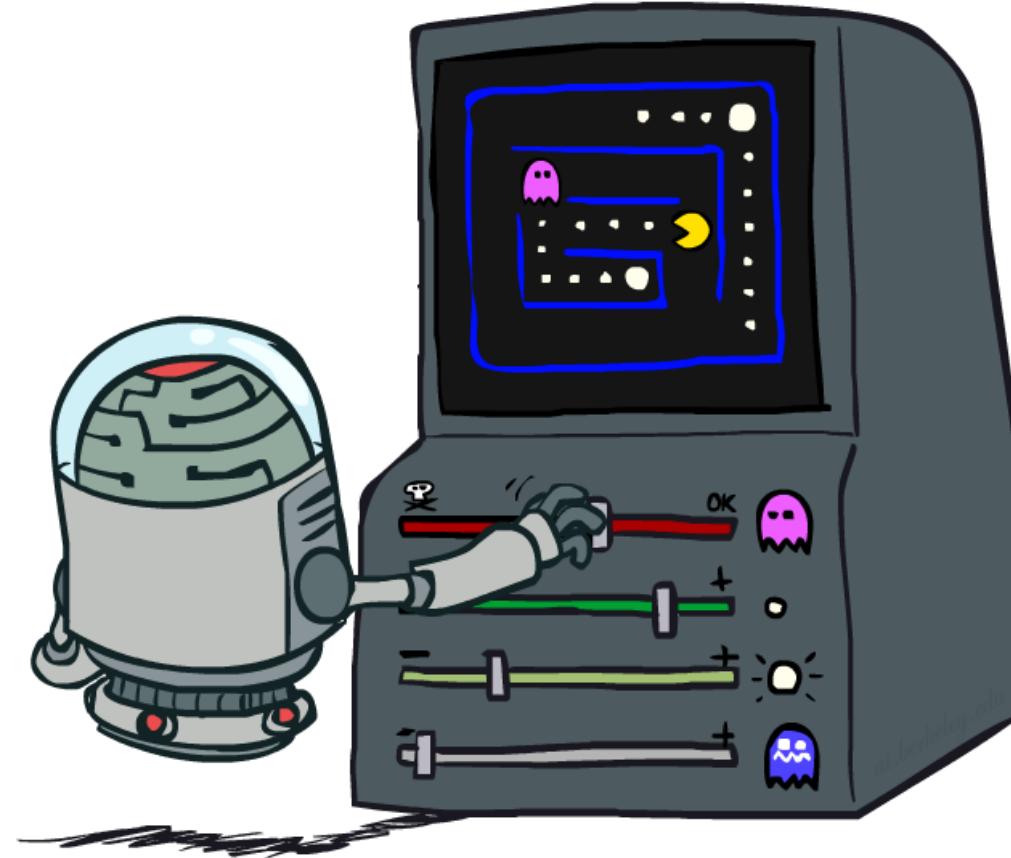


CS 5522: Artificial Intelligence II

Reinforcement Learning III (Advanced Topics)



Instructor: Wei Xu

Ohio State University

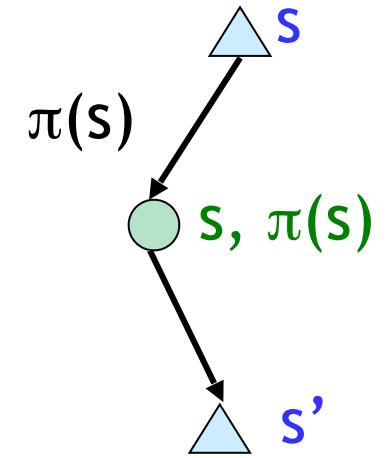
Reinforcement Learning

- We still assume an MDP:
 - A set of states $s \in S$
 - A set of actions (per state) A
 - A model $T(s,a,s')$
 - A reward function $R(s,a,s')$
- Still looking for a policy $\pi(s)$
- New twist: don't know T or R , so must try out actions
- Big idea: Compute all averages over T using sample outcomes



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



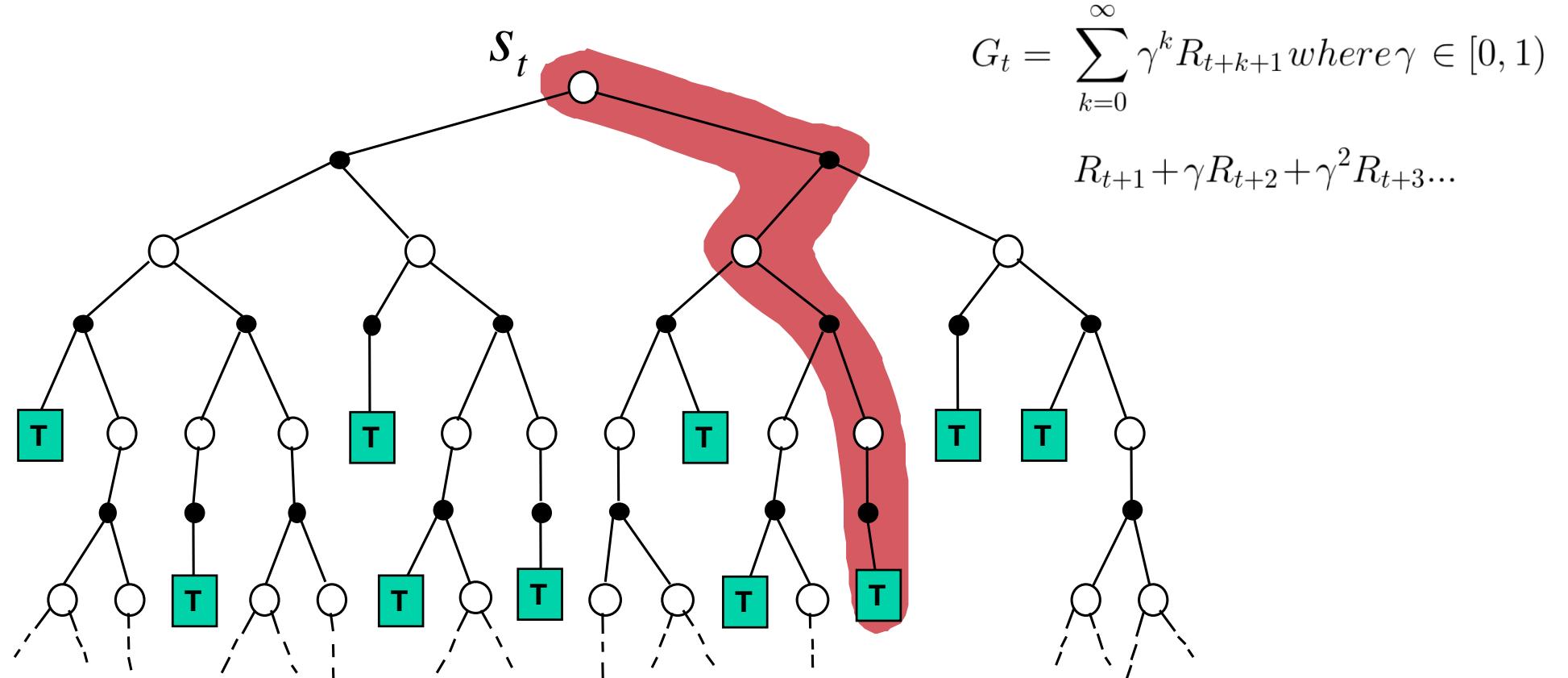
Sample of $V(s)$: $sample = R(s, \pi(s), s') + \gamma V^\pi(s')$

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

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

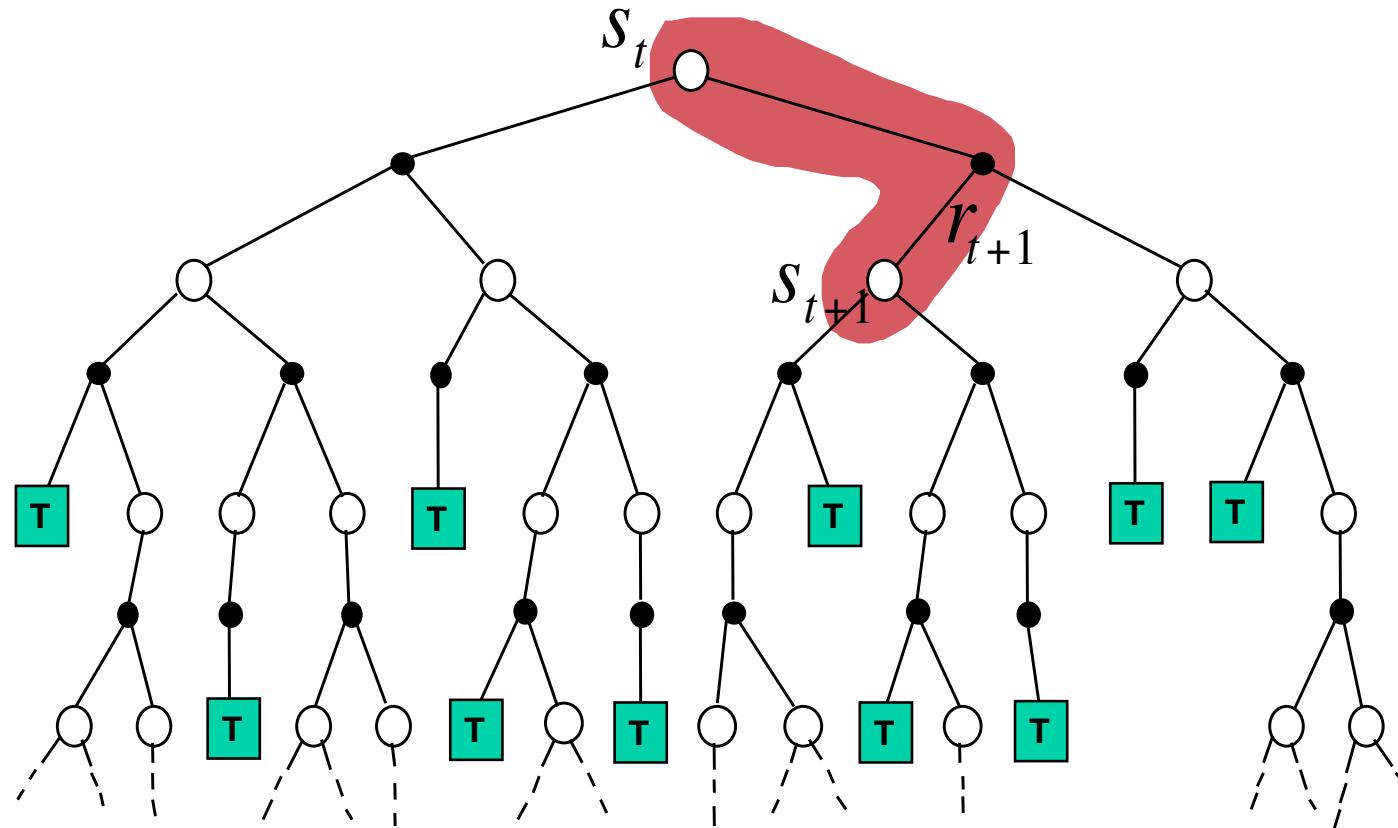
A Unified View*: Direct Estimation (Monte-Carlo)

$$V(S_t) \leftarrow V(S_t) + \alpha (G_t - V(S_t))$$



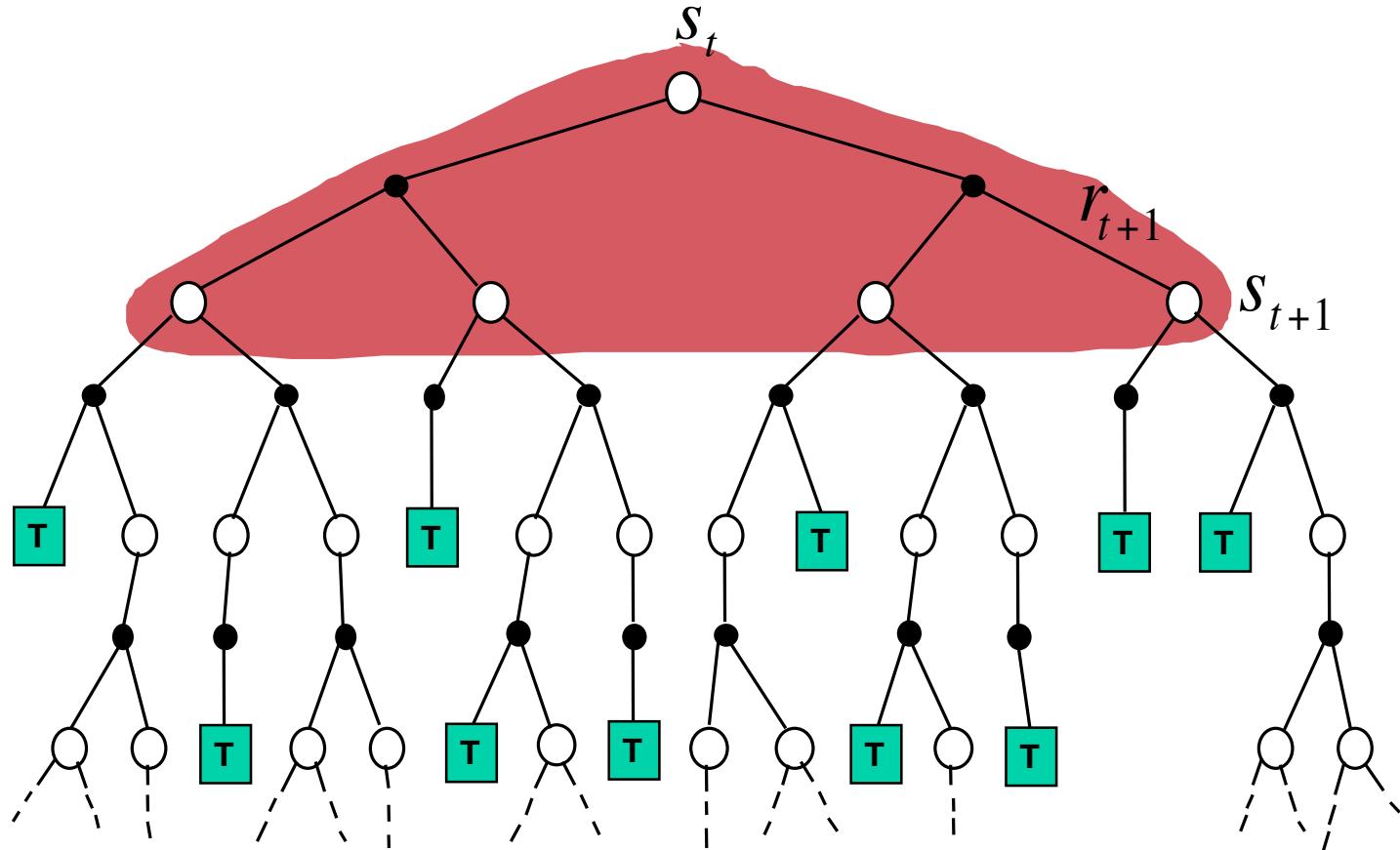
A Unified View*: Temporal-Difference

$$V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$



A Unified View*: Bellman Updates (Dynamic Programming)

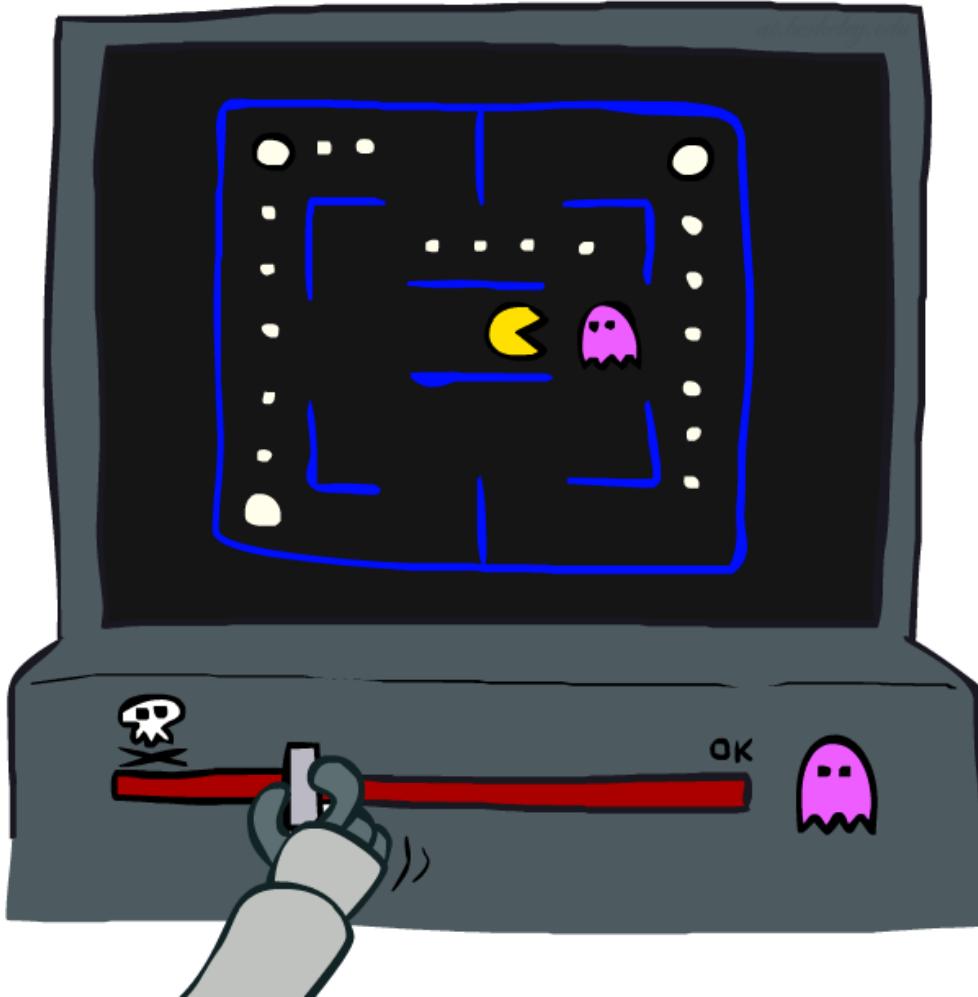
$$V(S_t) \leftarrow \mathbb{E}_\pi [R_{t+1} + \gamma V(S_{t+1})]$$



Large-Scale Reinforcement Learning

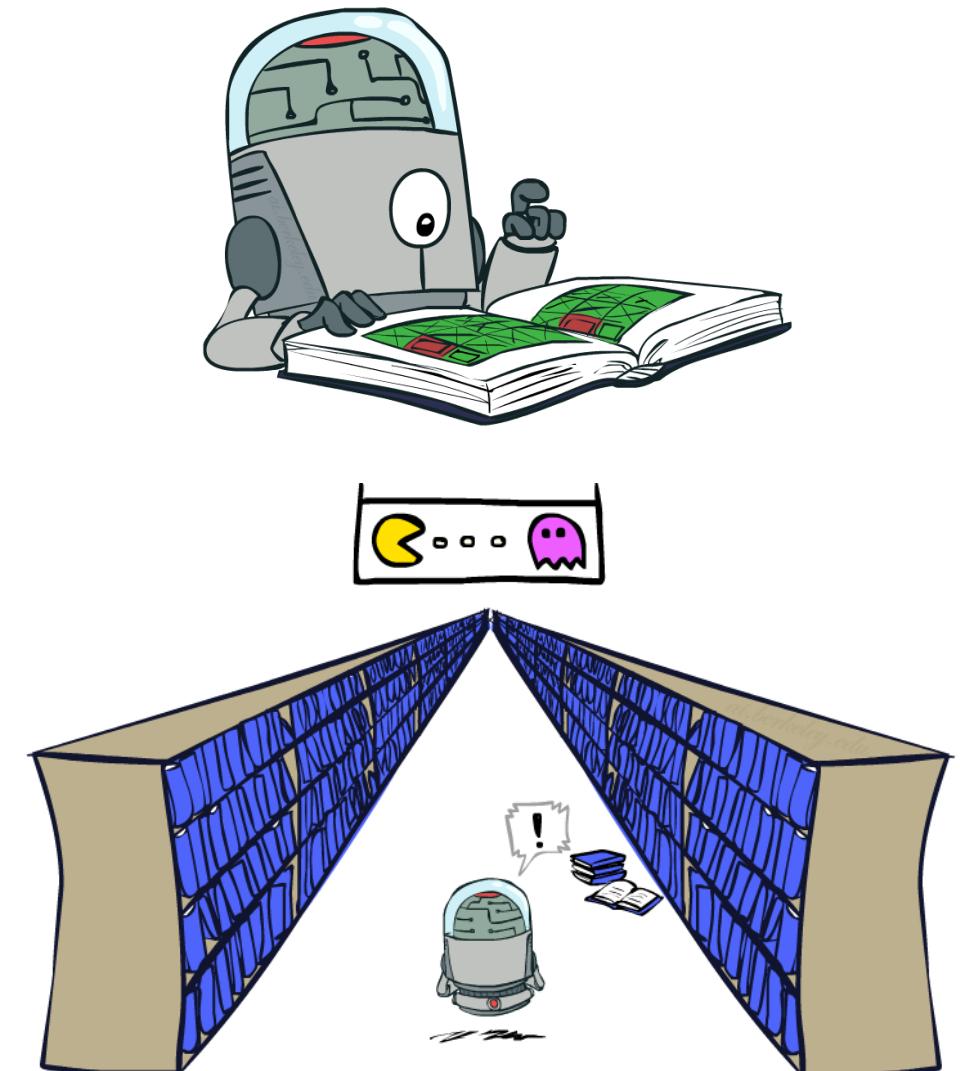
- Reinforcement learning can be used to solve *large* problems
 - Backgammon: 10^{20} states
 - Go: 10^{170} states
 - Helicopter: continuous state space
- Solution:
 - Estimate value function with **function approximation**
 - Generalize from seen states to unseen states
 - Update (a small number of) parameters using temporal-difference learning.

Approximate Q-Learning



Generalizing Across States

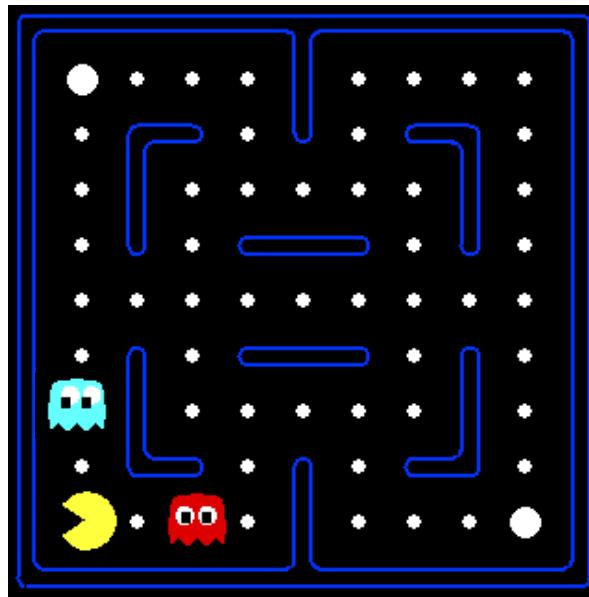
- Basic Q-Learning keeps a table of all q-values
- In realistic situations, we cannot possibly learn about every single state!
 - Too many states to visit them all in training
 - Too many states to hold the q-tables in memory
- Instead, we want to generalize:
 - Learn about some small number of training states from experience
 - Generalize that experience to new, similar situations
 - This is a fundamental idea in machine learning, and we'll see it over and over again



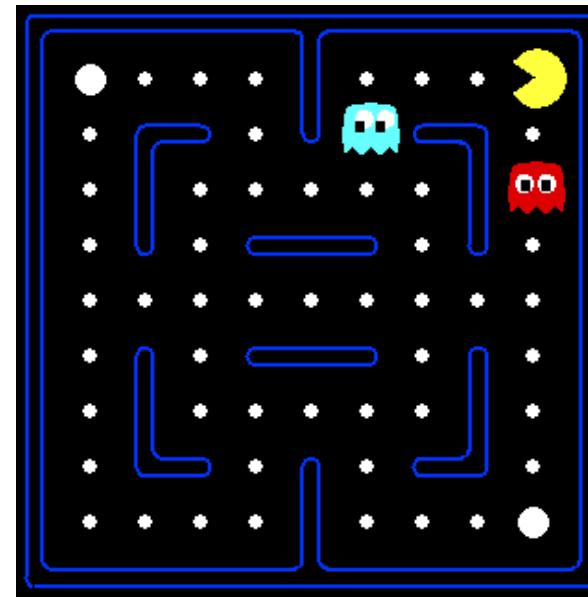
[demo - RL pacman]

Example: Pacman

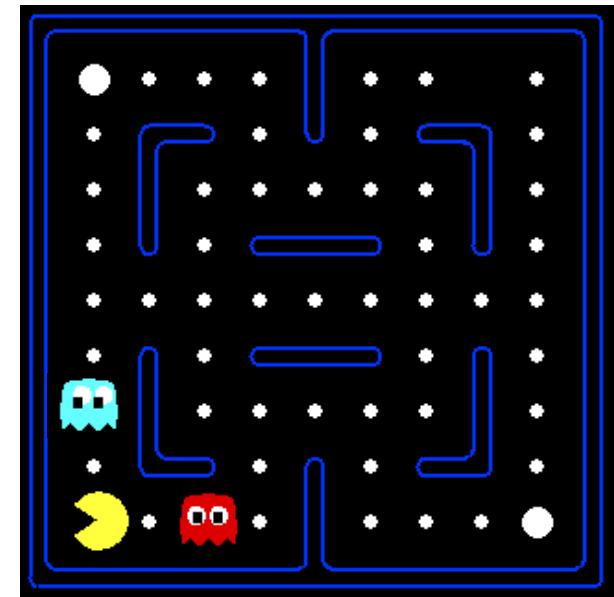
Let's say we discover through experience that this state is bad:



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



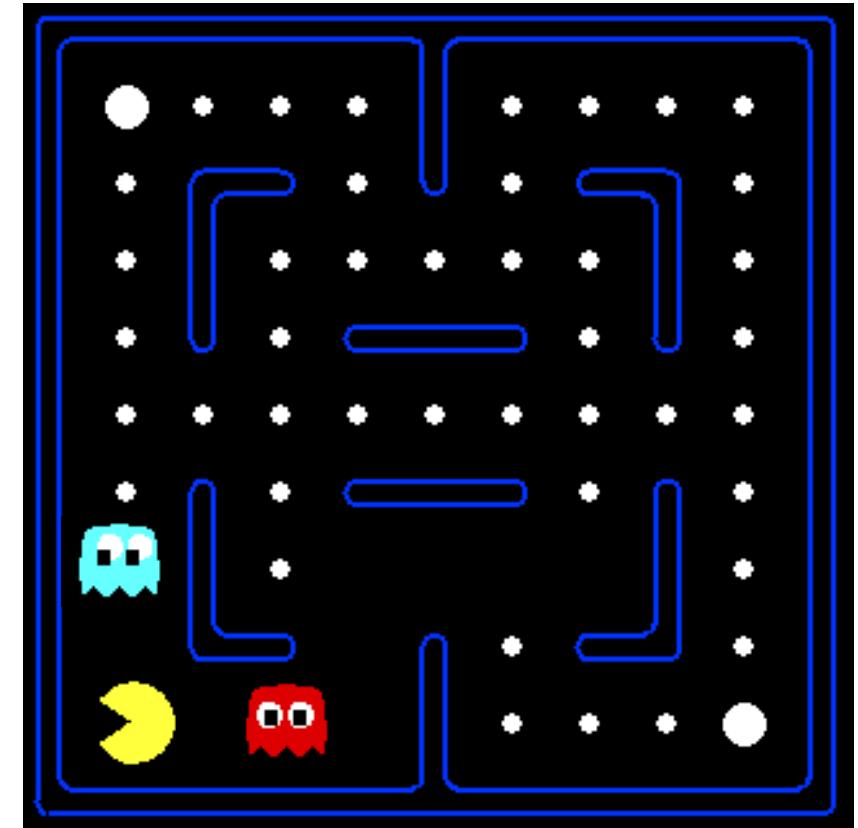
Or even this one!



[Demo: Q-learning - pacman - tiny - watch all (L11D5)]
[Demo: Q-learning - pacman - tiny - silent train (L11D6)]
[Demo: Q-learning - pacman - tricky - watch all (L11D7)]

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
 - $1 / (\text{dist to dot})^2$
 - 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:

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

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

- 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

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

- Q-learning with linear Q-functions:

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

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

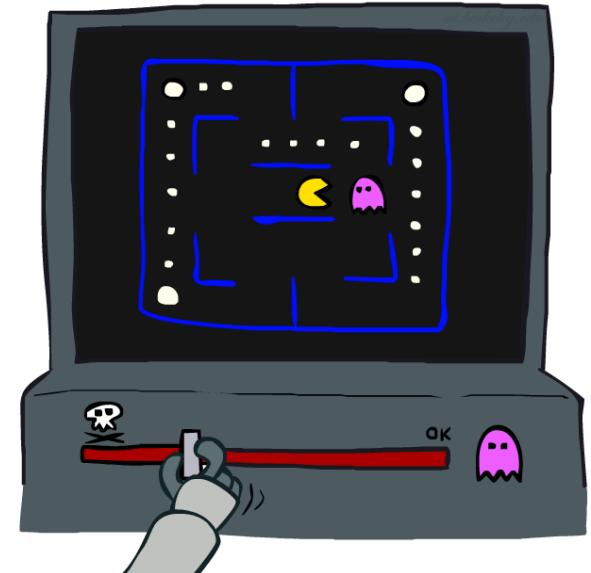
$Q(s, a) \leftarrow Q(s, a) + \alpha \text{ [difference]}$ Exact Q's

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

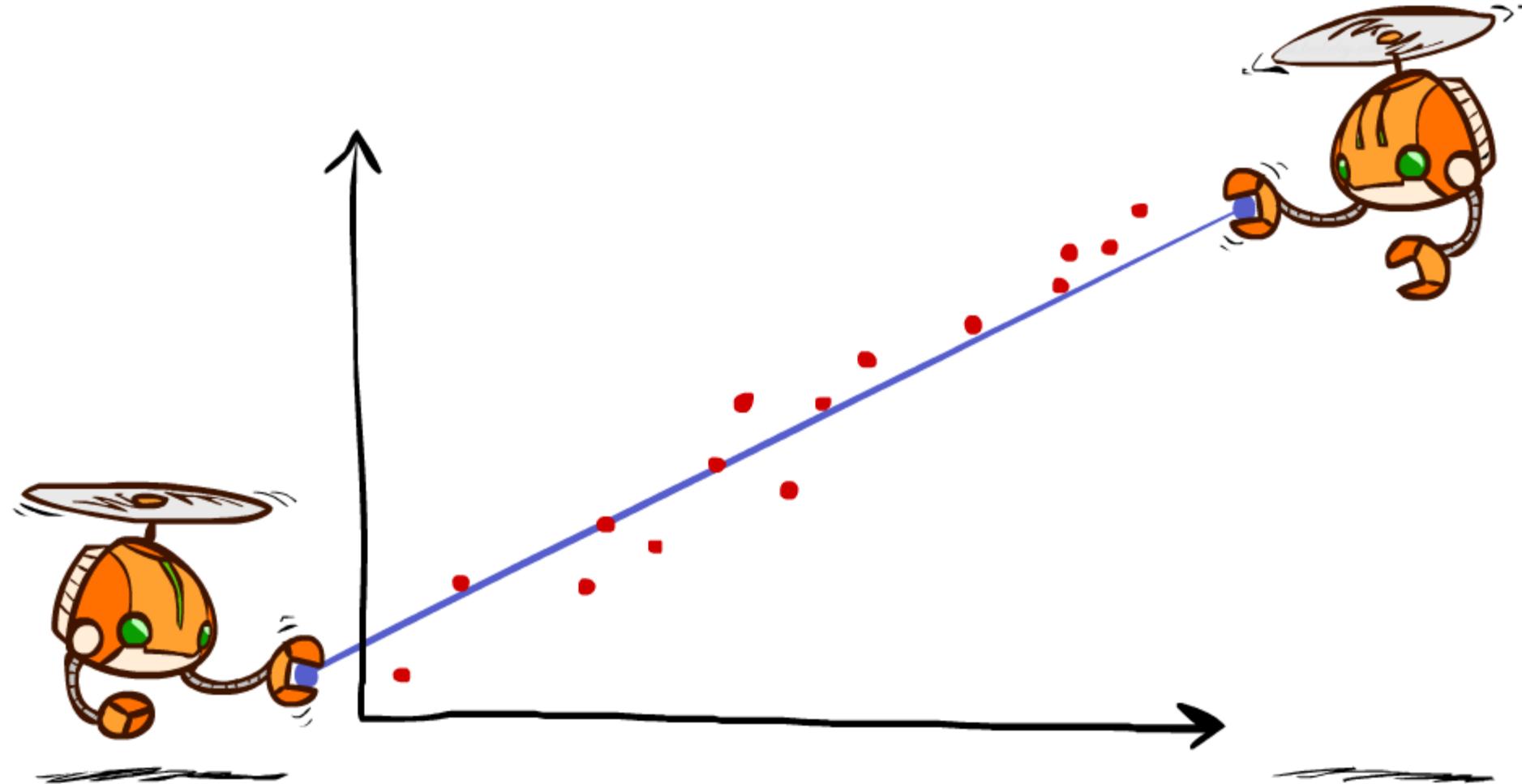
- Intuitive interpretation:

- 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

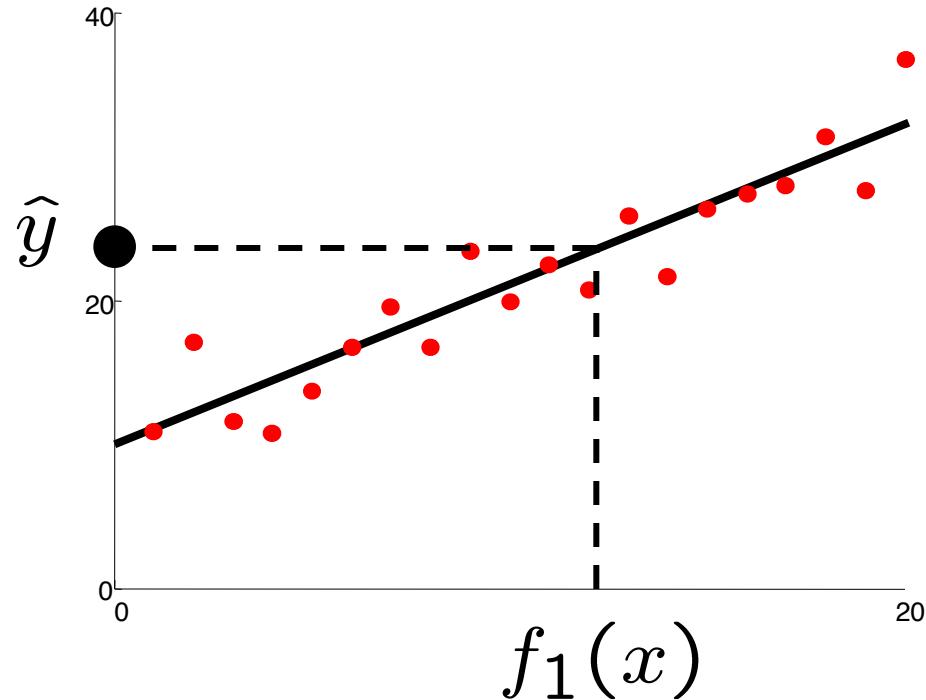
- Formal justification: online least squares



Q-Learning and Least Squares

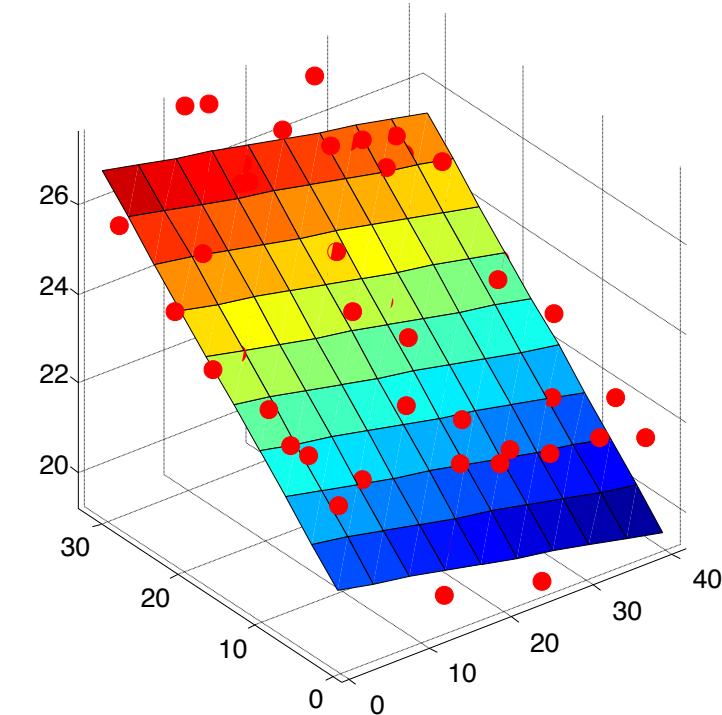


Linear Approximation: Regression*



Prediction:

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

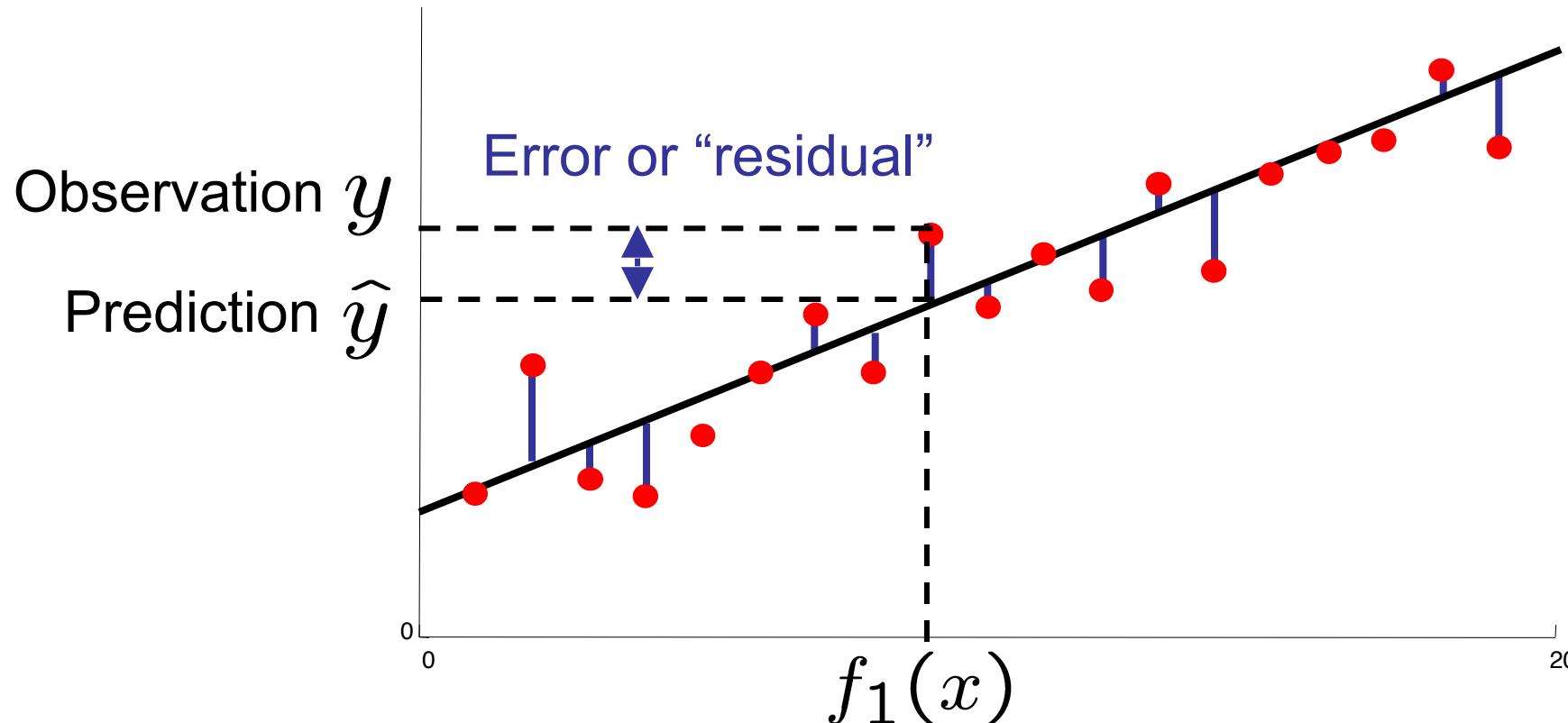


Prediction:

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

Optimization: Least Squares*

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



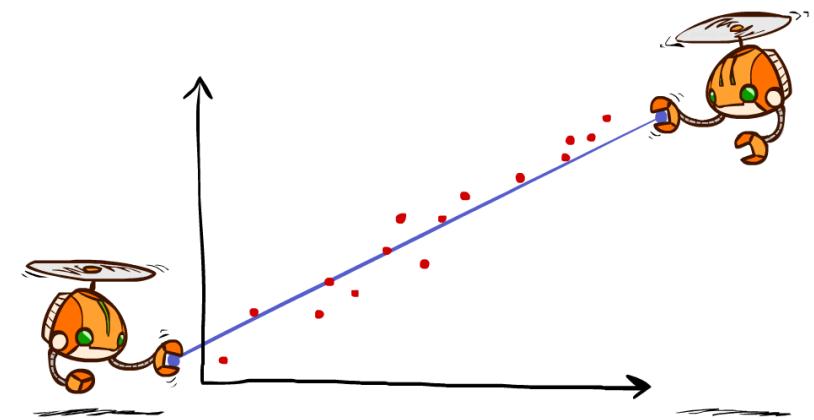
Minimizing Error

Imagine we had only one point x , with features $f(x)$, target value y , and weights w ; minimizing least square errors by Stochastic Gradient Decent:

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

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

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



Approximate q update explained:

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

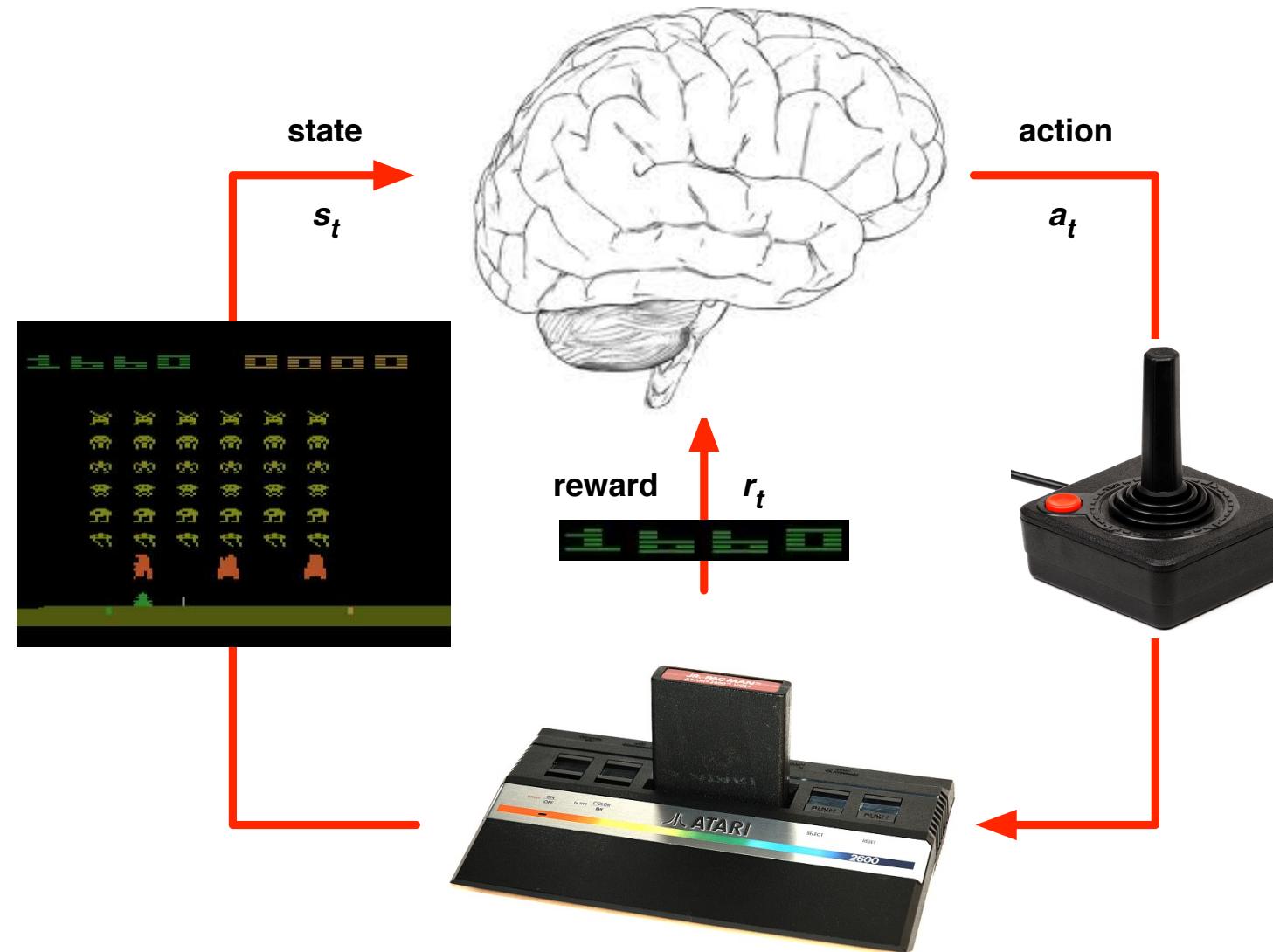
“target”

“prediction”

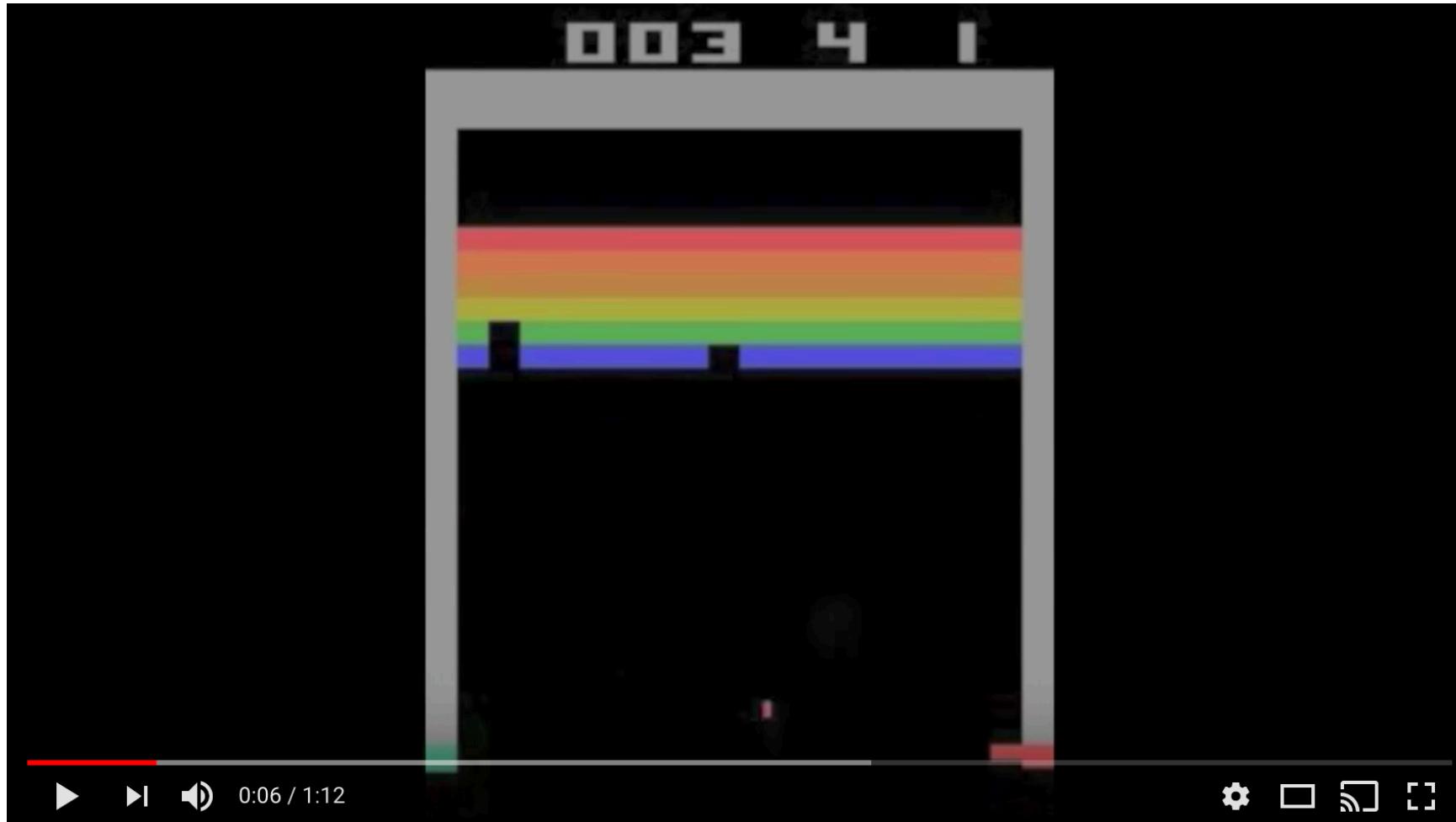
Various Function Approximator *

- We can consider differentiable function approximators, e.g.
 - Linear combinations of features
 - Neural network
 - Decision tree
 - Nearest neighbor
 - ...

Deep Reinforcement Learning in Atari *

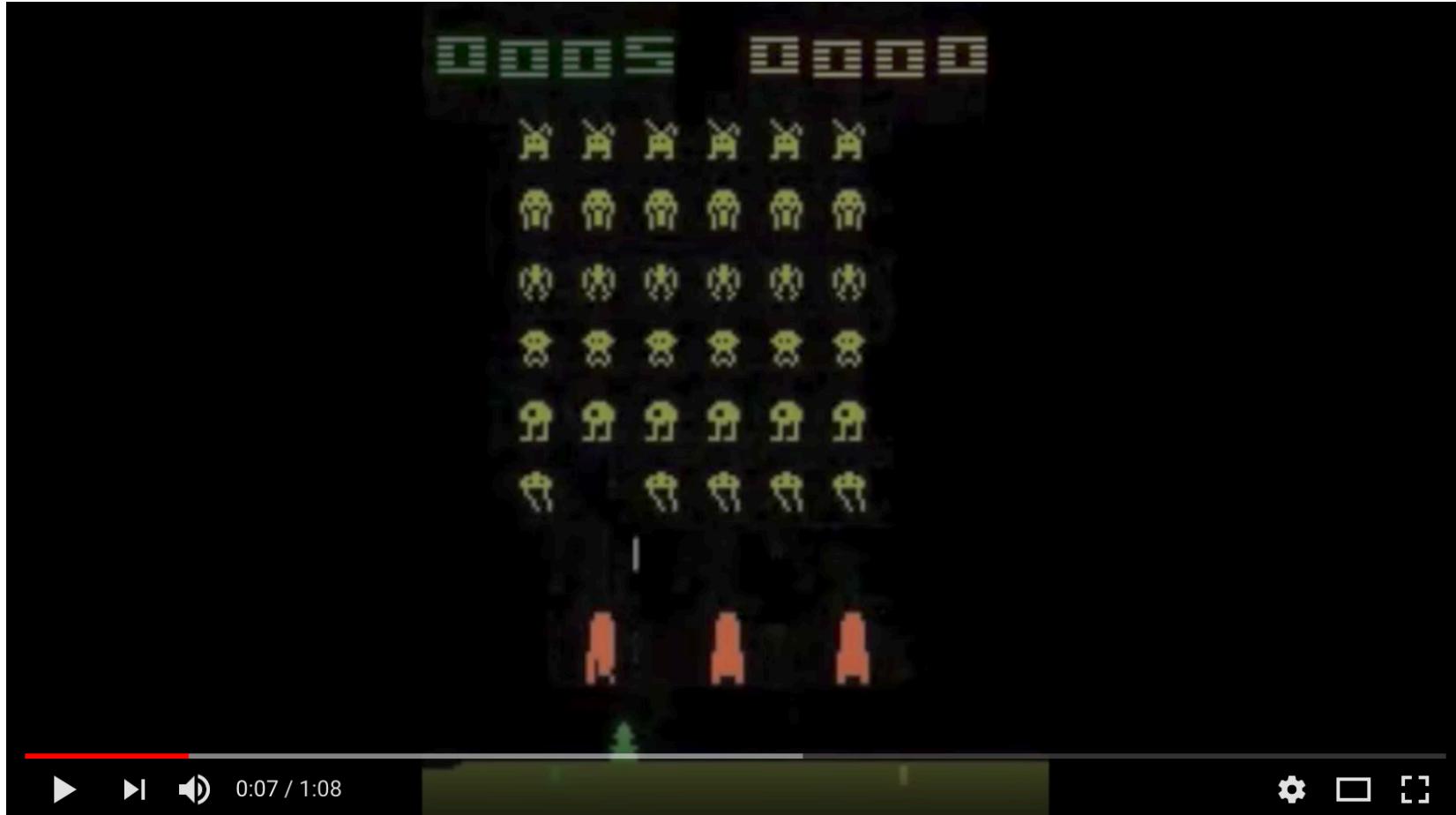


Demo: DQN for Atari Breakout*



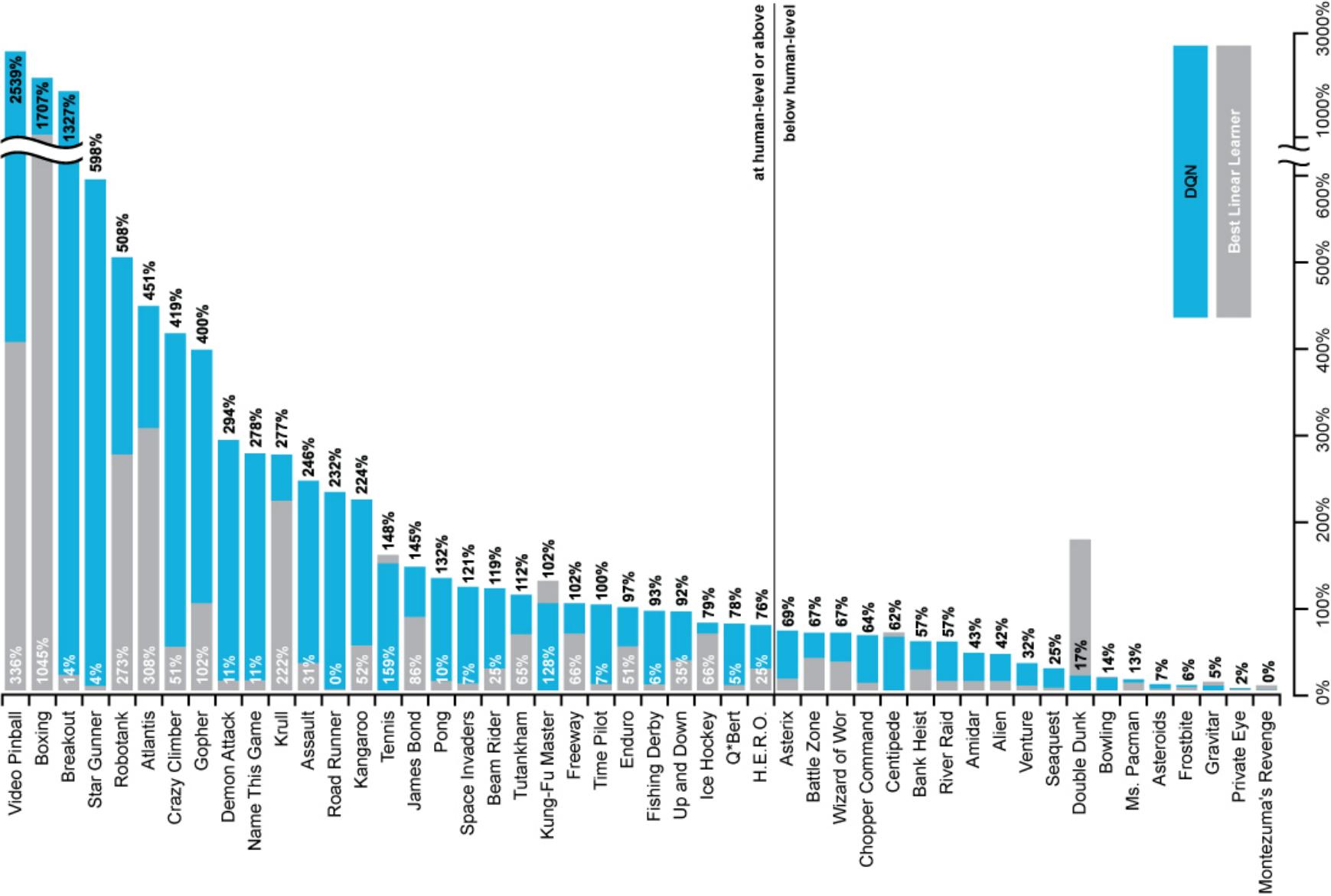
<https://www.youtube.com/watch?v=TmPfTpjtdgg>

Demo: DQN for Space Invaders*



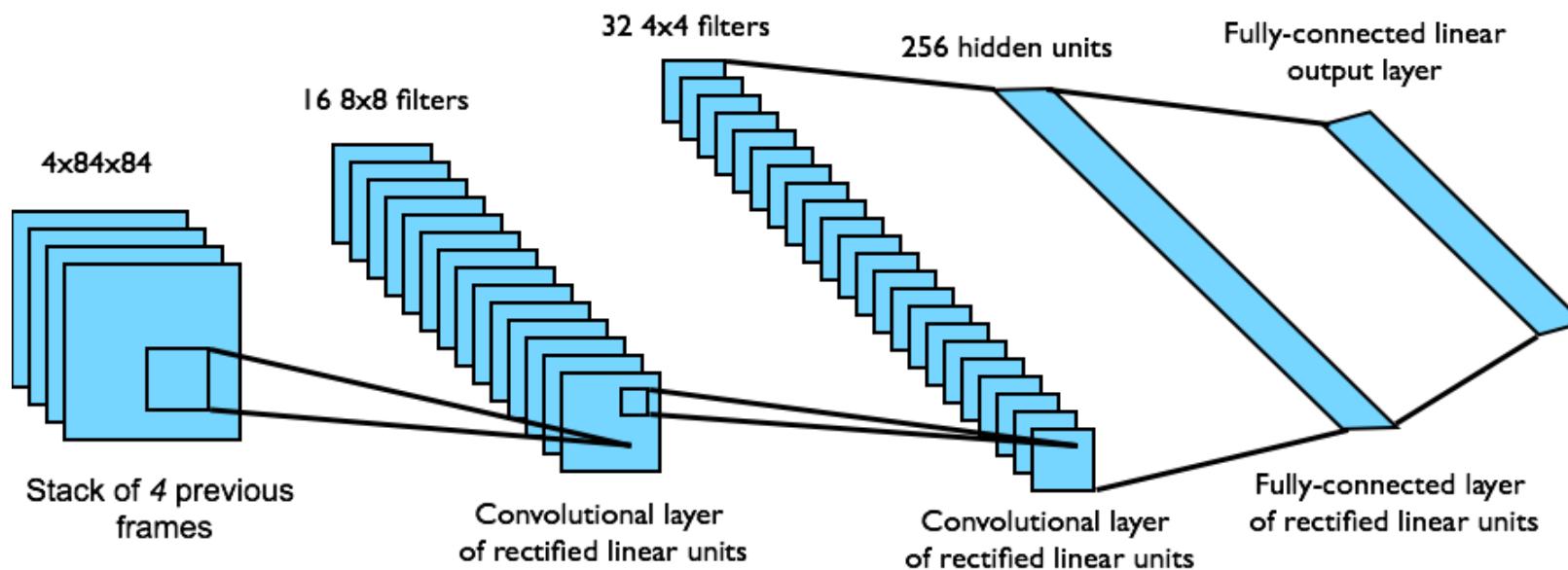
<https://www.youtube.com/watch?v=W2CAghUiofY>

DQN Results in Atari *



Deep Q-Network (DQN) in Atari *

- ▶ End-to-end learning of values $Q(s, a)$ from pixels s
- ▶ Input state s is stack of raw pixels from last 4 frames
- ▶ Output is $Q(s, a)$ for 18 joystick/button positions
- ▶ Reward is change in score for that step



Network architecture and hyperparameters fixed across all games

[Mnih et al. 2013; 2016]

Deep Q-Network (DQN) in Atari *

DQN uses **experience replay** and **fixed Q-targets**

- Take action a_t according to ϵ -greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory \mathcal{D}
- Sample random mini-batch of transitions (s, a, r, s') from \mathcal{D}
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning targets

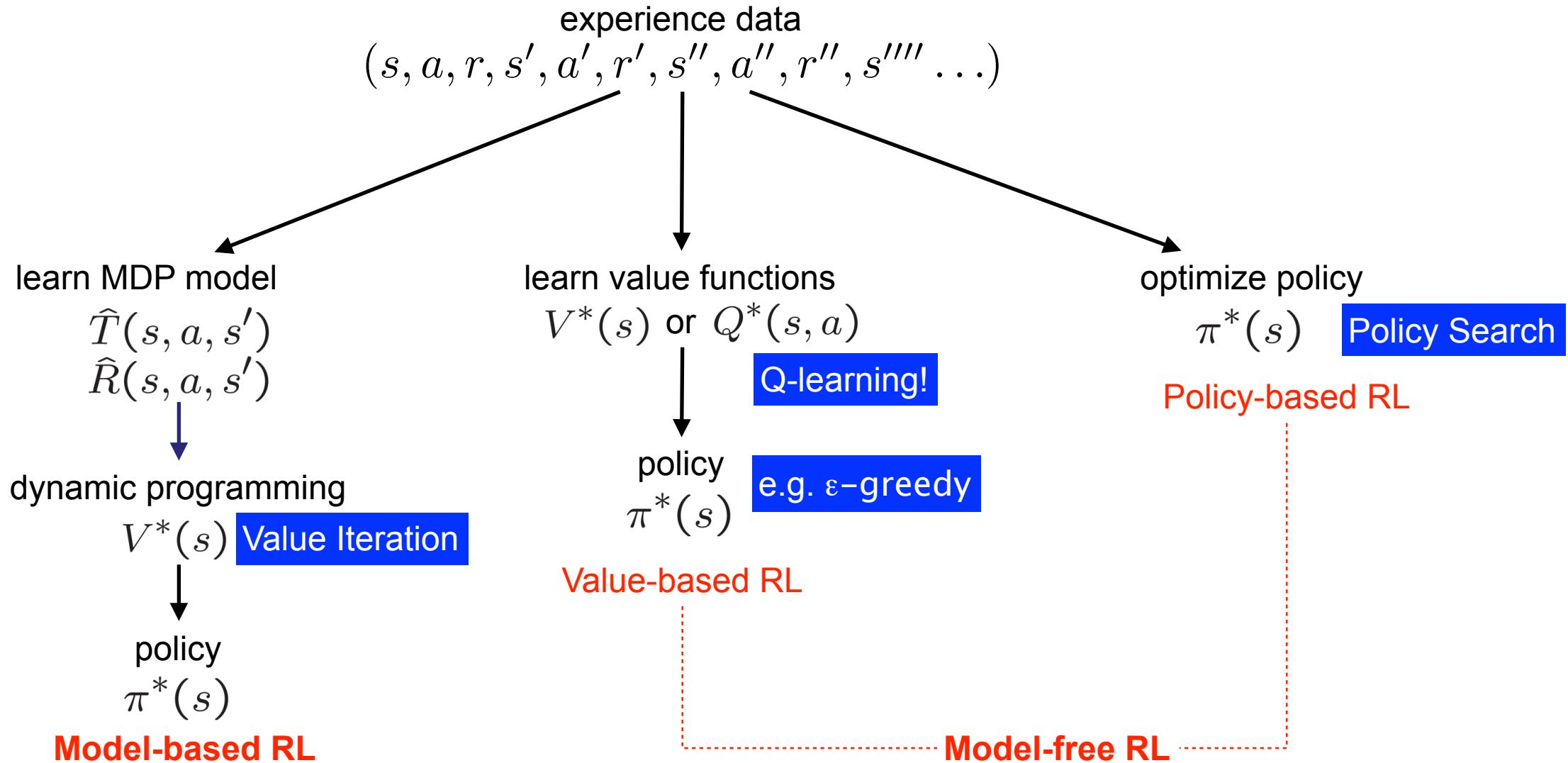
$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i} \left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i) \right)^2 \right]$$

- Using variant of stochastic gradient descent

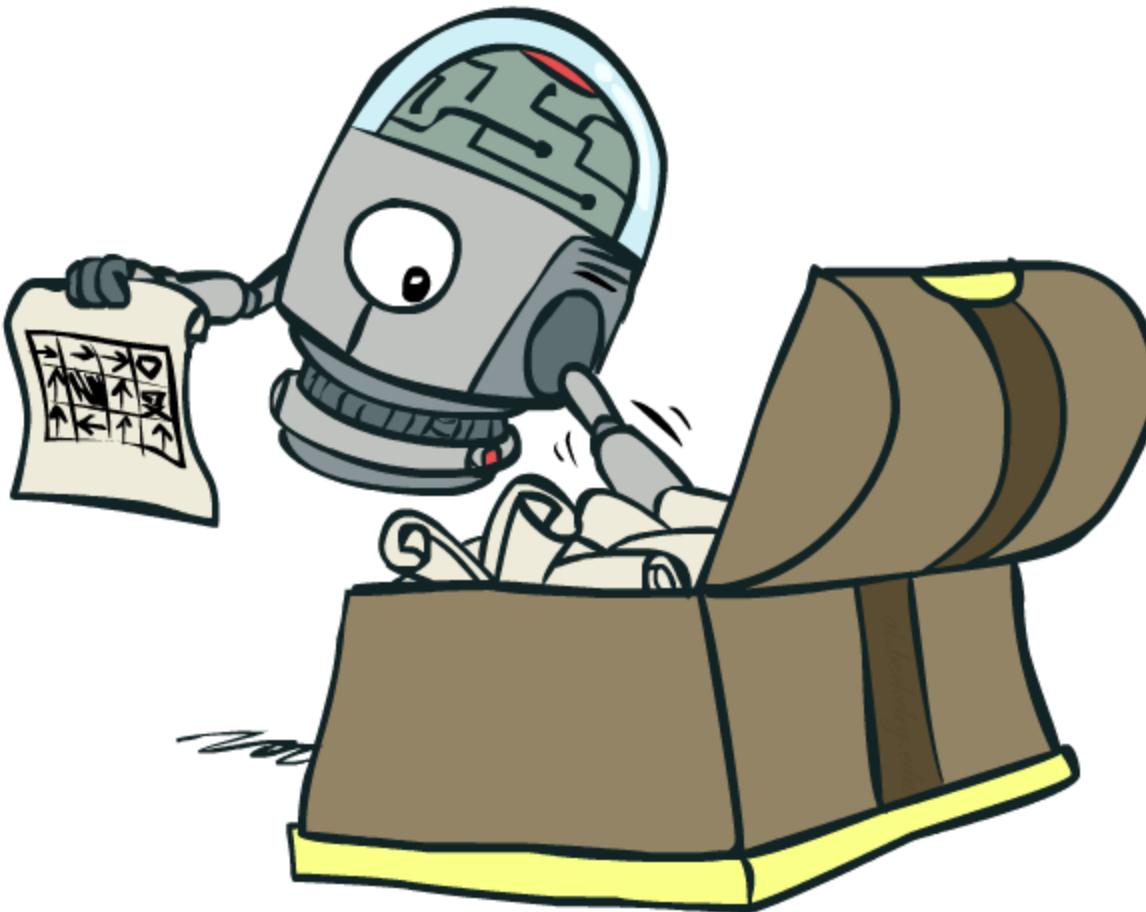
Deep Q-Network (DQN) in Atari *

	Replay Fixed-Q	Replay Q-learning	No replay Fixed-Q	No replay Q-learning
Breakout	316.81	240.73	10.16	3.17
Enduro	1006.3	831.25	141.89	29.1
River Raid	7446.62	4102.81	2867.66	1453.02
Seaquest	2894.4	822.55	1003	275.81
Space Invaders	1088.94	826.33	373.22	301.99

Summary: Approaches To Reinforcement Learning



Policy Search

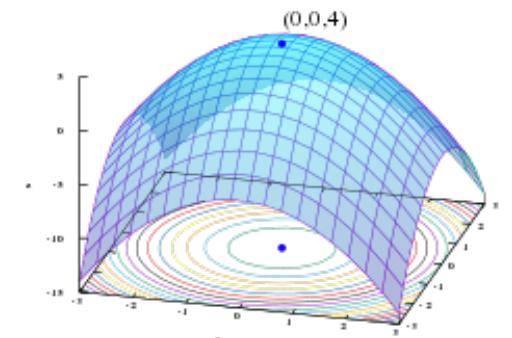


Policy Search

- 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)
 - We'll see this distinction between modeling and prediction again later in the course
- 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

- Simple Idea: start with an ok solution then fine-tune by hill climbing on feature weights
 - 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
- Hill climbing attempts to maximize (or minimize) a function $f(x)$:
 - x is a vector of continuous and/or discrete values
 - At each iteration, adjust a single element in x , and see whether it improves the value of $f(x)$
 - Accept any change that improves $f(x)$, and repeat
 - Note that hill climbing differs from gradient ascent methods, which adjust all the elements in x at each iteration according to the gradient of the hill.



Why Policy Optimization ?

- Often the feature-based policies that work well (win games, maximize utilities) aren't the ones that approximate V/Q best
- Often π can be simpler than Q or V:
 - Q-learning: need to efficiently solve $\arg \max_a Q_w(s, a)$
 - Challenge for continuous / high-dimensional action spaces (e.g. robotic grasp)
- Can learn stochastic policies
 - We can also parametrize the policy $\pi_\theta(s, a) = P(a|s, \theta)$

Stochastic Policy *



- Two-player game of rock-paper-scissors
 - Scissors beats paper
 - Rock beats scissors
 - Paper beats rock
- Consider policies for *iterated* rock-paper-scissors
 - A deterministic policy is easily exploited
 - A uniform random policy is optimal

Policy Objective Functions *

- Goal: given policy $\pi_\theta(s, a)$ with parameters θ , find best θ
- But how do we measure the quality of a policy π_θ ?
- In episodic environments we can use the **start value**

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta} [v_1]$$

- In continuing environments we can use the **average value**

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

- Or the **average reward per time-step**

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) \mathcal{R}_s^a$$

- where $d^{\pi_\theta}(s)$ is **stationary distribution** of Markov chain for π_θ

Policy Gradient *

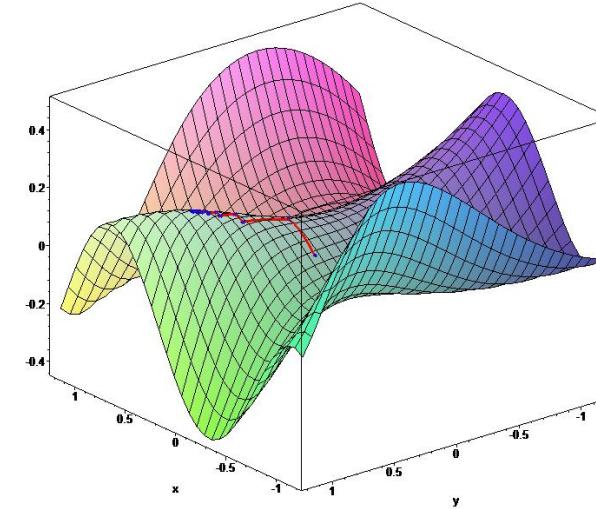
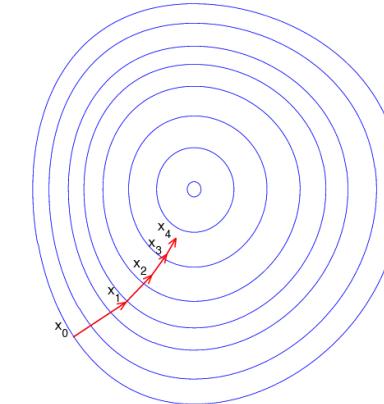
- Let $J(\theta)$ be any policy objective function
- Policy gradient algorithms search for a *local* maximum in $J(\theta)$ by ascending the gradient of the policy, w.r.t. parameters θ

$$\Delta\theta = \alpha \nabla_{\theta} J(\theta)$$

- Where $\nabla_{\theta} J(\theta)$ is the **policy gradient**

$$\nabla_{\theta} J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$$

- and α is a step-size parameter



Computing Gradients by Finite Differences *

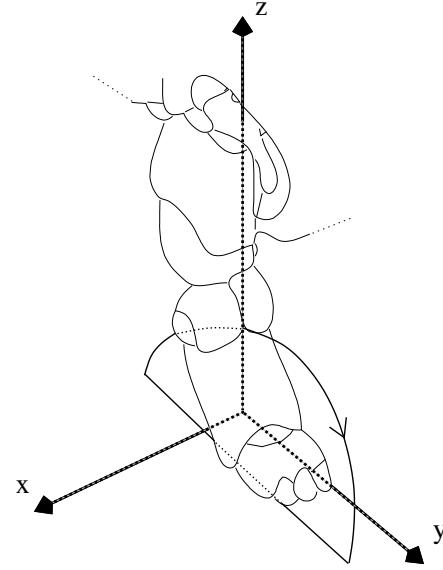
- To evaluate policy gradient of $\pi_\theta(s, a)$
- For each dimension $k \in [1, n]$
 - Estimate k th partial derivative of objective function w.r.t. θ
 - By perturbing θ by small amount ϵ in k th dimension

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon}$$

where u_k is unit vector with 1 in k th component, 0 elsewhere

- Uses n evaluations to compute policy gradient in n dimensions
- Simple, noisy, inefficient - but sometimes effective
- Works for arbitrary policies, even if policy is not differentiable

Example: AIBO Walk by Finite Difference Policy Gradient



- Goal: learn a fast AIBO walk (useful for Robocup)
- AIBO walk policy is controlled by 12 numbers (elliptical loci)
- Adapt these parameters by finite difference policy gradient
- Evaluate performance of policy by field traversal time

Likelihood Ratio Policy Gradient *

- We now compute the policy gradient *analytically*
- Assume policy π_θ is differentiable whenever it is non-zero
- and we know the gradient $\nabla_\theta \pi_\theta(s, a)$
- **Likelihood ratios** exploit the following identity

$$\begin{aligned}\nabla_\theta \pi_\theta(s, a) &= \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} \\ &= \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a)\end{aligned}$$

- The **score function** is $\nabla_\theta \log \pi_\theta(s, a)$

Softmax Policy *

- Weight actions using linear combination of features $\phi(s, a)^\top \theta$
- Probability of action is proportional to exponentiated weight

$$\pi_\theta(s, a) \propto e^{\phi(s, a)^\top \theta}$$

- The score function is

$$\nabla_\theta \log \pi_\theta(s, a) = \phi(s, a) - \mathbb{E}_{\pi_\theta} [\phi(s, \cdot)]$$

One-step MDPs *

- Consider a simple class of **one-step** MDPs
 - Starting in state $s \sim d(s)$
 - Terminating after one time-step with reward $r = \mathcal{R}_{s,a}$
- Use likelihood ratios to compute the policy gradient

$$J(\theta) = \mathbb{E}_{\pi_\theta} [r]$$

$$= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s, a) \mathcal{R}_{s,a}$$

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) \mathcal{R}_{s,a} \\ &= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) r]\end{aligned}$$

Policy Gradient Theorem *

- The policy gradient theorem generalises the likelihood ratio approach to multi-step MDPs
- Replaces instantaneous reward r with long-term value $Q^\pi(s, a)$
- Policy gradient theorem applies to start state objective, average reward and average value objective

Theorem

*For any differentiable policy $\pi_\theta(s, a)$,
for any of the policy objective functions $J = J_1, J_{avR}$, or $\frac{1}{1-\gamma}J_{avV}$,
the policy gradient is*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a)]$$

Monte-Carlo Policy Gradient (REINFORCE)*

- Update parameters by stochastic gradient ascent
- Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

```
function REINFORCE
    Initialise  $\theta$  arbitrarily
    for each episode  $\{s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$  do
        for  $t = 1$  to  $T - 1$  do
             $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$ 
        end for
    end for
    return  $\theta$ 
end function
```

(Williams, 1992)

Example: Policy Optimization Success Stories



Kohl and Stone, 2004



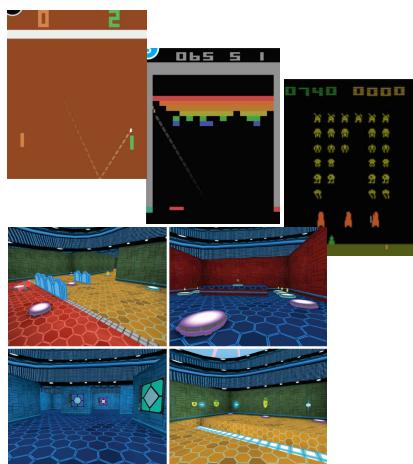
Ng et al, 2004



Tedrake et al, 2005



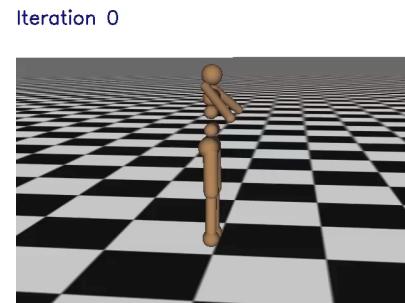
Kober and Peters, 2009



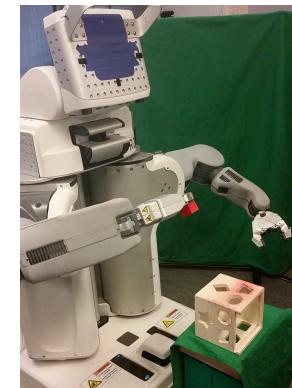
Mnih et al, 2015
(A3C)



Silver et al, 2014
(DPG)
Lillicrap et al, 2015
(DDPG)



Schulman et al,
2016 (TRPO + GAE)



Levine*, Finn*, et
al, 2016
(GPS)



Silver*, Huang*, et
al, 2016
(AlphaGo**)

Flying a Helicopter



How to: fly a helicopter

<https://www.youtube.com/watch?v=l9ScVZ437xA&t=45s>

<https://www.youtube.com/watch?v=icXbEiUg9i0> (RC helicopter)

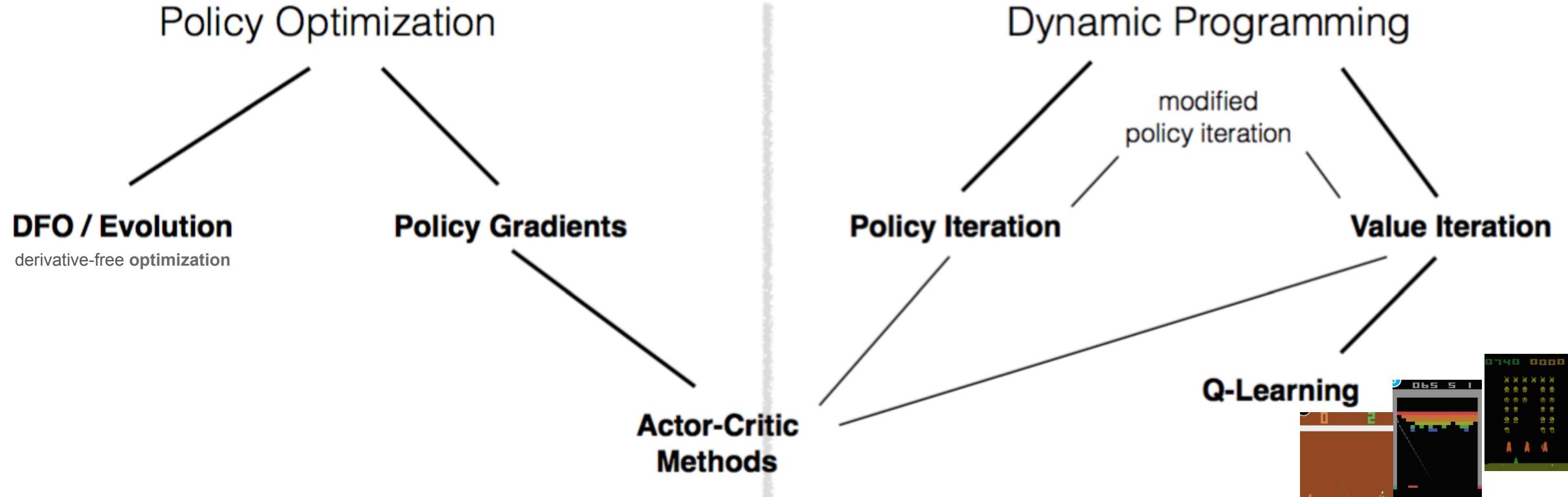
Demo: Autonomous Helicopter



<https://www.youtube.com/watch?v=VCdxqn0fcnE>

[Ng et al. 2004; Abbeel et al. 2010]

Reinforcement Learning Landscape *



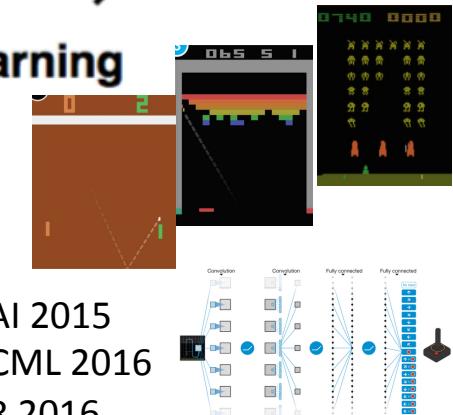
DQN: Mnih et al, Nature 2015

Double DQN: Van Hasselt et al, AAAI 2015

Dueling Architecture: Wang et al, ICML 2016

Prioritized Replay: Schaul et al, ICLR 2016

David Silver ICML 2016 tutorial



Conclusion

- We're done with Part I: Search and Planning!
- We've seen how AI methods can solve problems in:
 - Search
 - Games
 - Markov Decision Problems
 - Reinforcement Learning
- Next up: Part II: Uncertainty and Learning!

