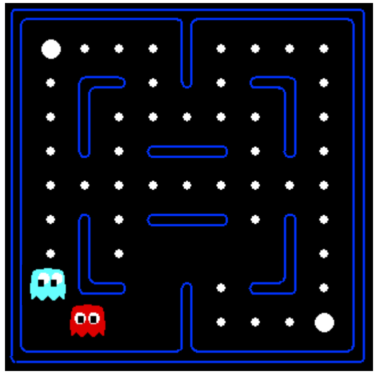
$f_{DOT}(s, \text{NORTH}) = 0.5$

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

$a = \text{NORTH}$

$r = -500$

s'



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

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

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

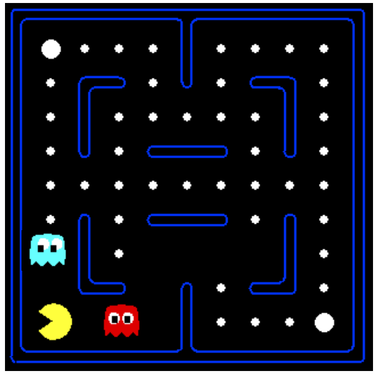
$\text{difference} = -501$ $\Rightarrow$

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

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

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

s



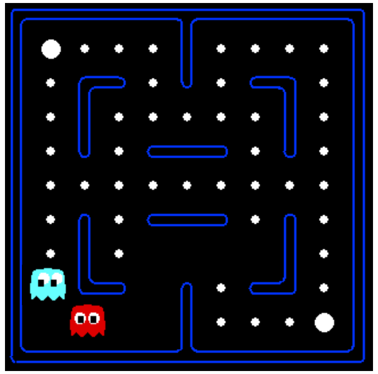
$f_{DOT}(s, \text{NORTH}) = 0.5$

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

$a = \text{NORTH}$

$r = -500$

s'



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

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

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

difference $= -501$

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

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

α = 0.0399

Approximate Q update explained

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

- Q-learning with linear functions
  - Receive a sample transition (s, a, r, s')
  - Consider the sample estimate

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

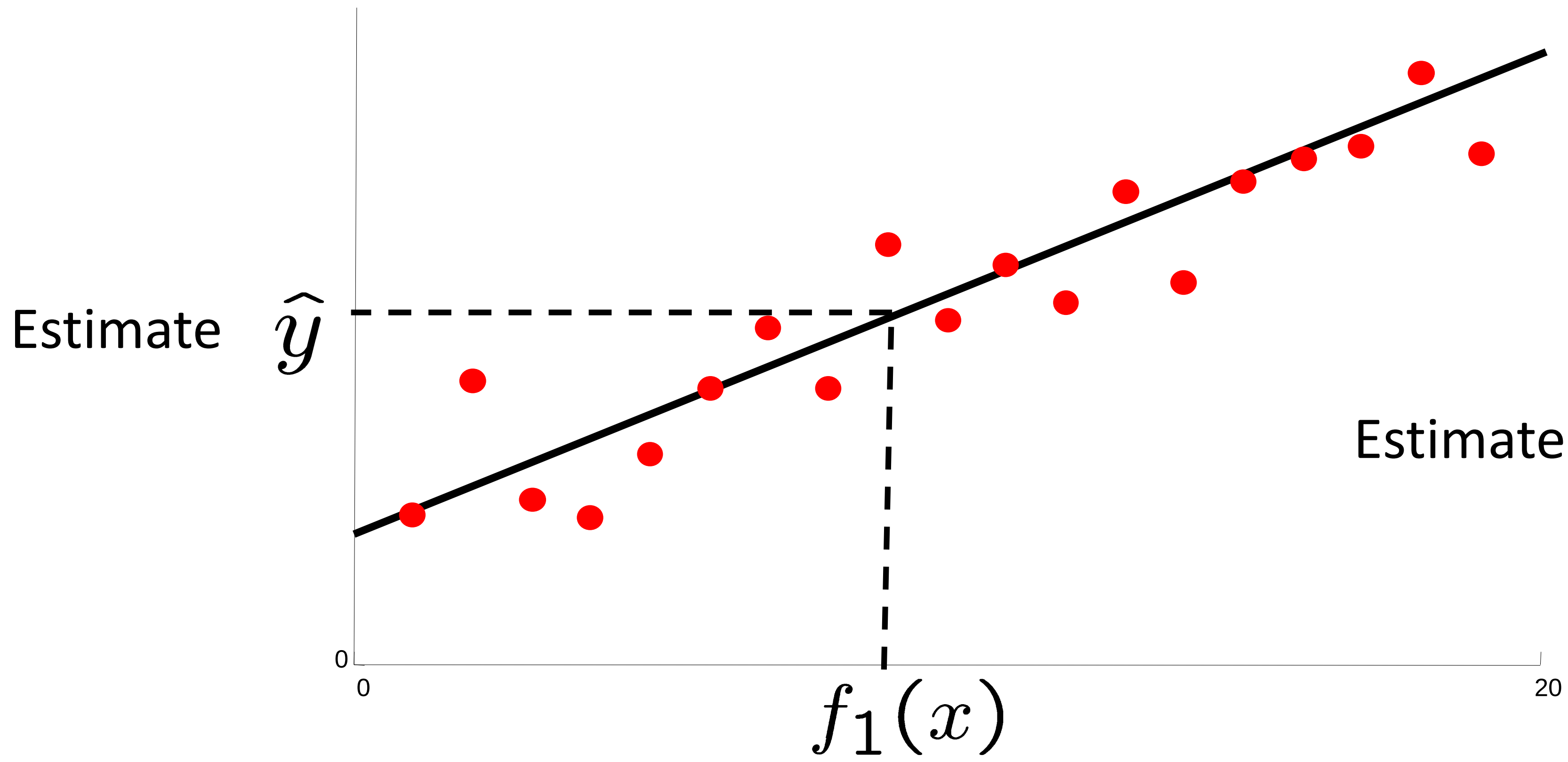
  - Update each weight w_k

$$w_k \leftarrow w_k + \alpha[\,\textit{target} - Q_w(s,a)\,] f_k(s,a)$$

- Interpretation:

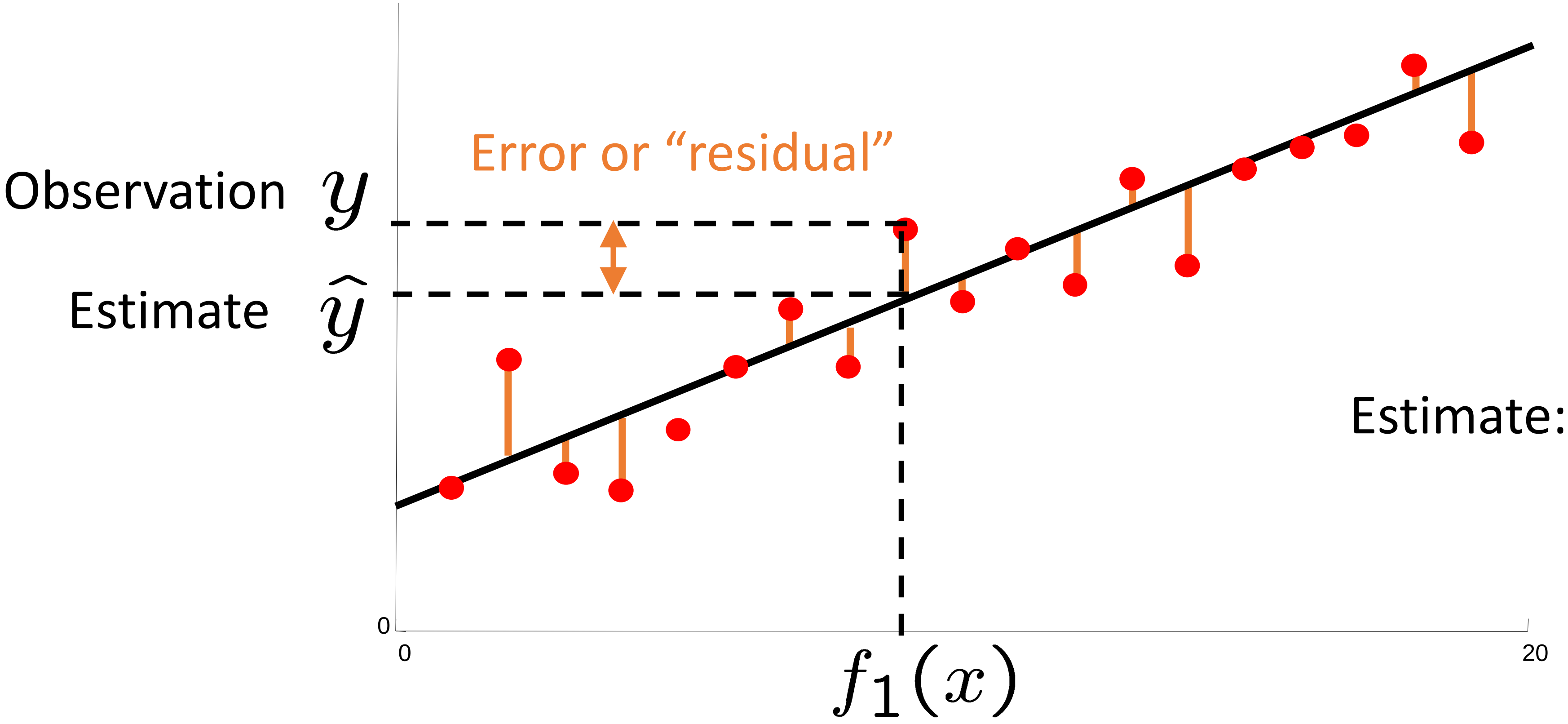
Gradient descent on $\left(R(s,a,s') + \gamma \max_{a'} Q_w(s',a') - Q_w(s,a)\right)$

Q-learning and least squares (revisit later)



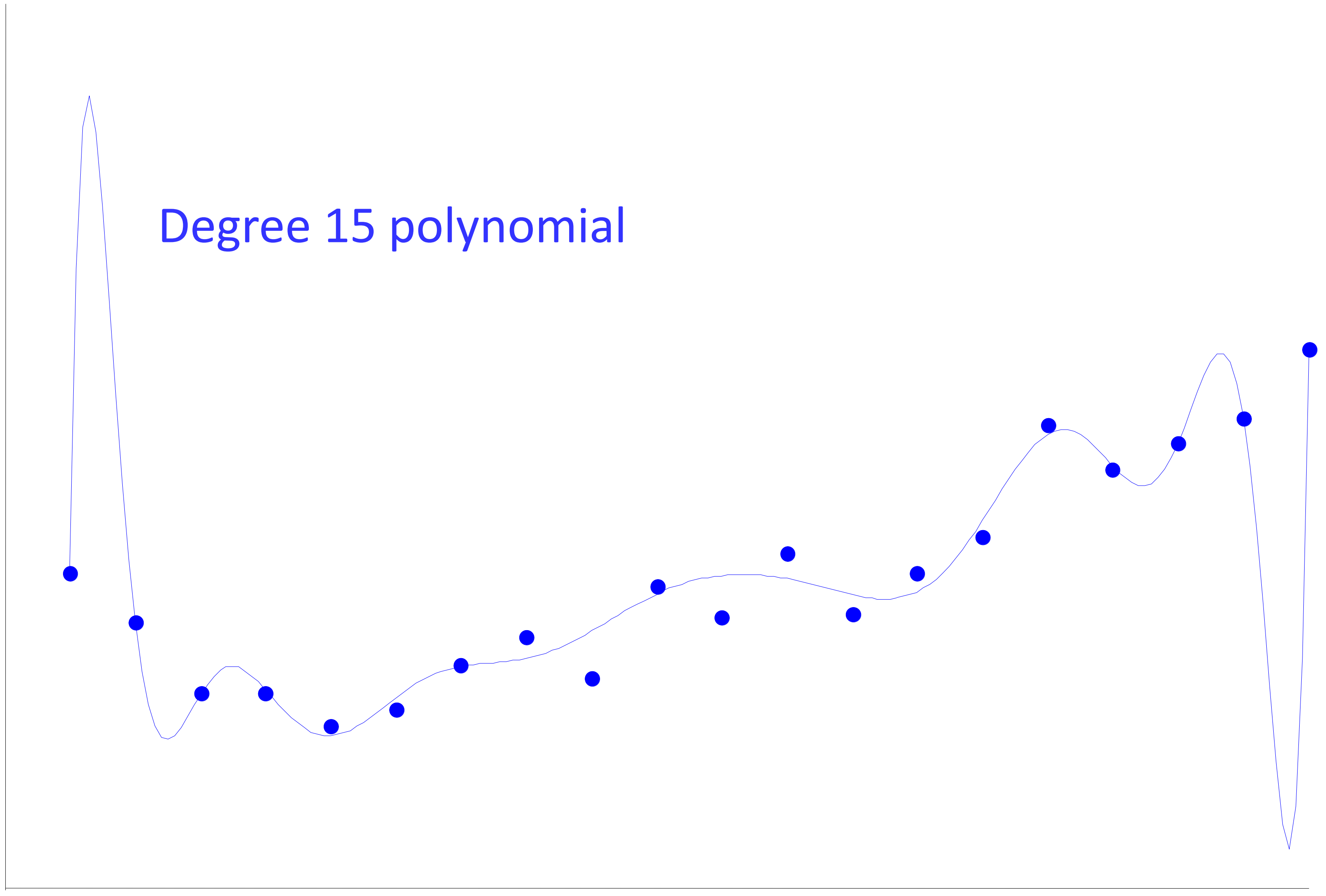
Estimate: $\hat{y} = w_0 + w_1 f_1(x)$

Q-learning and least squares (revisit later)



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

Do not overfit!



Other function approximations

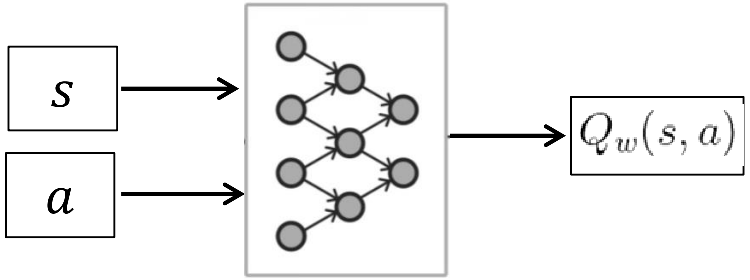
- Instead of a table, we have parametrized Q (or V) function: $Q_w(s,a)$
  - It can be a linear function in features f_i

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

  - Or some other function, e.g., polynomial

$$Q_w(s,a) = w_{11} f_1(s,a) + w_{12} f_1(s,a)^2 + w_{13} f_1(s,a)^3 + \ldots$$

  - Or a neural network (learn the features f_i too!)

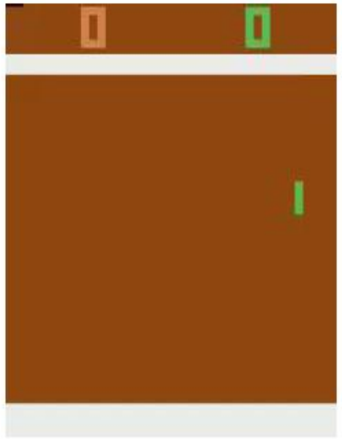


- Update rule

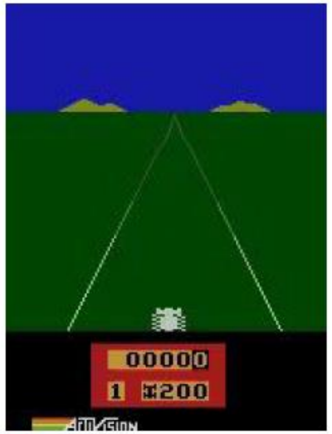
$$w_k \leftarrow w_k + \alpha \left[r + \gamma \max_{a} Q_w(s',a') - Q_w(s,a) \right] \frac{dQ}{dw_k}(s,a)$$

"target" "prediction"

Deep Q-Networks [Mnih et al, 2013]

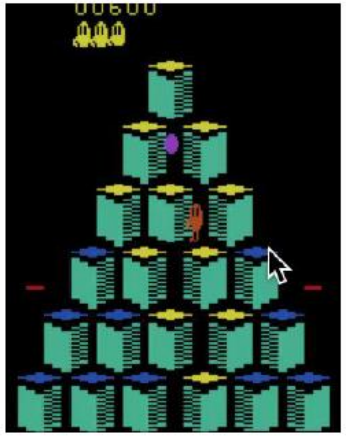


Pong



Enduro

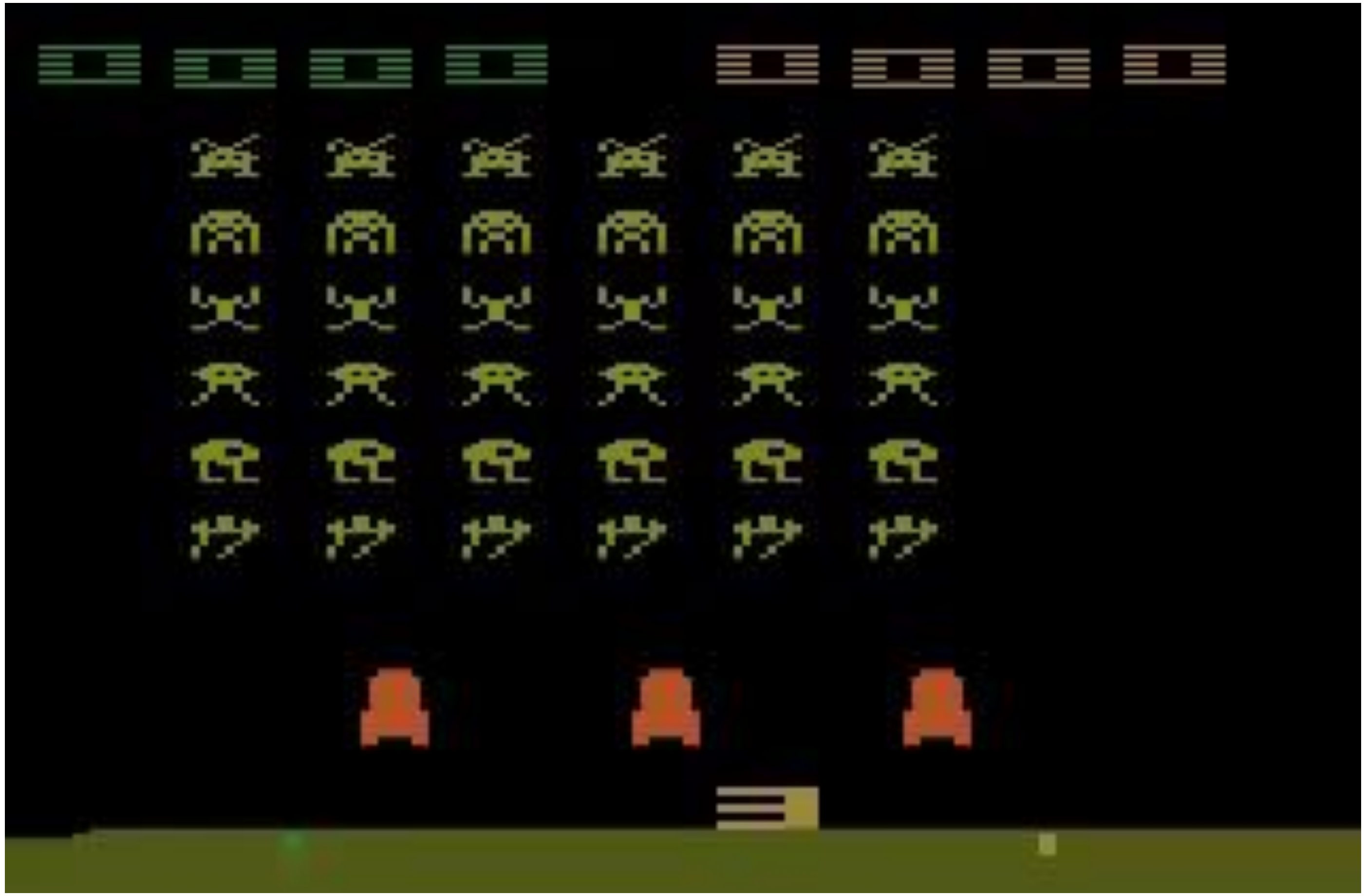
Beamrider



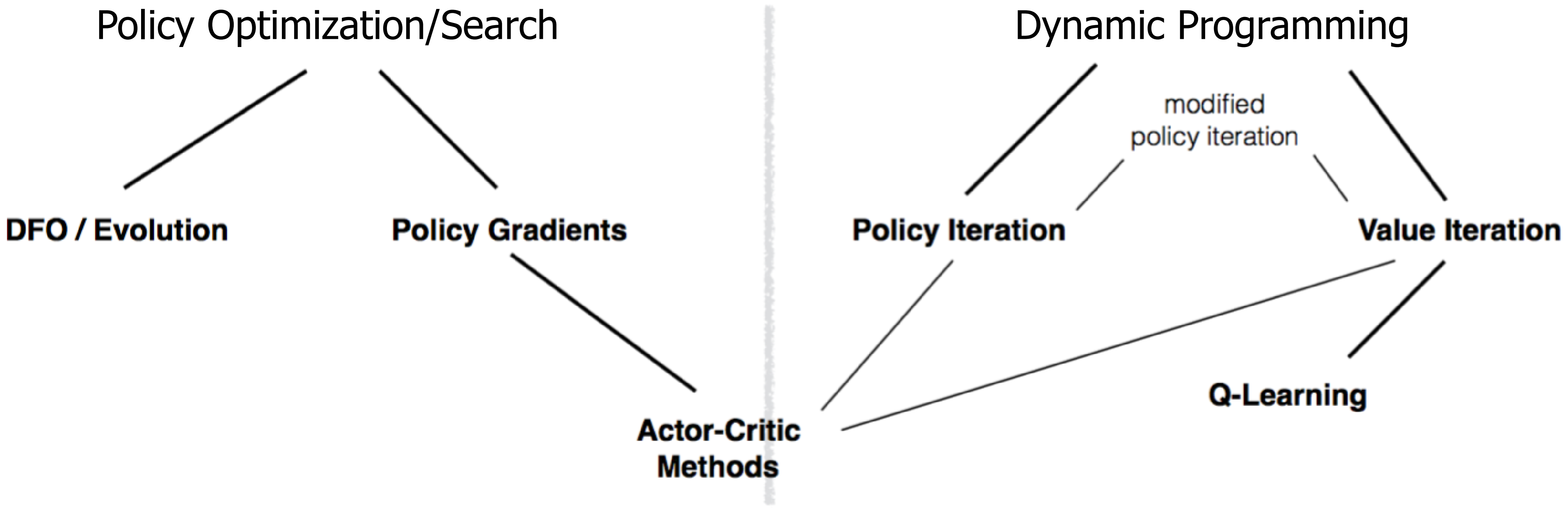
Q*bert

- Replace hand-crafted features with deep neural networks (CNN with 3M parameters now)
- Map pixels to actions
- Same approximate Q-Learning algorithm with effective tricks (e.g., use experience replay)

Deep Q-Networks [Mnih et al, 2013]



RL landscape



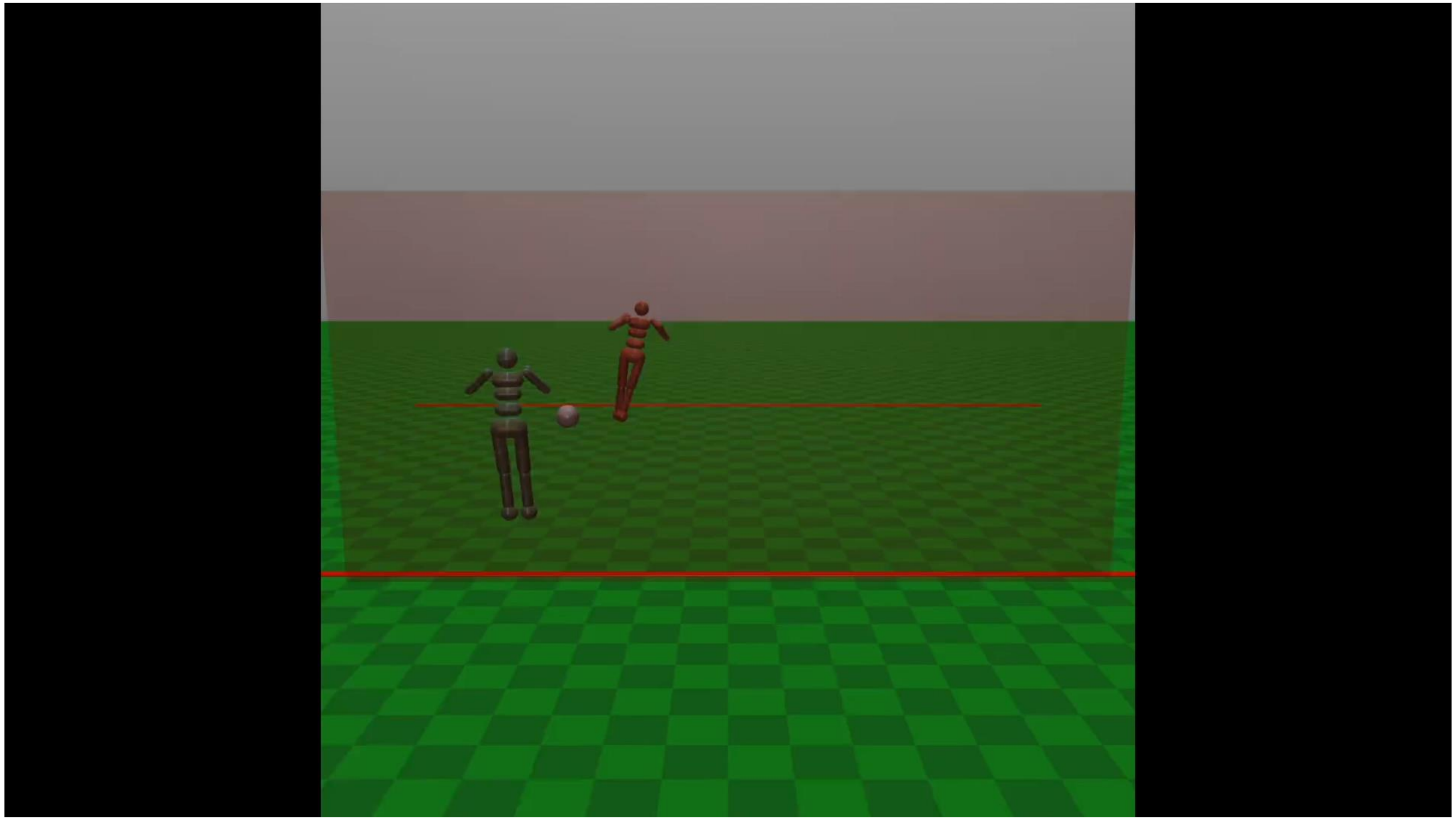
Policy search

- Often the feature-based policies that work well (win games, maximize utilities) aren't the ones that approximate V/Q best
- Policy search: Instead of learning V/Q values, directly learn policies that maximize rewards
  - Simplest approach (*derivative-free optimization*)
    - Start with some parameterized policy $\pi_\theta(\boldsymbol{a}|\boldsymbol{s})$
    - Perturb the weights a bit and see if the policy improves
  - *Policy gradient methods*: Train a parameterized policy $\pi_\theta(\boldsymbol{a}|\boldsymbol{s})$ using gradient ascent
  - Can be combined with value learning (actor-critic)
- Issues
  - How do we tell the policy got better?
  - Need to runa many sample episodes!

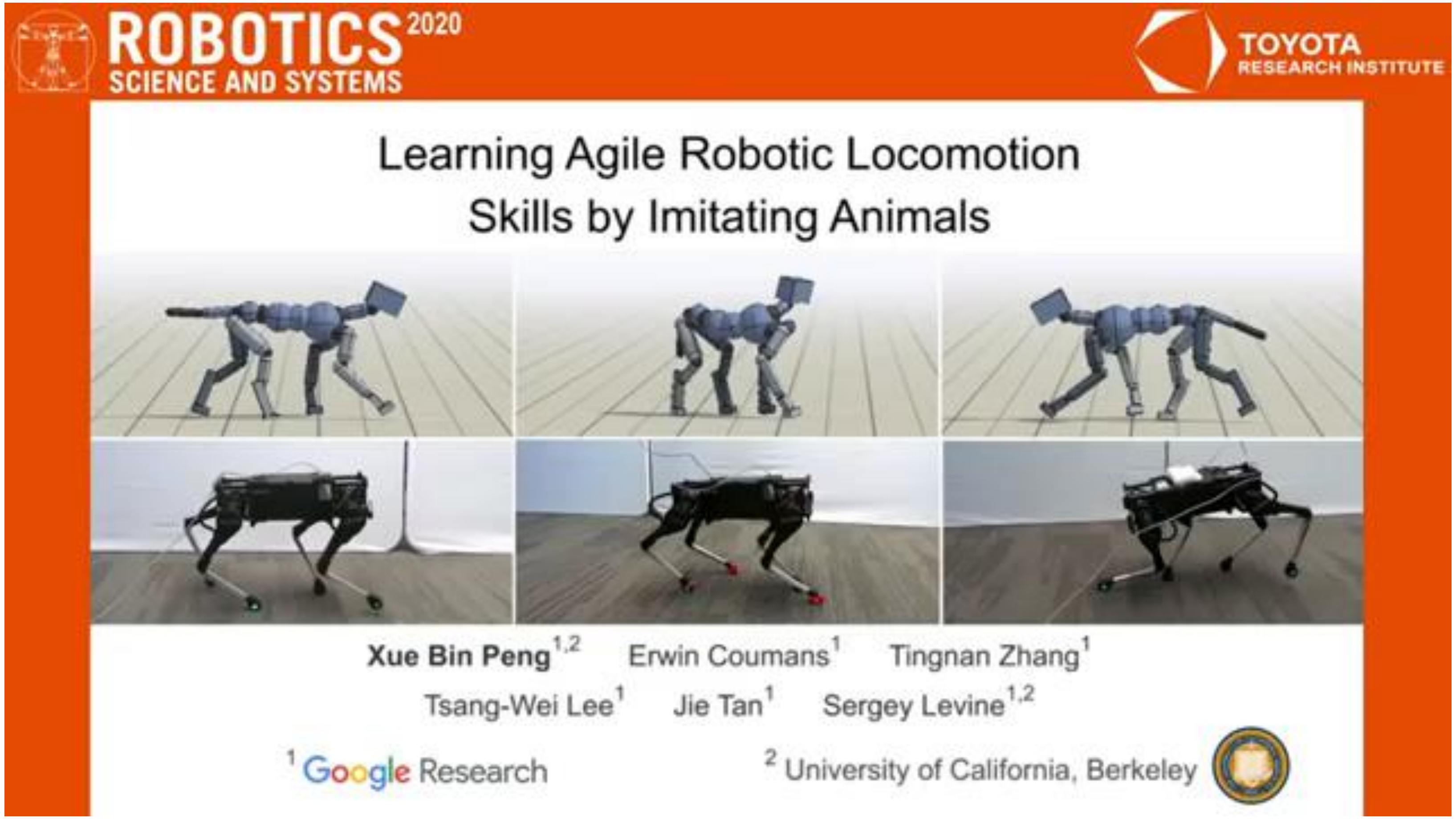
See Andrej Karpathy's "Pong from Pixels" blog for an introduction

Learning to walk [Schulman et al, 2016]

Playing soccer [Bansal et al, 2017]



Learning robotic locomotion skills [Peng et al, 2020]



Learning dexterous manipulation [Rajeswaran et al. 2018]

What an RL agent should optimize for?

max_{π}