

---

# **Lecture 5: Computational Cognitive Modeling**

---

**Reinforcement Learning (pt. 2)**

**email address for instructors:**  
instructors-ccm-spring2018@nyucll.org

**course website:**  
<https://brendenlake.github.io/CCM-site/>

# Reinforcement Learning

## Three levels of description (*David Marr, 1982*)

### Computational

Why do things work the way they do?  
What is the goal of the computation?  
What are the unifying principles?



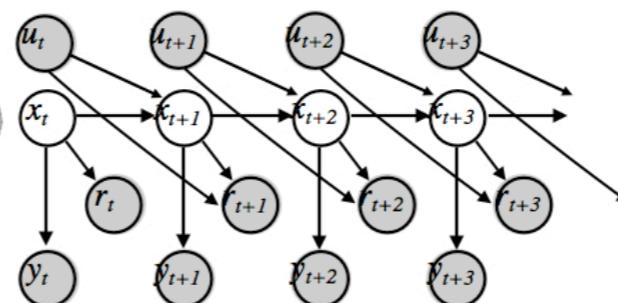
maximize:

$$R_t = r_{t+1} + r_{t+2} + \dots + r_T$$

Bellman

### Algorithmic

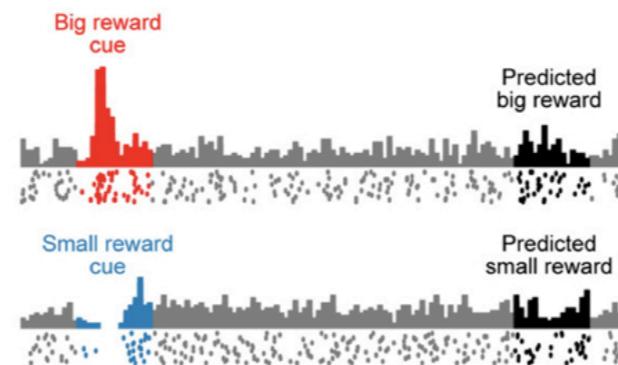
What representations can implement such computations?  
How does the choice of representations determine the algorithm?



Dynamic programming,  
TD methods, Monte  
Carlo

### Implementational

How can such a system be built in hardware?  
How can neurons carry out the computations?



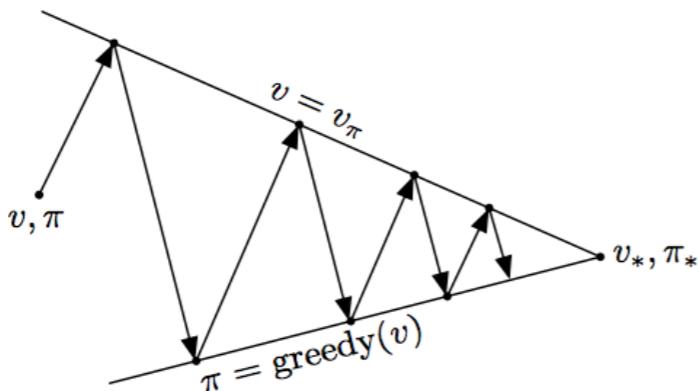
Neural firing patterns,  
prediction errors,  
system level  
neuroscience

# Overview for Today

- Temporal difference methods
- The explore-exploit dilemma
- Generalization and function approximation

# Dynamic Programming/Value iteration

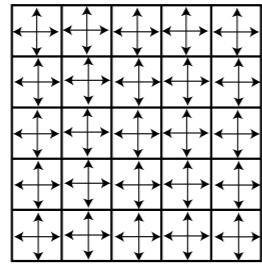
# Monte Carlo



Rewards & State Transitions

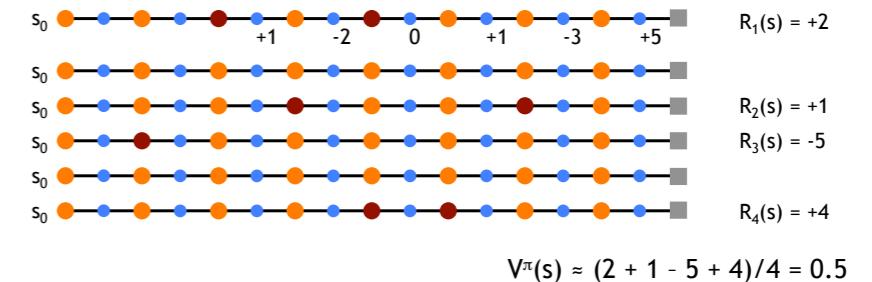
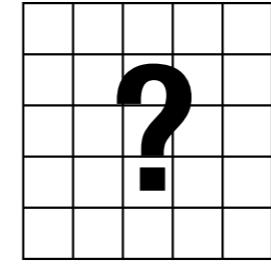
|    |     |    |    |    |    |
|----|-----|----|----|----|----|
| -1 | -1  | -1 | -1 | -1 | -1 |
|    | A   |    | B  |    |    |
| -1 |     |    | +5 |    |    |
| -1 | +10 |    |    | B' |    |
| -1 |     |    |    |    |    |
| -1 | A'  |    |    |    |    |
| -1 | -1  | -1 | -1 | -1 | -1 |

Agent's Policy ( $\pi$ )



$\gamma=0$

Value Function (V)



- Generally require “model” of environment (i.e., knowledge of state transitions, reward, and policy at a point in environment)
- Curse of dimensionality
- Proveably converges to optimal
- Solution benefits from “bootstrapping”
- Does **not** require “model”
- May not even estimate some part of environment
- Convergence more sensitive to issues like sufficient exploration
- Solution does not benefit from “bootstrapping”

# Blending the ideas....

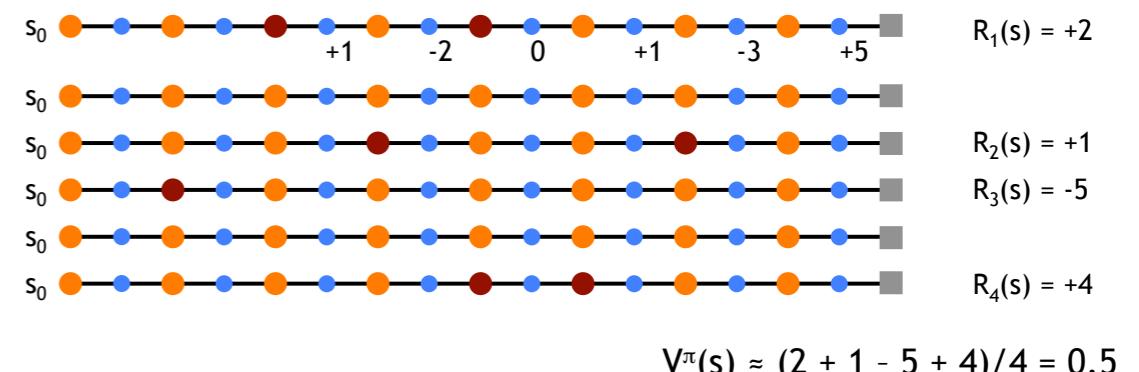
- The first-visit MC algorithm has following steps

Let  $R$  be the return following first visit to state  $s$ . Append  $R$  to list  $\text{Returns}[s]$ .  $V(s) = \text{average}(\text{Returns}[s])$

- Incremental implementation:

$$V(s) = V(s) + \frac{1}{n(s)} [R - V(s)]$$

where  $n(s)$  is number of first visits to  $s$ .

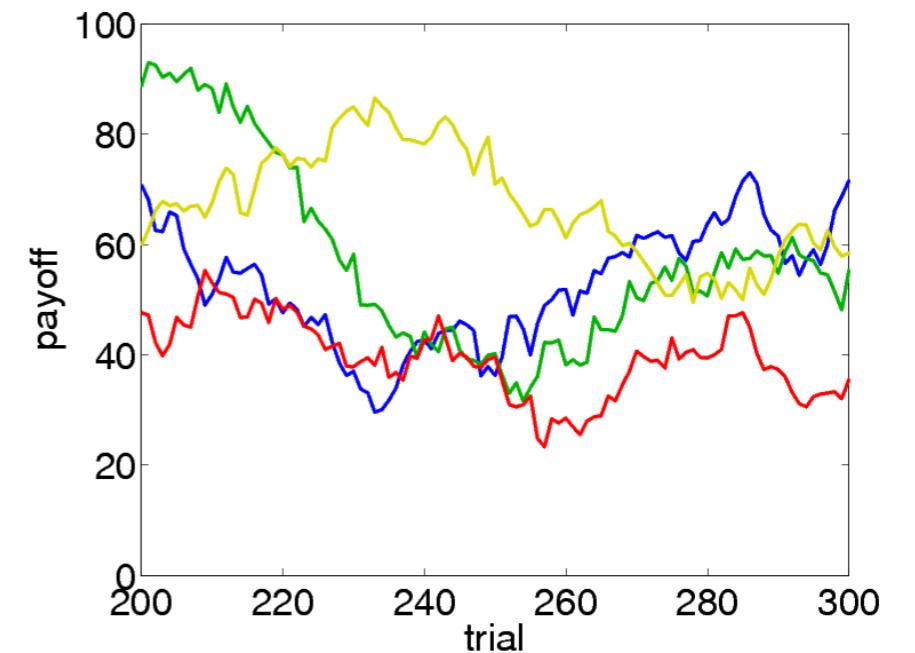


# Blending the ideas....

Now consider a constant step size Monte-carlo update:

$$V(s) = V(s) + \alpha[R - V(s)]$$

Why might this be useful?



(hint)

# Temporal difference prediction

Policy evaluation is often referred to as a prediction problem: we are trying to predict how much return we'll get from being in state  $s$  and following our policy.

Monte carlo incremental update

$$V(s) = V(s) + \alpha[R - V(s)]$$

 target: *actual* return from  $s_t$  to end of episode

Still have to wait until episode terminates...

Temporal Difference update TD(0):

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$



target: *estimate* of the return... using BOOTSTRAPPING!

# Evaluating the world when you don't know anything about it

$$\alpha = 0.9 \quad \gamma = 1 \quad \pi - \text{random}$$

Initialize

|   |   |   |
|---|---|---|
| d | e | f |
| a | b | c |



|   |   |   |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |

$$c \rightarrow f \quad V(c) \leftarrow 0 + 0.9[100 + 0 - 0] = 90$$

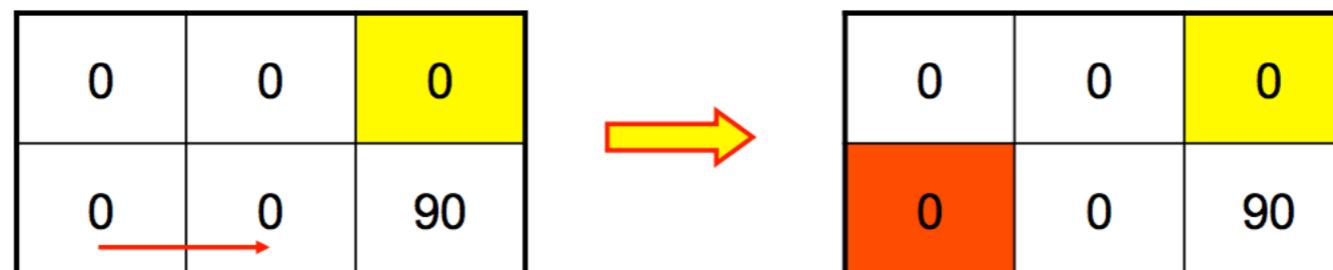
|   |   |   |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |



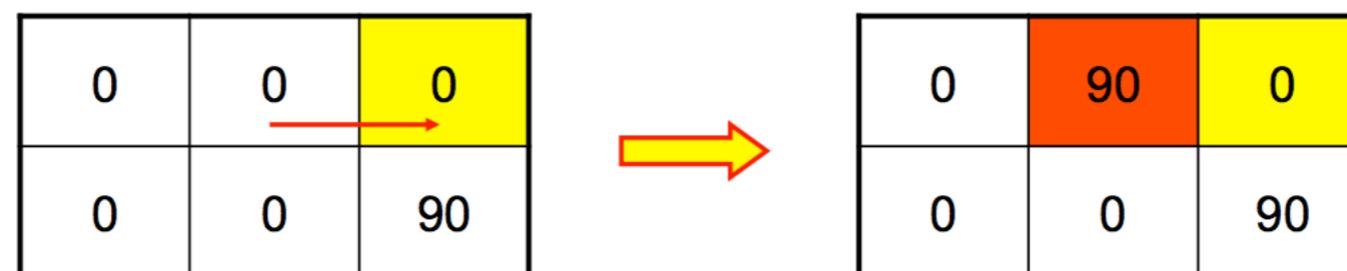
|   |   |    |
|---|---|----|
| 0 | 0 | 0  |
| 0 | 0 | 90 |

# Evaluating the world when you don't know anything about it

$$a \rightarrow b \quad V(a) \leftarrow 0 + 0.9[0 + 0 - 0] = 0$$

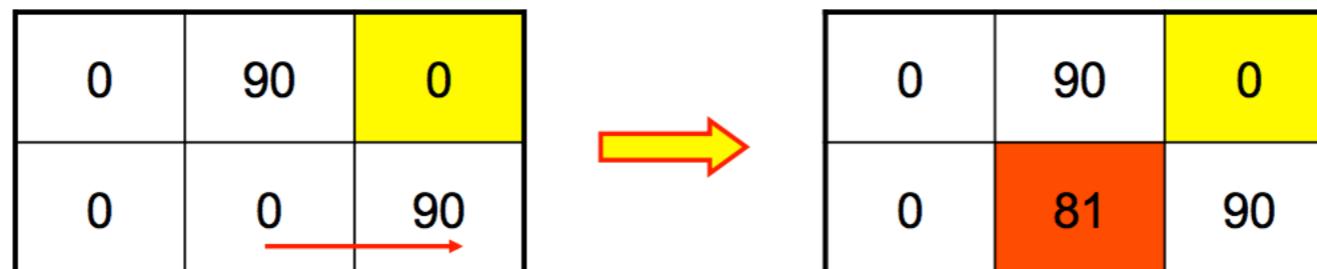


$$e \rightarrow f \quad V(e) \leftarrow 0 + 0.9[100 + 0 - 0] = 90$$

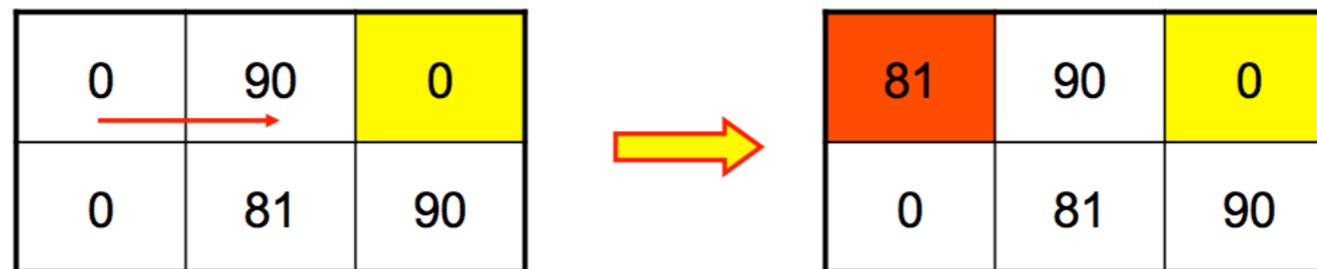


# Evaluating the world when you don't know anything about it

$$b \rightarrow c \quad V(b) \leftarrow 0 + 0.9[0 + 90 - 0] = 81$$



$$d \rightarrow e \quad V(d) \leftarrow 0 + 0.9[0 + 90 - 0] = 81$$



# Evaluating the world when you don't know anything about it

$$a \rightarrow b \quad V(a) \leftarrow 0 + 0.9[0 + 81 - 0] \approx 73$$

|    |    |    |
|----|----|----|
| 81 | 90 | 0  |
| 0  | 81 | 90 |

|    |    |    |
|----|----|----|
| 81 | 90 | 0  |
| 73 | 81 | 90 |

$$c \rightarrow f \quad V(c) \leftarrow 90 + 0.9[100 + 0 - 90] = 99$$

|    |    |    |
|----|----|----|
| 81 | 90 | 0  |
| 73 | 81 | 90 |

|    |    |    |
|----|----|----|
| 81 | 90 | 0  |
| 73 | 81 | 99 |

# Evaluating the world when you don't know anything about it

$$e \rightarrow f \quad V(e) \leftarrow 90 + 0.9[100 + 0 - 90] = 99$$

|    |    |    |
|----|----|----|
| 81 | 90 | 0  |
| 73 | 81 | 99 |

|    |    |    |
|----|----|----|
| 81 | 99 | 0  |
| 73 | 81 | 99 |

$$c \rightarrow b \quad V(c) \leftarrow 99 + 0.9[0 + 81 - 99] \approx 83$$

|    |    |    |
|----|----|----|
| 81 | 99 | 0  |
| 73 | 81 | 99 |

|    |    |    |
|----|----|----|
| 81 | 99 | 0  |
| 73 | 81 | 83 |

# Evaluating the world when you don't know anything about it

$$\gamma = 0.9$$



|    |    |    |
|----|----|----|
| 52 | 66 | 0  |
| 49 | 57 | 76 |

bellman solution!

# Temporal difference prediction

Temporal Difference update TD(0):

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

Bellman recurrence relation

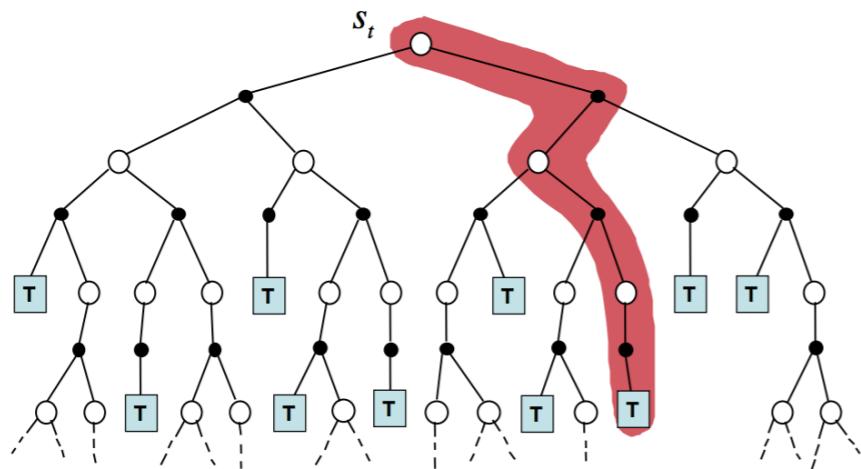
$$V^\pi(s) = E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s\}$$

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma V^\pi(s')]$$

## Simple Monte Carlo

$$V(s_t) \leftarrow V(s_t) + \alpha [R_t - V(s_t)]$$

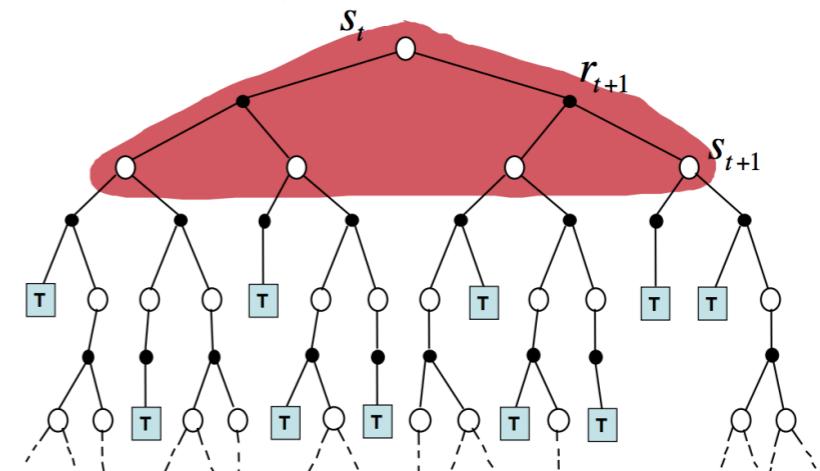
where  $R_t$  is the actual return following state  $s_t$ .



Monte Carlo uses an estimate of the actual return.

## Dynamic Programming

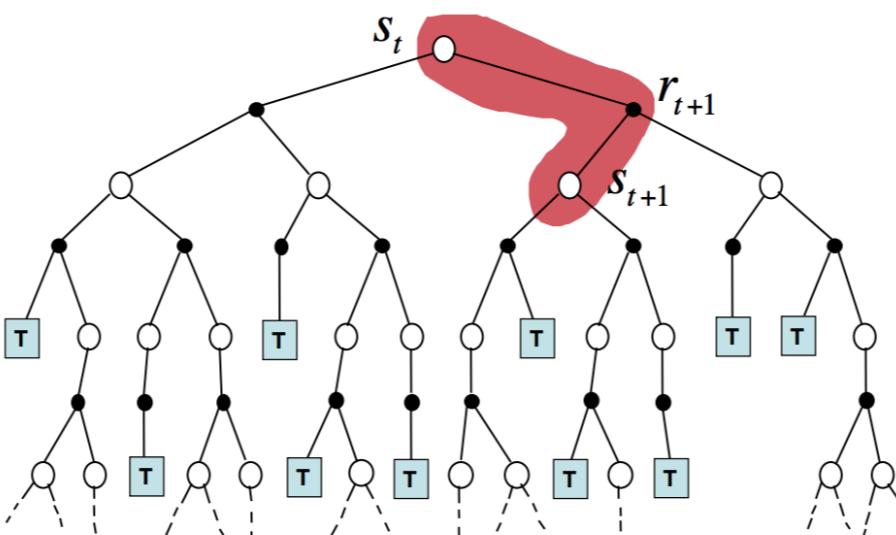
$$V(s_t) \leftarrow E_\pi \{r_{t+1} + \gamma V(s_t)\}$$



The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because  $V^\pi$  is not known and the current estimate is used instead.

## Simplest TD Method

$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$



TD samples the expected value and uses the current estimate of the value.

# Advantages of TD learning methods

- Don't need a model of the environment
- Online and incremental so can be fast (don't need to wait until end of episode as in MC)
- Update based on actual experience ( $r_{t+1}$ )
- Converges to the true values if you lower step size/learning rate as learning continues
- TD bootstraps: it updates estimate based on other estimates (like DP/value iteration).
- TD samples: updates are based on a single run/path through the state space (like MC)

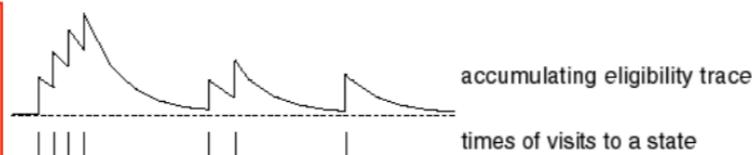
# TD(0) is still kind of slow: Eligibility traces

- The benefits of bootstrapping only extend between adjacent states ( $s$  to  $s'$ ). As a result you have to cross that particular state transition many times for the value to “propagate” backwards
- New variable called *eligibility trace*. The eligibility trace for state at time  $t$  is denoted

$$e_t(s) \in \mathbb{R}^+$$

On each step, decay all traces by  $\gamma\lambda$  and increment the trace for the current state by

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$



$\gamma$  discount rate

$\lambda$  trace-decay parameter

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$$

$$\Delta V_t(s) = \alpha \delta_t e_t(s)$$

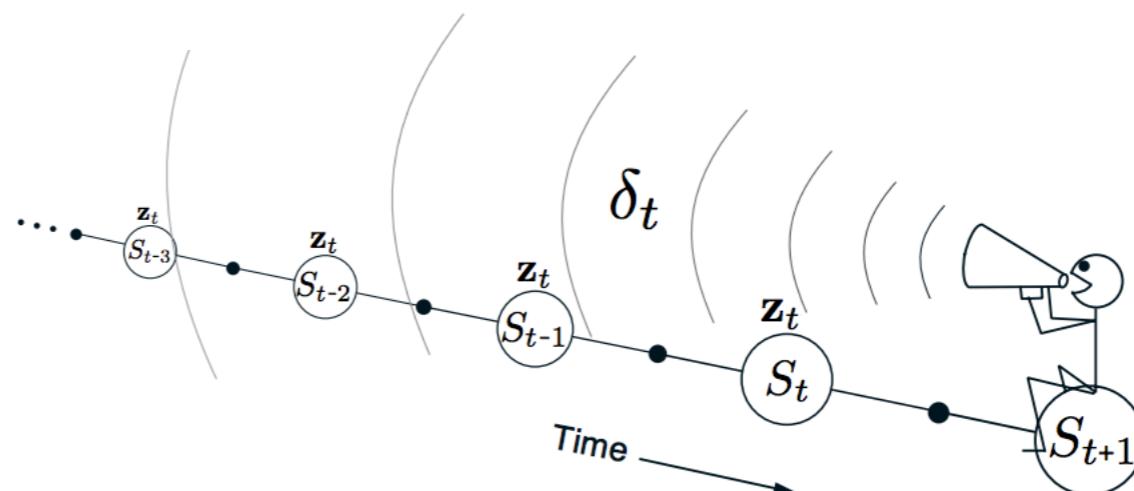
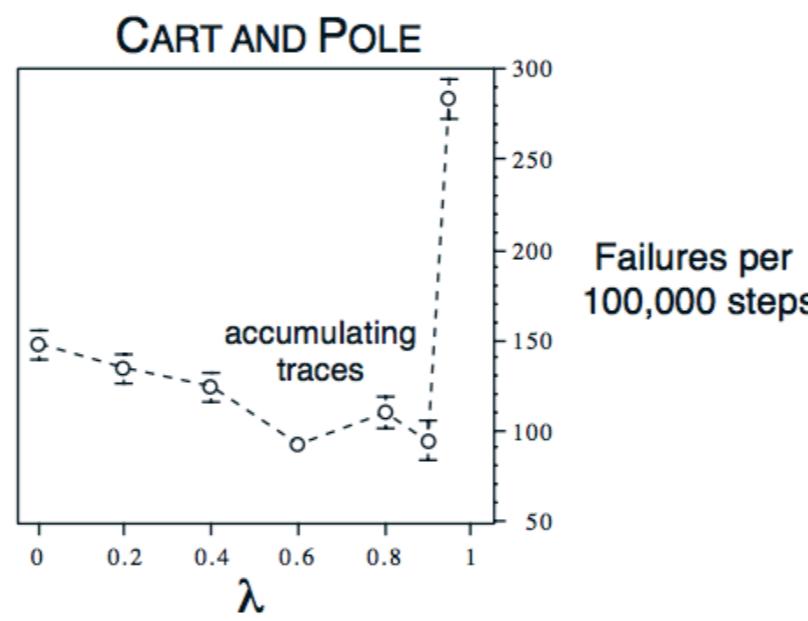
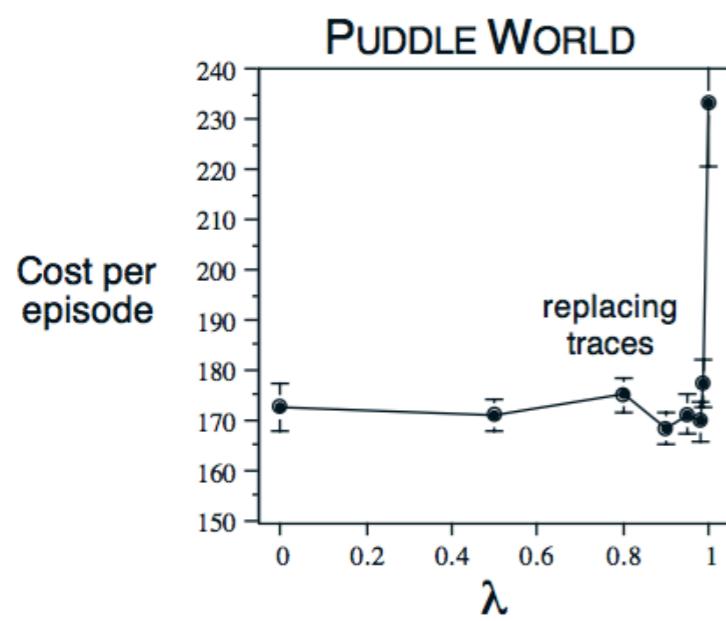
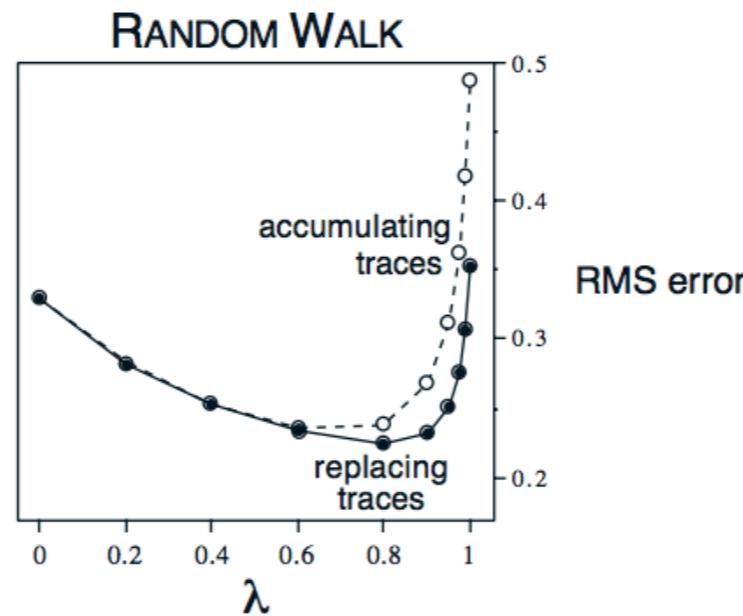
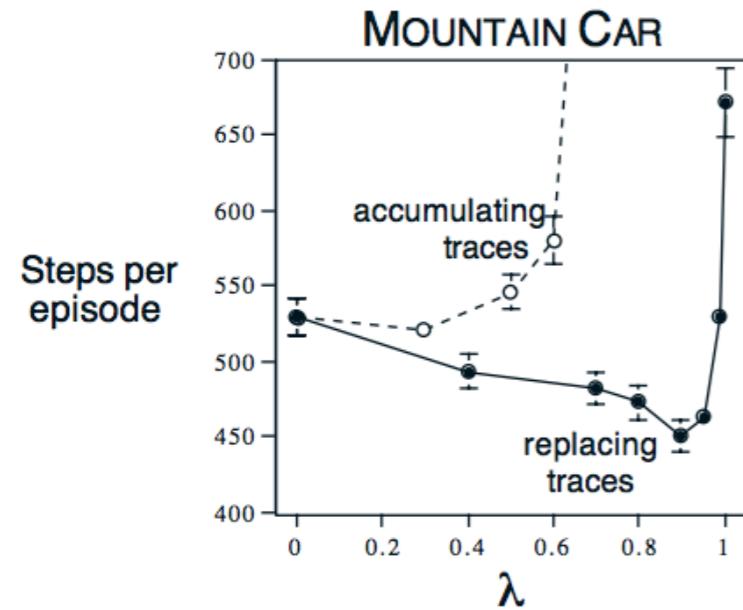


Figure 12.5: The backward or mechanistic view. Each update depends on the current TD error combined with the current eligibility traces of past events.

# TD(0) is still kind of slow: Eligibility traces



intermediate values  
empirically work  
best!

# Learning for control: Learning Q-values

- Learning the value of different states can be a little obtuse because what you really want to do is learn how to act!
- Instead can make sense to learn  $Q^\pi(s, a)$

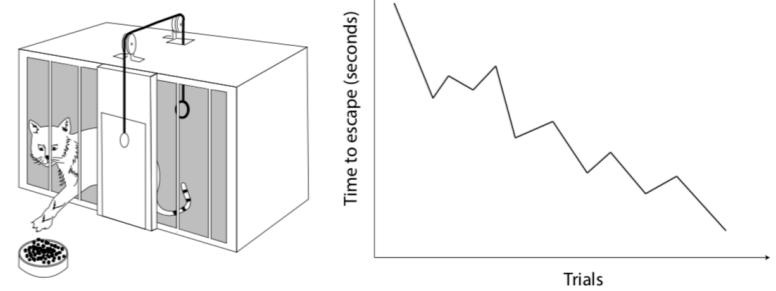


Figure 1: Left: An illustration of Thorndike's puzzle box experiments. Right: The time recorded to escape the box is reduced over repeated trials as the cat becomes more efficient at selecting the actions which lead to escape.

## SARSA update rule:

$$\Delta Q_t(s_t, a_t) = \alpha[r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$$

- Choose a policy and estimate the Q-values using SARSA rule. Change policy toward greediness with respect to Q values.
- Converges with probability 1 to optimal policy and Q-value if you visit all state-action pairs infinitely many times and the policy converges to be a greedy policy.
- Easy to know what to do! Just choose the action with highest Q value!

# Learning for control: Learning Q-values

SARSA update rule:

$$\Delta Q_t(s_t, a_t) = \alpha[r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$$

- Initialise  $Q(s, a)$
- Repeat      many times
  - Pick  $s, a$
  - Repeat      each step to goal
    - \* Do  $a$ , observe  $r, s'$
    - \* Choose  $a'$  based on  $Q(s', a')$        $\epsilon$ -greedy
    - \*  $Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    - \*  $s = s', a = a'$
  - Until  $s$  terminal (where  $Q(s', a') = 0$ )

sarsa is known as an  
on-policy learning rule...

Use with policy iteration, i.e. change policy each time to be greedy wrt current estimate of  $Q$

# Learning for control: Learning Q-values

Q-learning update rule:

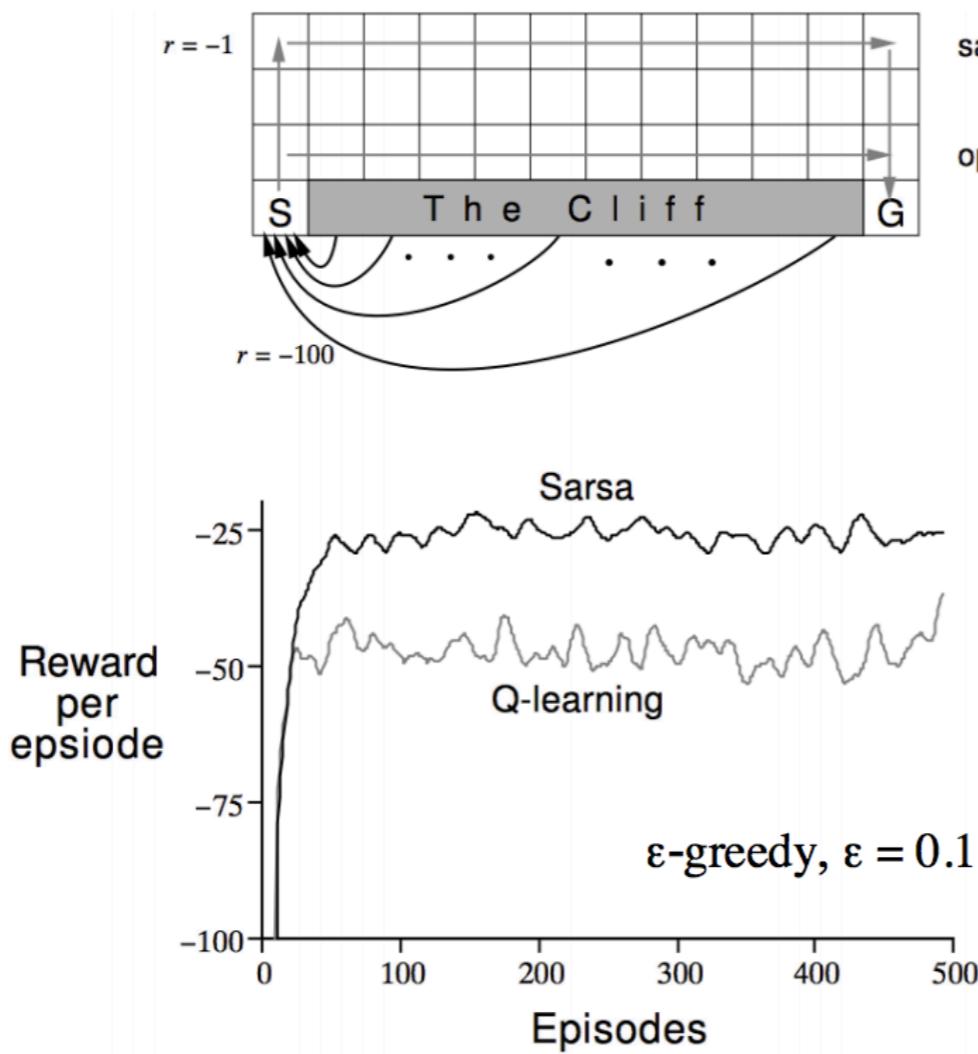
$$\Delta Q_t(s_t, a_t) = \alpha[r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)]$$

- Initialise  $Q(s, a)$
- Repeat many times
  - Pick  $s$  start state
  - Repeat each step to goal
    - \* Choose  $a$  based on  $Q(s, a)$   $\epsilon$ -greedy
    - \* Do  $a$ , observe  $r, s'$
    - \*  $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    - \*  $s = s'$
  - Until  $s$  terminal

Q-learning is known as an off-policy learning rule...

always update Q value with maximally best action in next state, even if you won't necessarily take that step yourself.

# Q-learning versus SARSA (Cliffwalking)

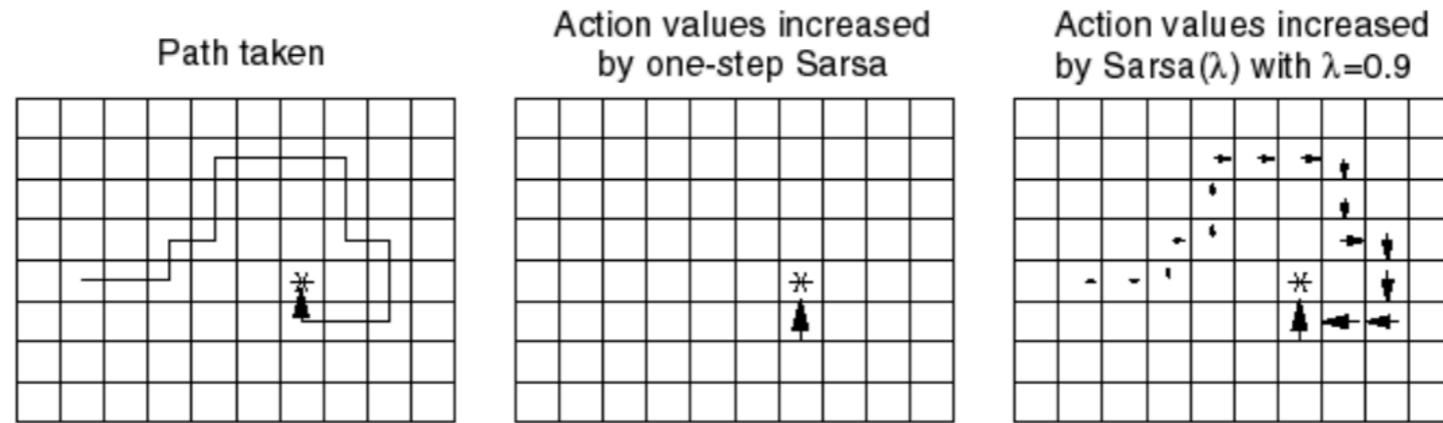


safe path  
optimal path  
Reward is on all transitions -1 except those into the the region marked "The Cliff."

**Q-learning** learns quickly **values for the optimal policy**, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the  $\epsilon$ -greedy action selection.  
**Sarsa** takes the action selection into account and learns the longer but safer path through the upper part of the grid.

If  $\epsilon$  were gradually reduced, then both methods would asymptotically converge to the optimal policy.

# SARSA(lambda)



- With one trial, the agent has much more information about how to get to the goal
  - not necessarily the *best* way
- Can considerably accelerate learning

# The Explore-Exploit Dilemma

TD methods require a bit of randomness in order to properly search the state space (we call this search process **exploration**).

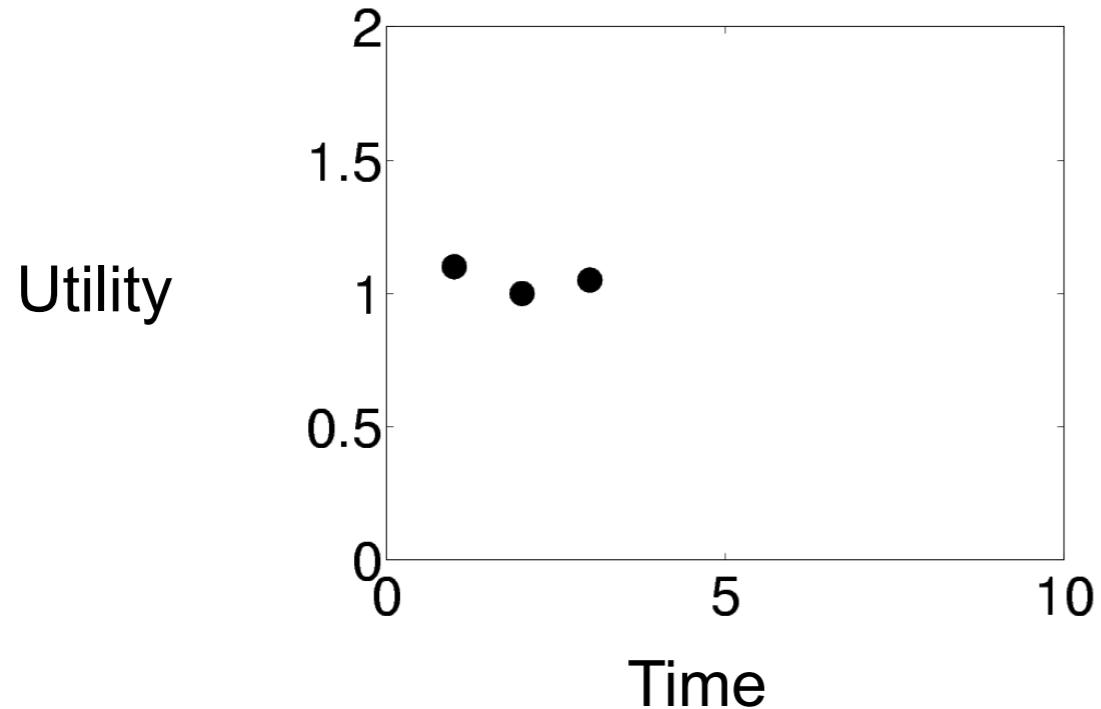
Reward maximization requires choosing what seems like the best action (**exploitation**). Effective learning in unknown environment requires proper balance of these tensions.

Classic dilemma in learned decision making

For unfamiliar outcomes, how to trade off learning about their quality/value against exploiting knowledge already gained.

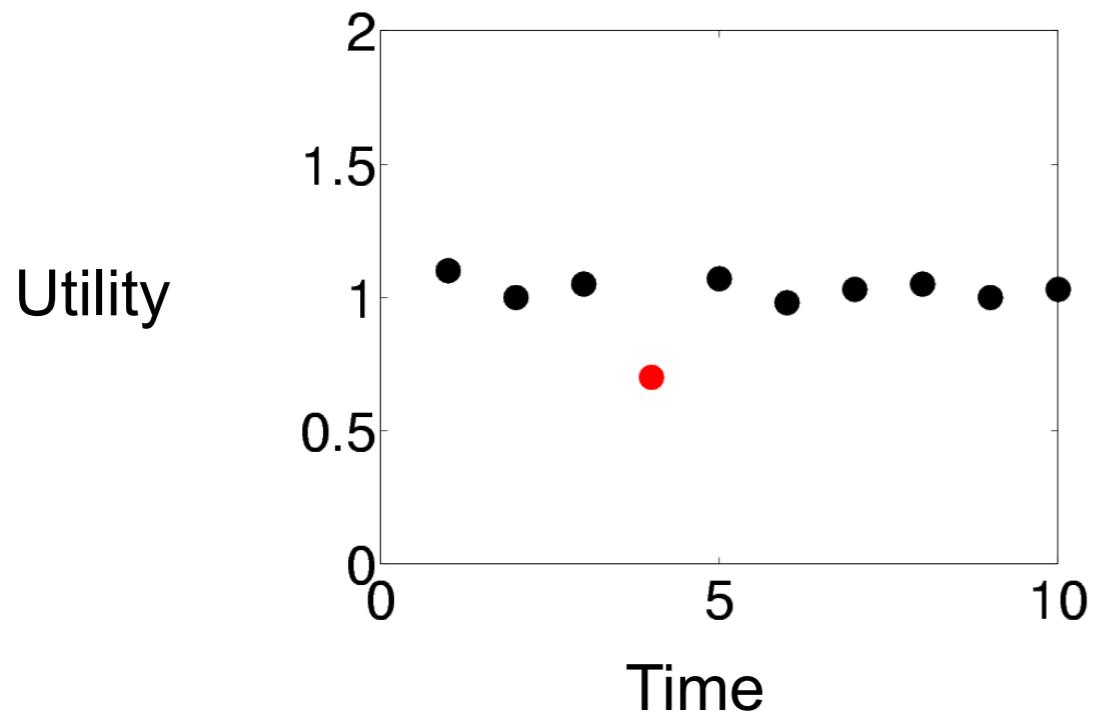


# Exploration vs. exploitation



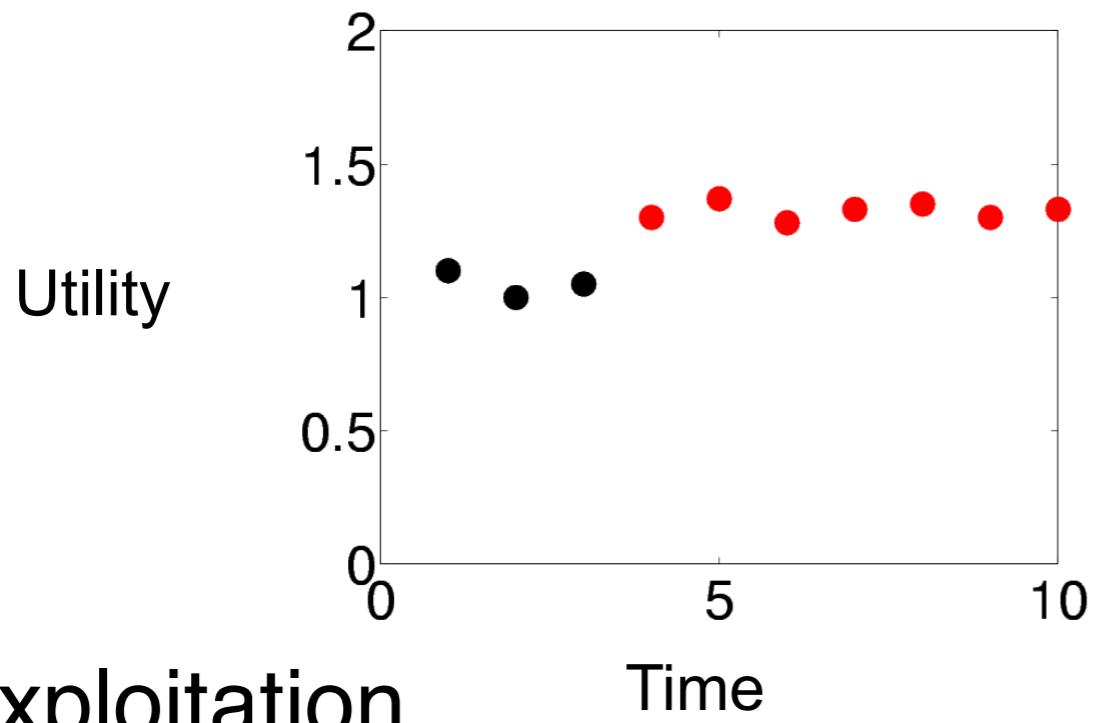
- Exploitation
  - Choose action expected to be **best**
  - May never discover something better

# Exploration vs. exploitation

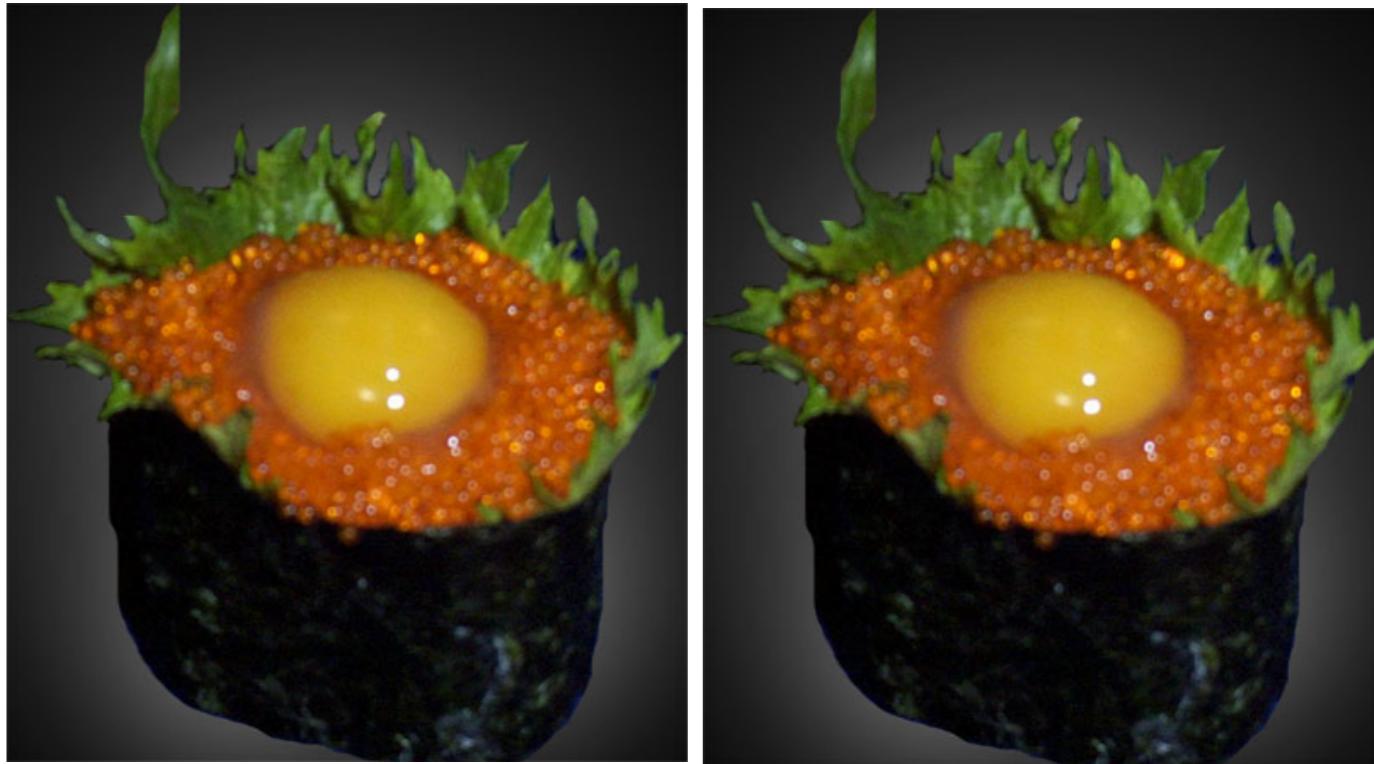


- Exploitation
  - Choose action expected to be **best**
  - May never discover something better
- Exploration:
  - Choose action expected to be **worse**

# Exploration vs. exploitation



- Exploitation
  - Choose action expected to be best
  - May never discover something better
- Exploration:
  - Choose action expected to be worse
  - Balanced by the long-term gain if it turns out better



# the N-armed bandit



another name for a popular psychology/neuroscience task:

- repeated choice between lotteries...
- ...whose properties are learned experientially
- (assume each bandit is just a weighted coin: no weird time-based lotteries)

overall approach:

1. learn Q-values for options
2. choose **the best ??**

**1. Greedy methods (e.g., epsilon greedy)**

**2. Softmax**

**3. Optimal exploration**

## Action Selection

**Greedy:** select the action  $a^*$  for which  $Q$  is highest:

$$Q_t(a^*) = \max_a Q_t(a)$$

So  $a^* = \arg \max_a Q_t(a)$  – and \* means “best”

**Example:** 10-armed bandit

Snapshot at time  $t$  for actions 1 to 10

$$Q_t(a) \rightarrow \boxed{0 \quad 0.3 \quad 0.1 \quad 0.1 \quad 0.4 \quad 0.05 \quad 0 \quad 0 \quad 0.05 \quad 0}$$

$$Q_t(a^*) = 0.4 \text{ and } a^* = ?$$

Maximises reward

**$\epsilon$ -greedy:** Select *random* action  $\epsilon$  of the time, else select greedy action

Sample all actions infinitely many times

So as  $k_a \rightarrow \infty$ ,  $Q$ s converge to  $Q^*$

Can reduce  $\epsilon$  over time

# Softmax Action Selection

$\epsilon$ -greedy: even if worst action is very bad, it will still be chosen with same probability as second-best – we may not want this. So:

Vary selection probability as a function of estimated goodness

Choose  $a$  at time  $t$  from among the  $n$  actions with probability

$$\frac{\exp(Q_t(a)/\tau)}{\sum_{b=1}^n \exp(Q_t(b)/\tau)}$$

Gibbs/Boltzmann distribution,  $\tau$  is temperature (from physics)

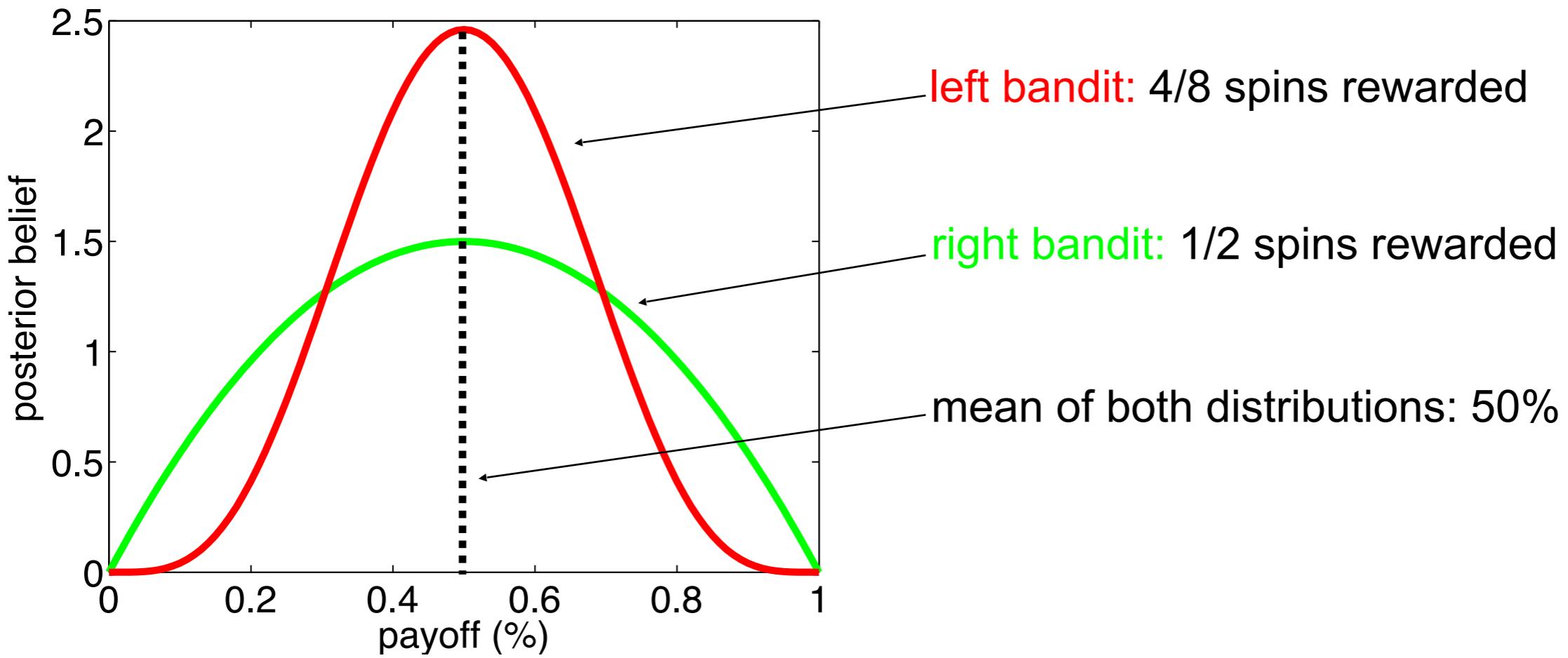
Effect of  $|\tau|$

As  $\tau \rightarrow \infty$ , probability  $\rightarrow 1/n$

As  $\tau \rightarrow 0$ , probability  $\rightarrow$  greedy

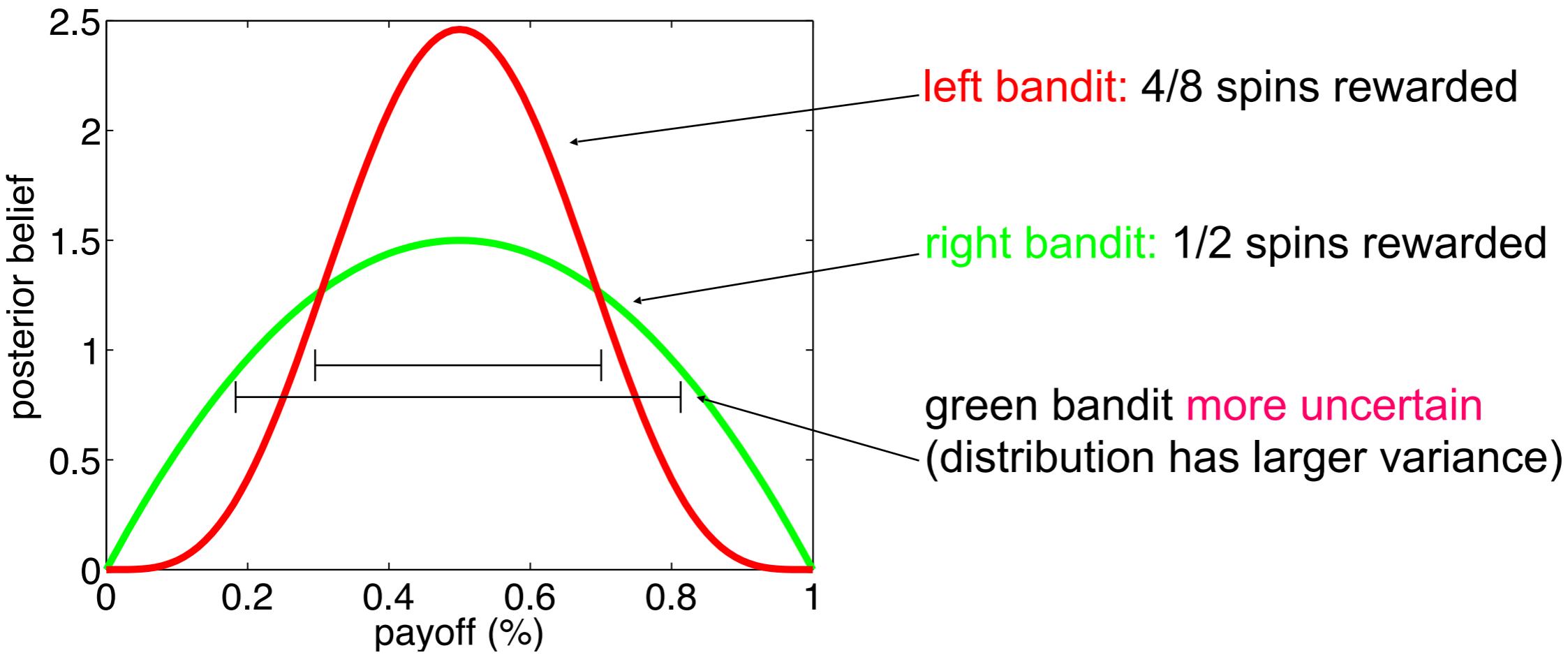
# wwBd?

remember our Bayesian analysis of learning?  
assign belief according to posterior probability  
of different chances of heads



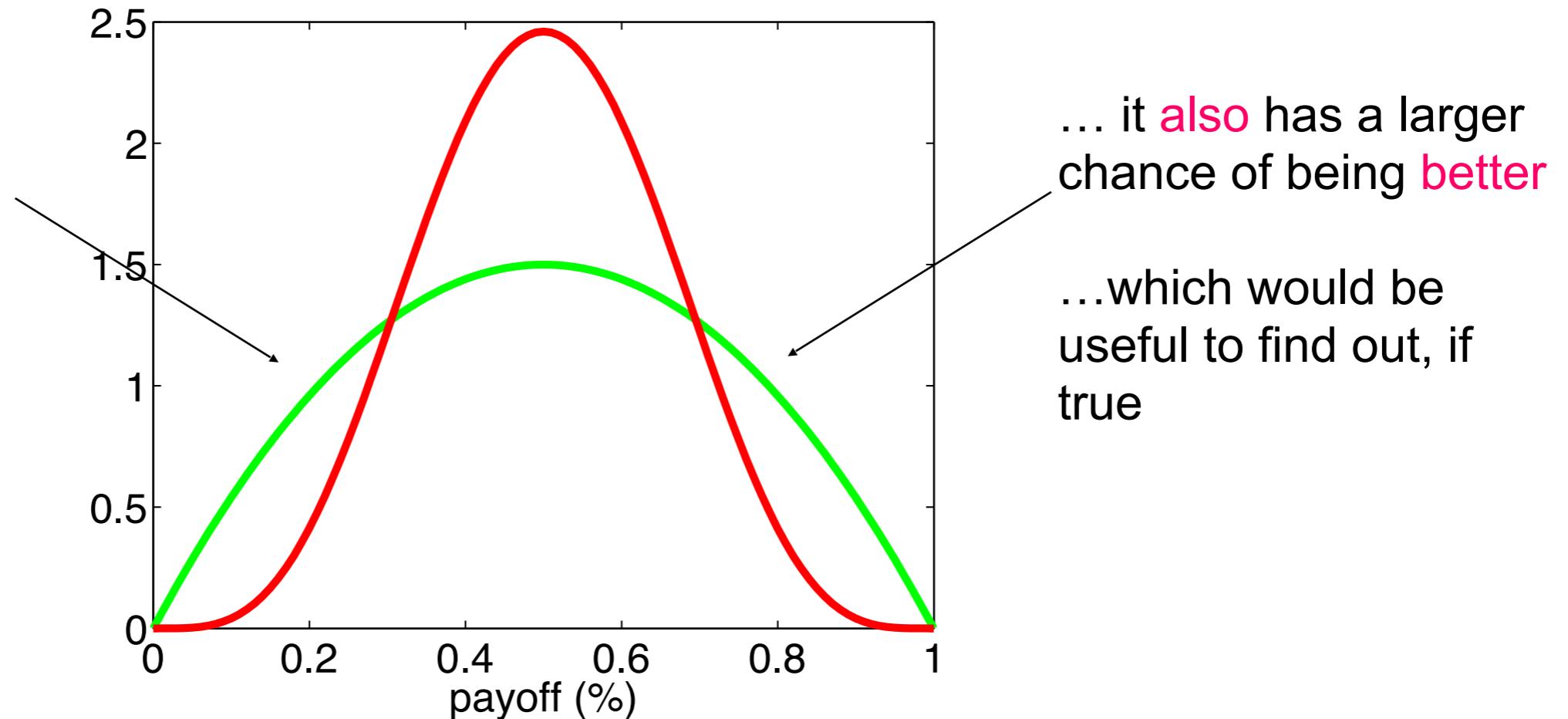
# wwBd?

remember our Bayesian analysis of learning?  
assign belief according to posterior probability  
of different chances of heads



# Gittins index

although green bandit has a larger chance of being worse...



“Gittins index”:

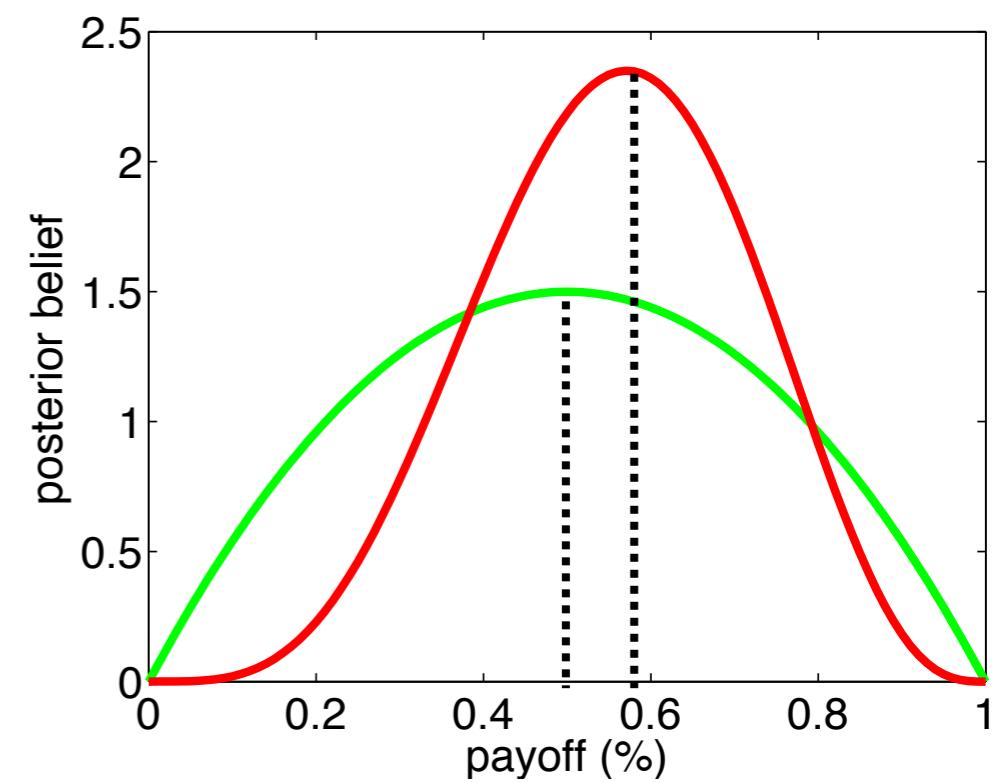
- choose on the basis of expected payoff (50%) plus “uncertainty bonus”
  - quantifies “value of information”: chance of finding something better & improving my future prospects
  - (very difficult to work out exactly, he solves for simple problems)
- note that Rescorla/Wagner model doesn't track uncertainty, only mean

# horizon



suppose I have so far been rewarded:

- 4 out of 7 spins on the **left bandit** (57%)
- 1 out of 2 spins on the **right bandit** (50%)



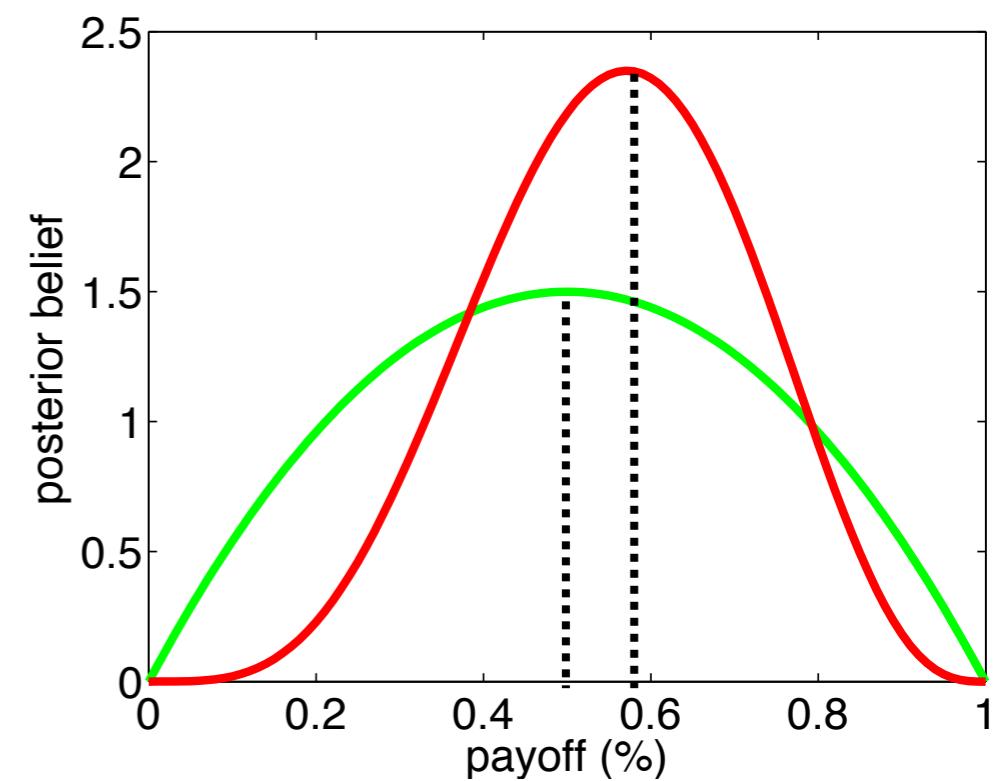
# horizon



suppose I have so far been rewarded:

- 4 out of 7 spins on the **left bandit** (57%)
- 1 out of 2 spins on the **right bandit** (50%)

... and I am allowed **only one more spin**  
now which should I choose?



# horizon

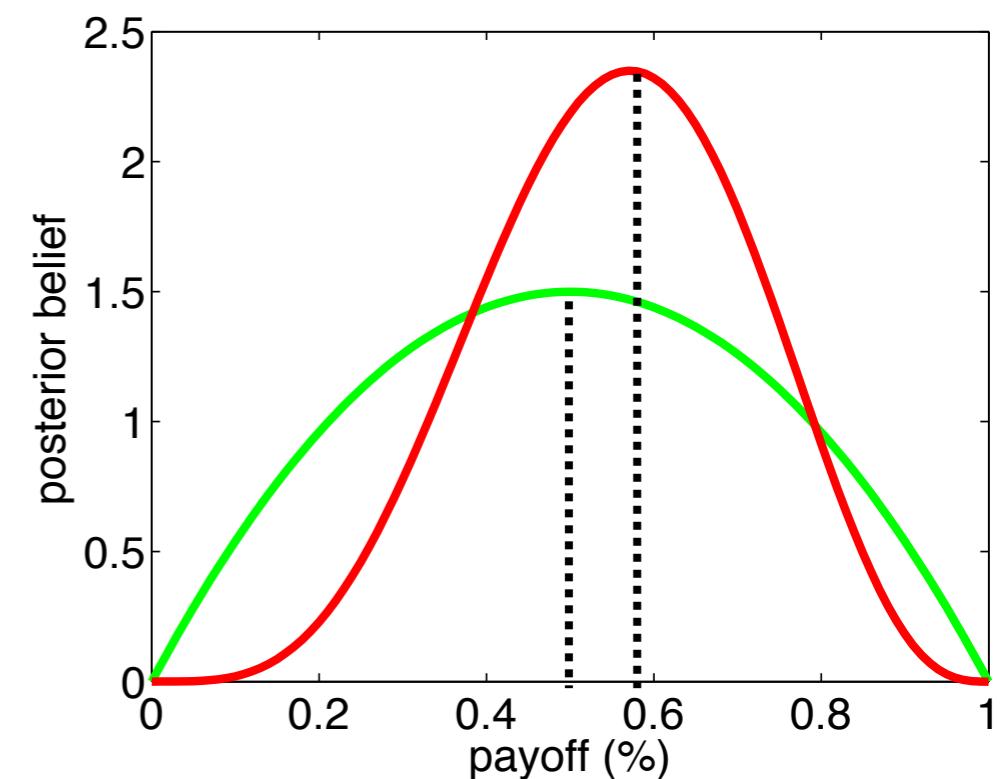


suppose I have so far been rewarded:

- 4 out of 7 spins on the **left bandit** (57%)
- 1 out of 2 spins on the **right bandit** (50%)

... and I am allowed **only one more spin**  
now which should I choose?

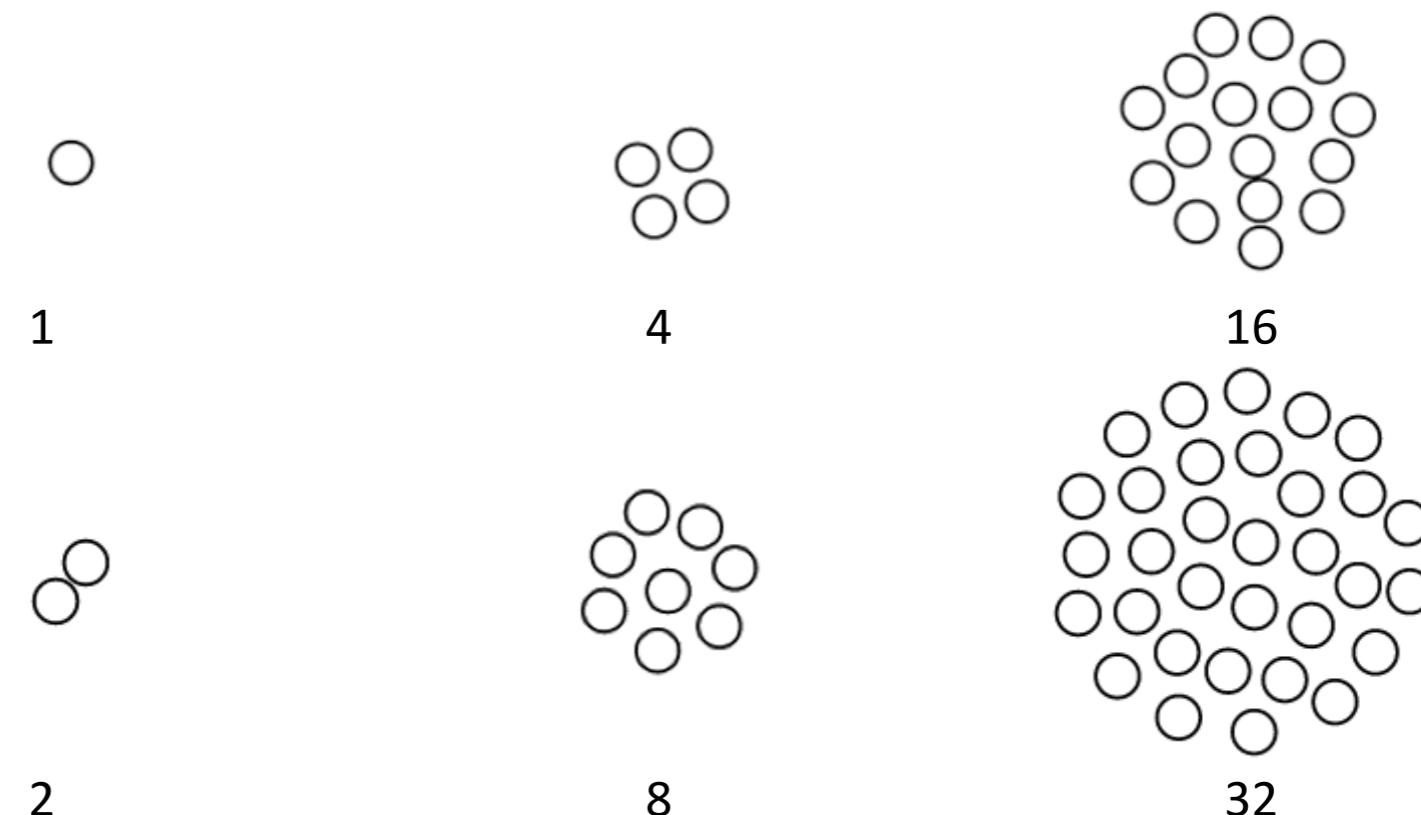
→ value of exploration depends on **temporal horizon**





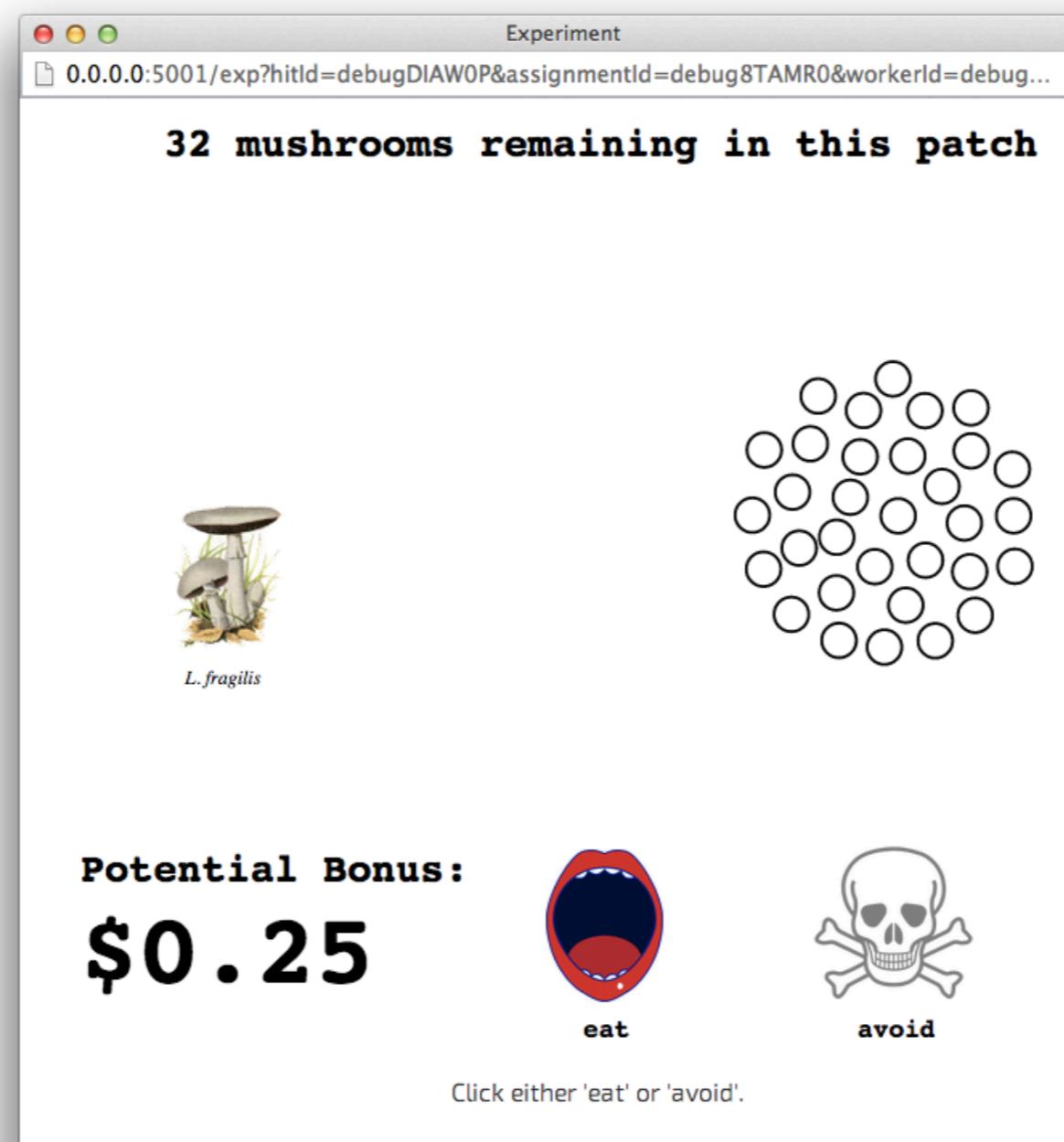
## experiment 1

are people more likely to approach (i.e., explore) a uncertain prospect when they expect to encounter it a greater number of time in the future?

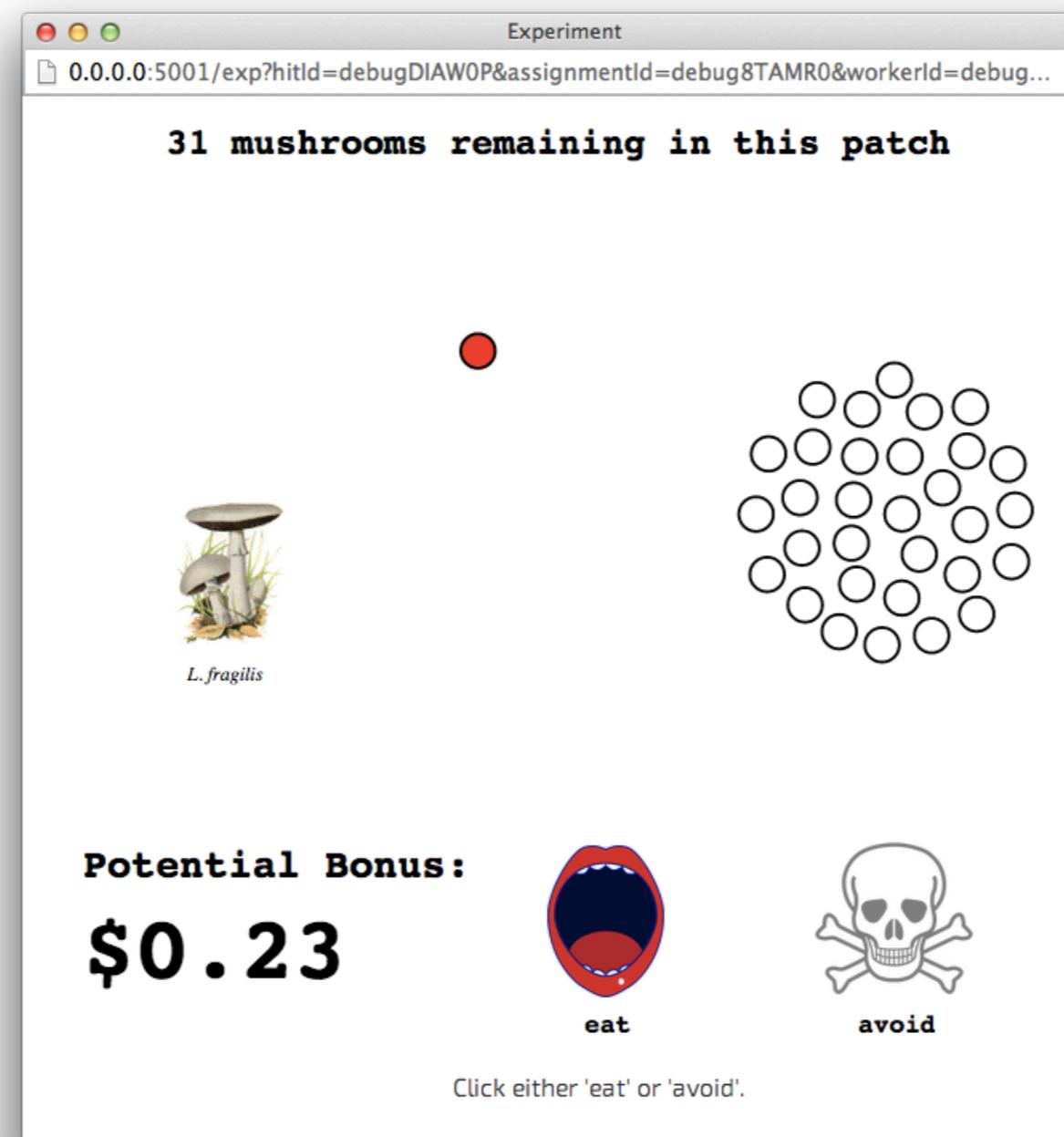


N = 143

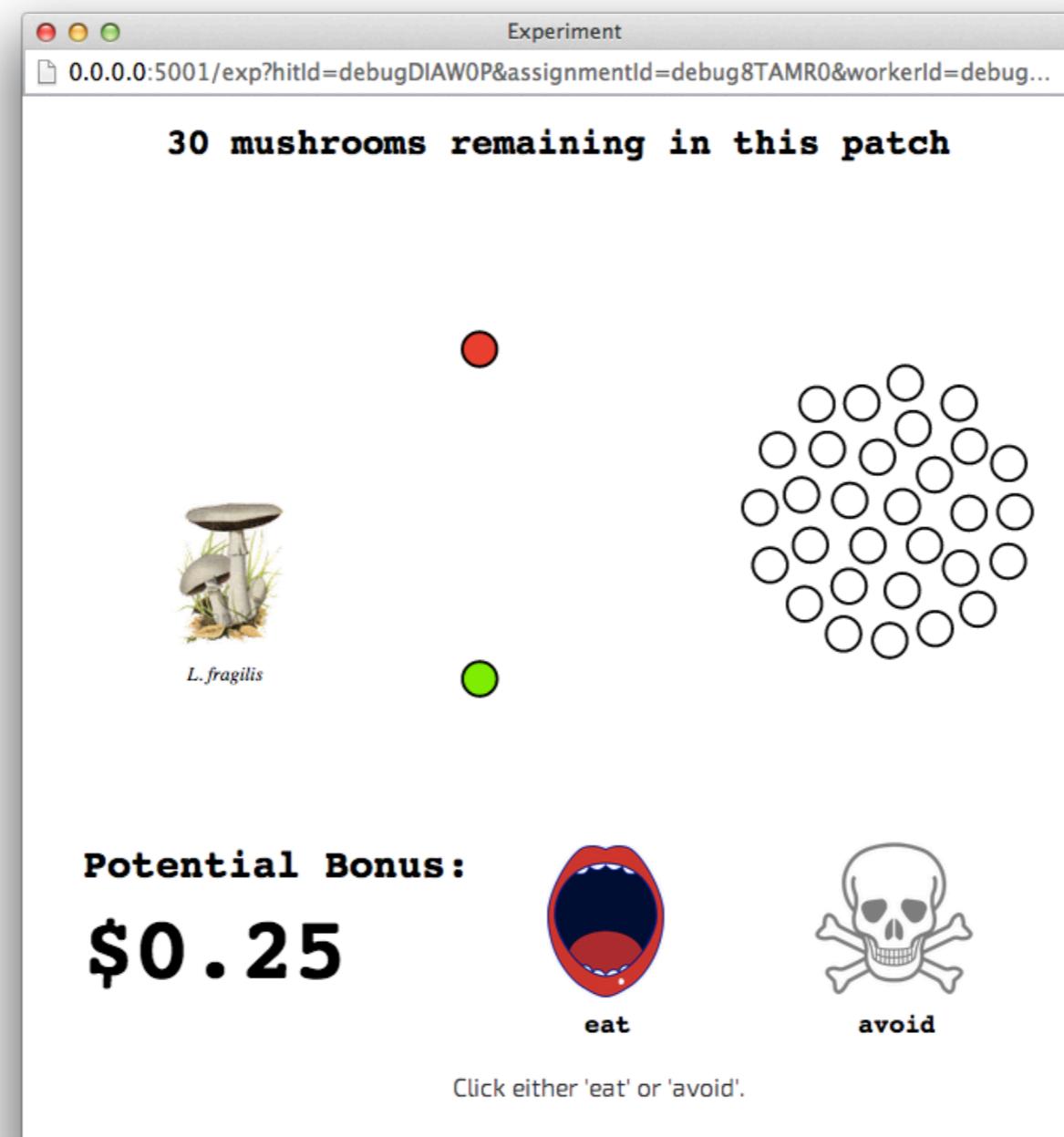
# experiment 1



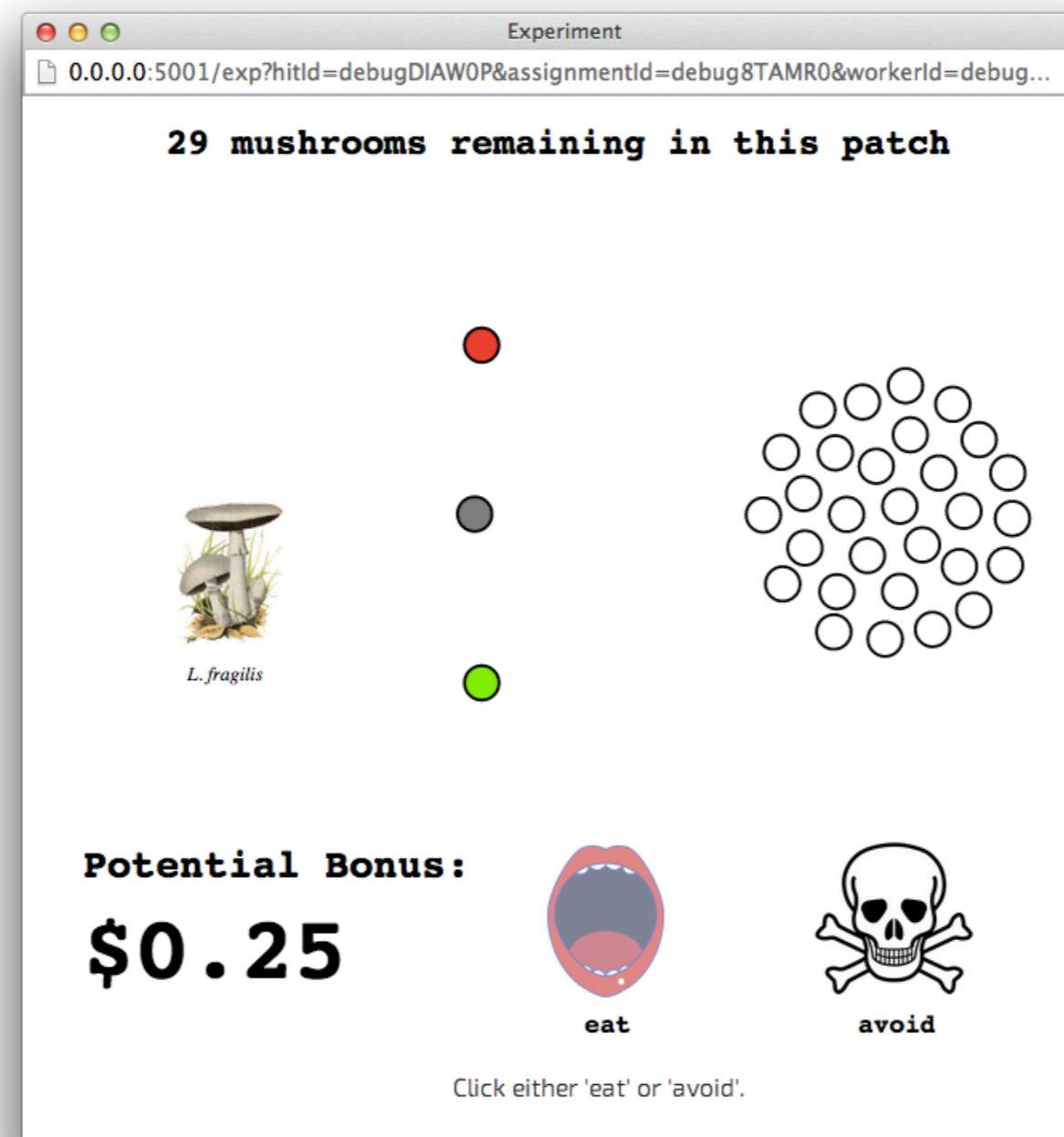
# experiment 1



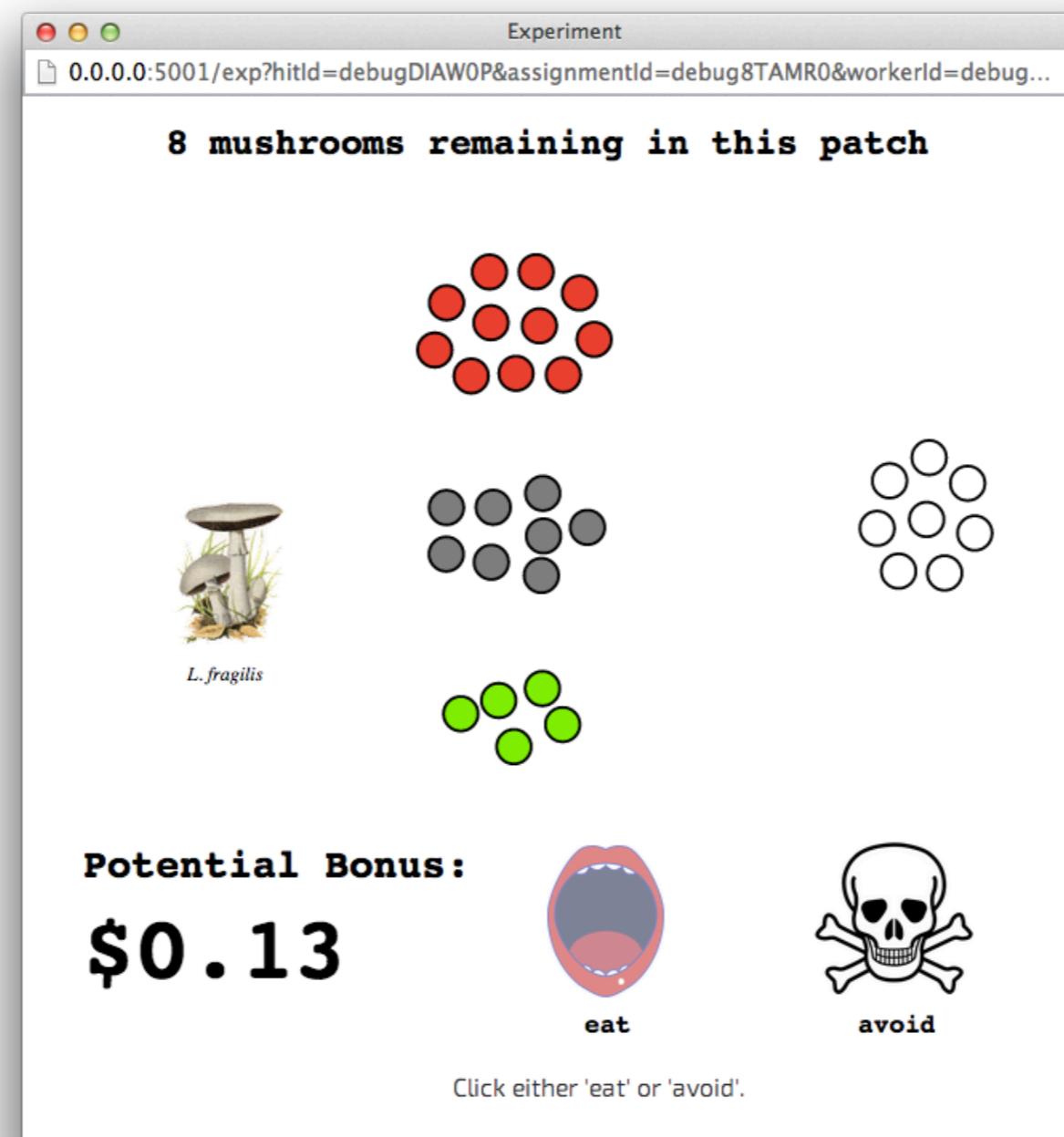
# experiment 1



# experiment 1

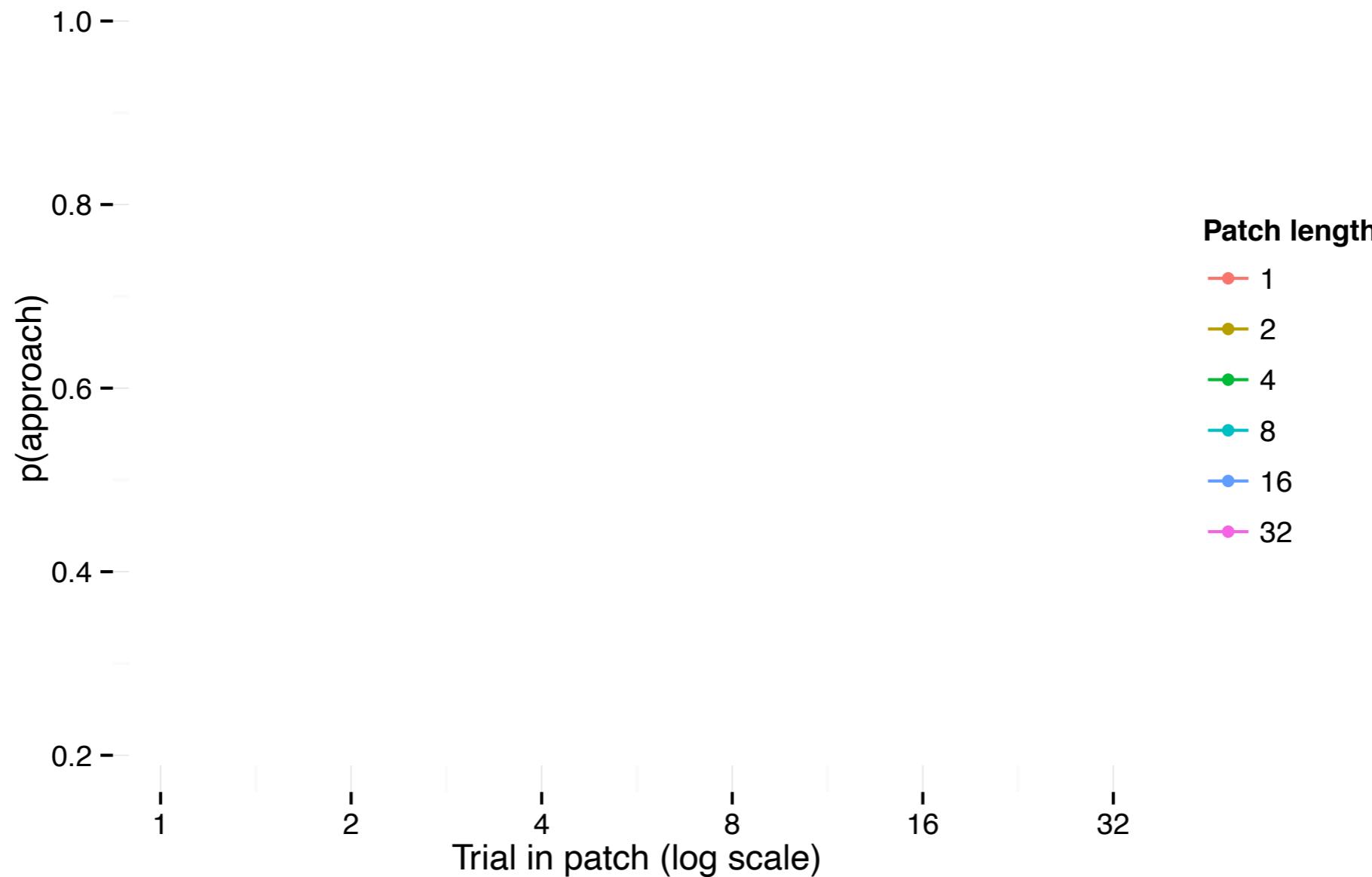


# experiment 1

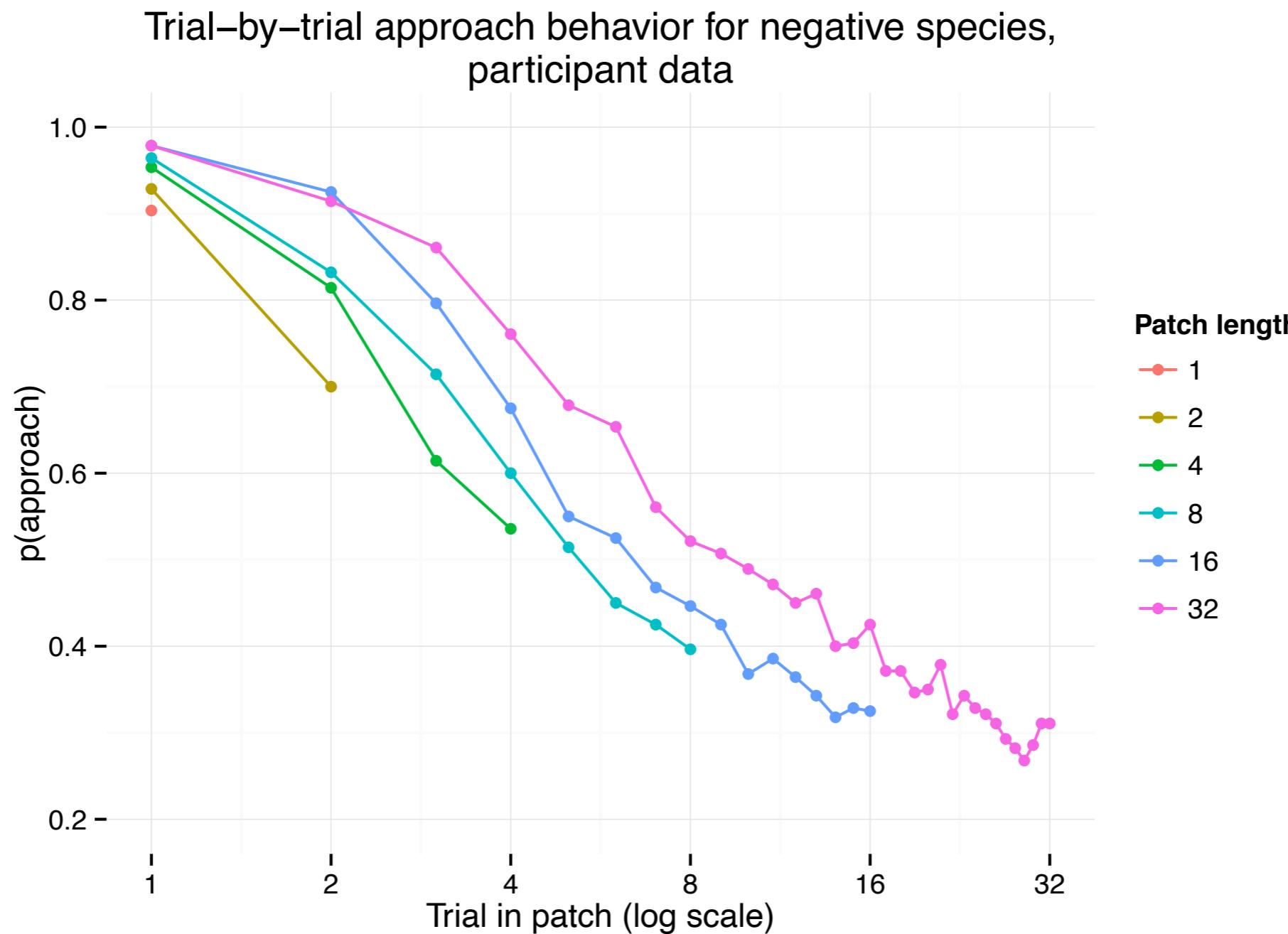


# experiment 1 - results

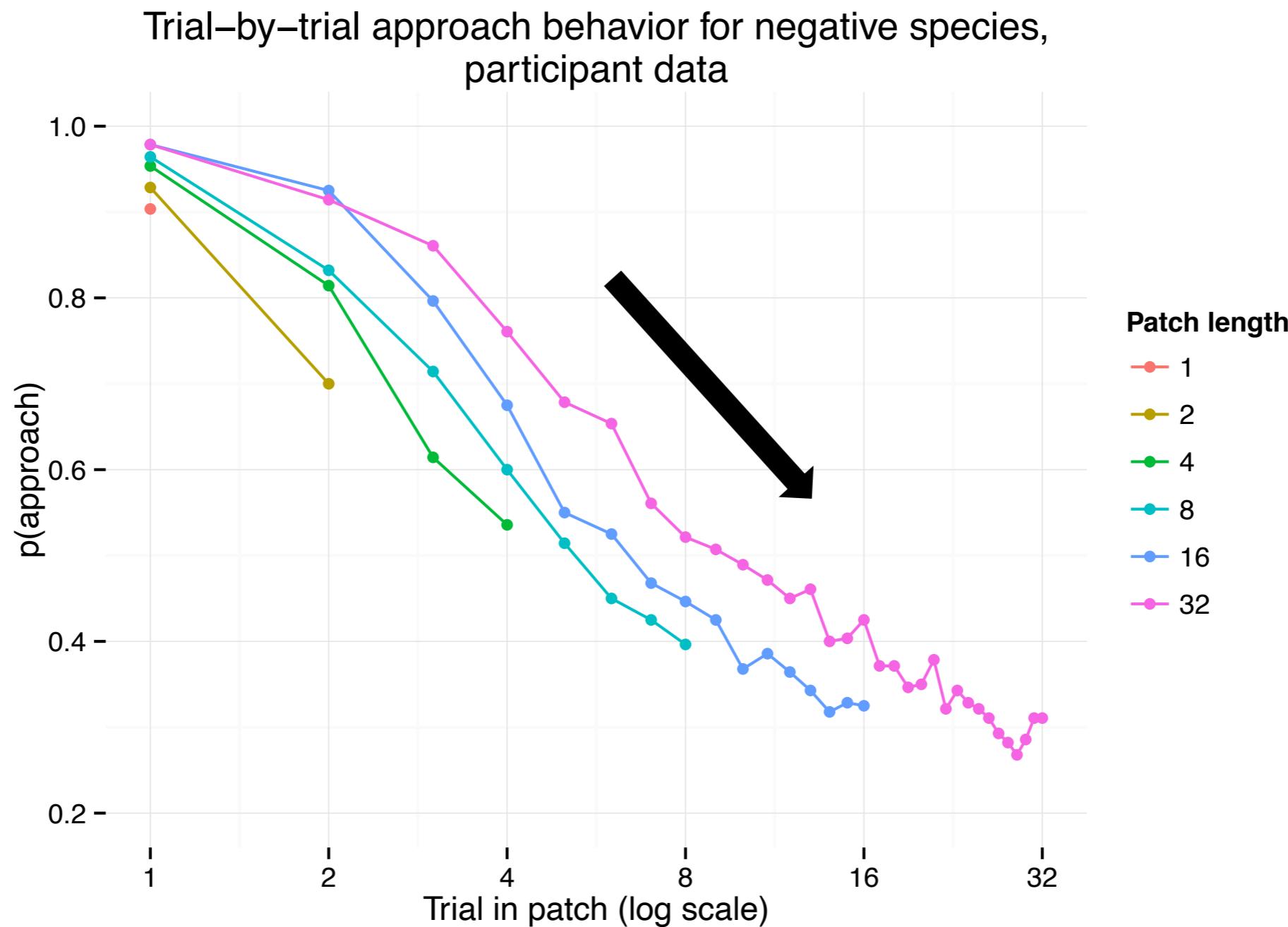
Trial-by-trial approach behavior for negative species,  
participant data



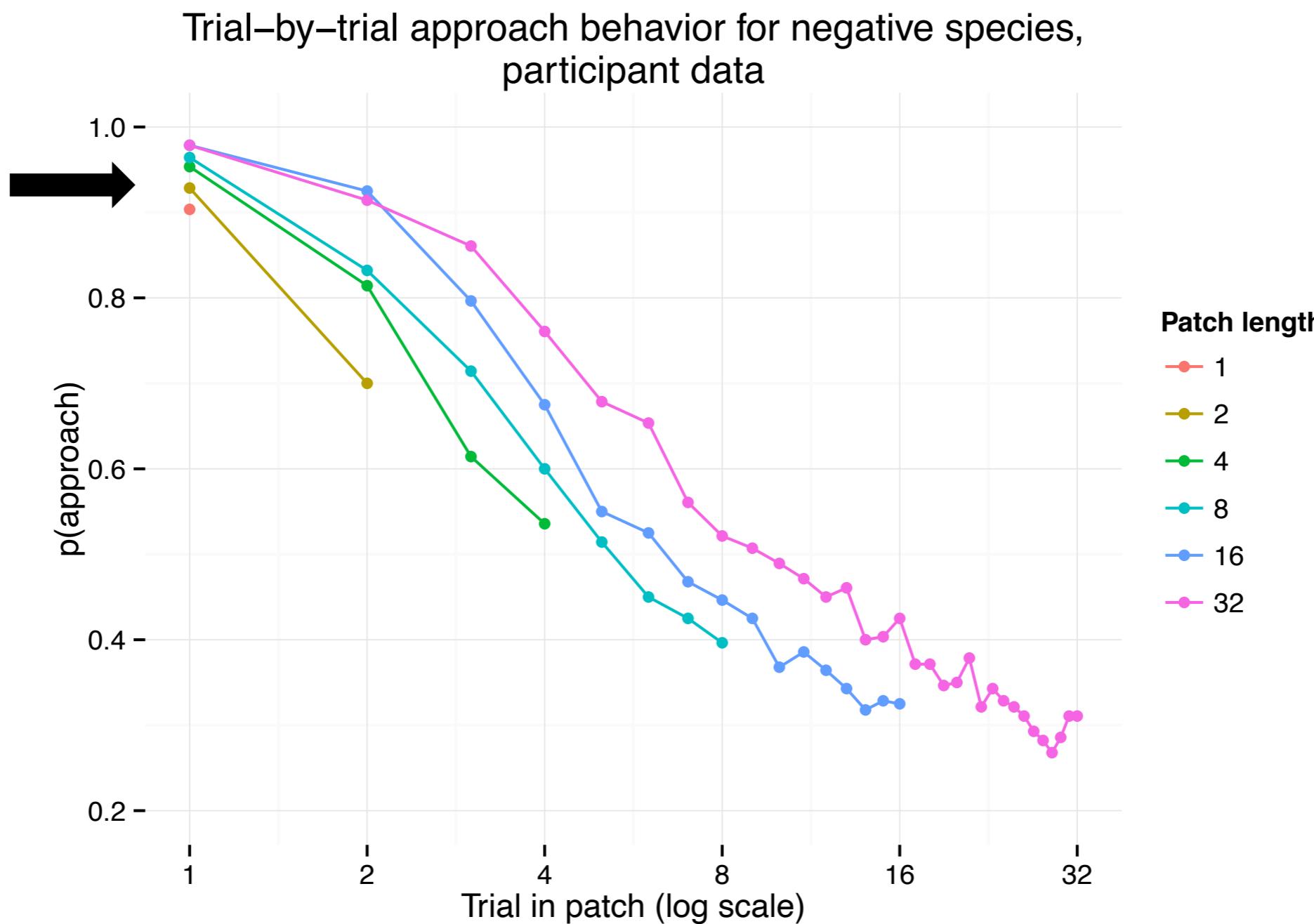
# experiment 1 - results



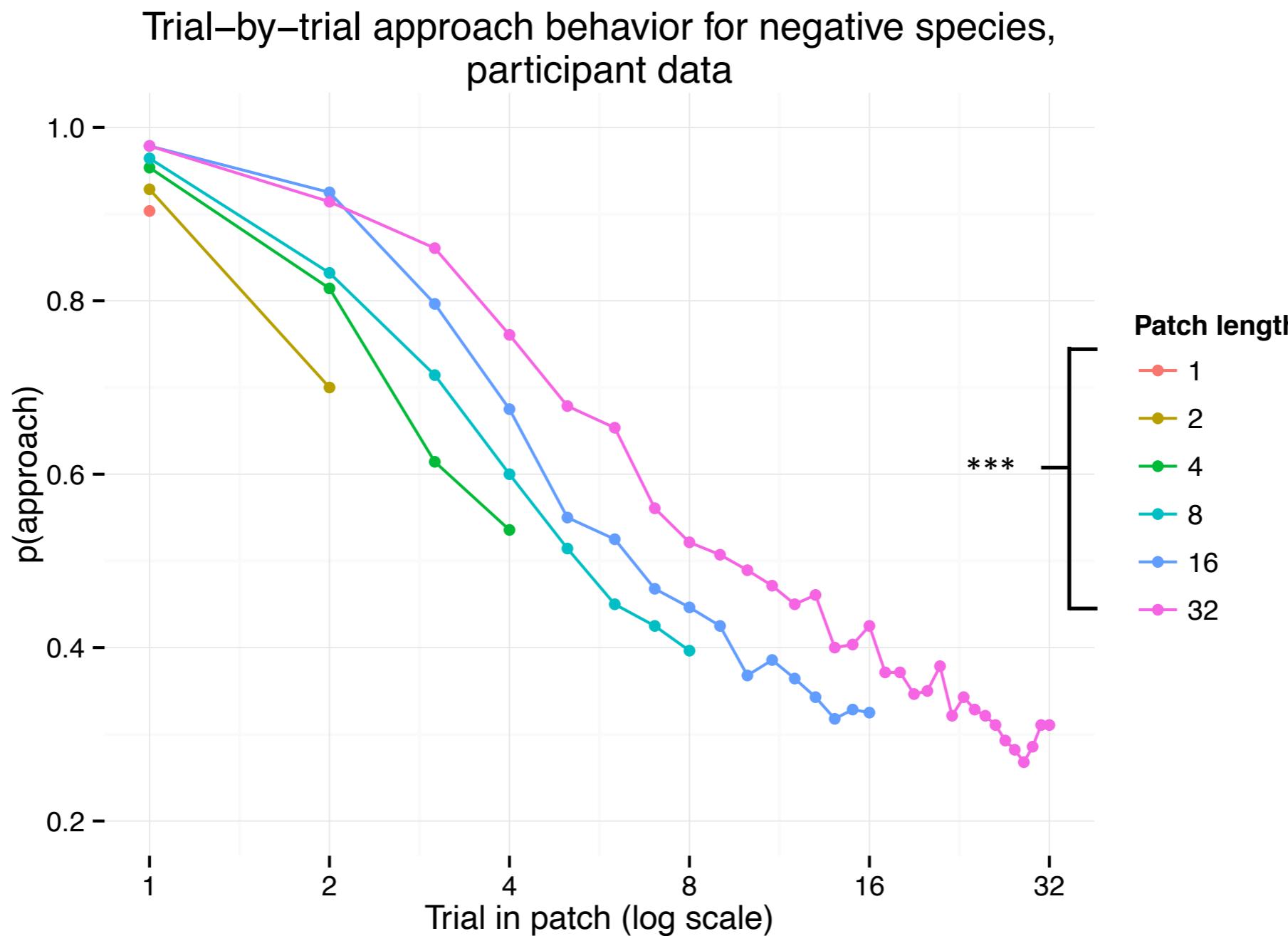
# experiment 1 - results



# experiment 1 - results



# experiment 1 - results



# Generalization and Function Approximation

- Tabular value functions

| state | action $\longrightarrow$ |     |     |     |
|-------|--------------------------|-----|-----|-----|
|       | 0.1                      | 0.3 | 0.7 | 0.2 |
| 0.4   | ...                      | ... | ... | ... |
| 0.2   | ...                      | ... | ... | ... |
|       | $Q(s,a)$                 |     |     |     |

What happens if the size of the state/action space is large?

- Large numbers of states/actions?
- Continuously-valued states/actions?
- Most states never experienced exactly before

Memory

Time

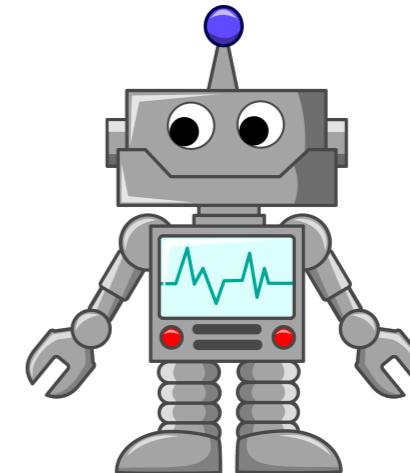
Data

**GENERALISATION:** how experience with small part of state space is used to produce good behaviour over large part of state space

# Generalization and Function Approximation

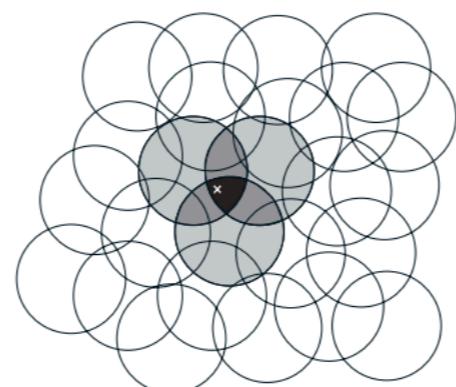
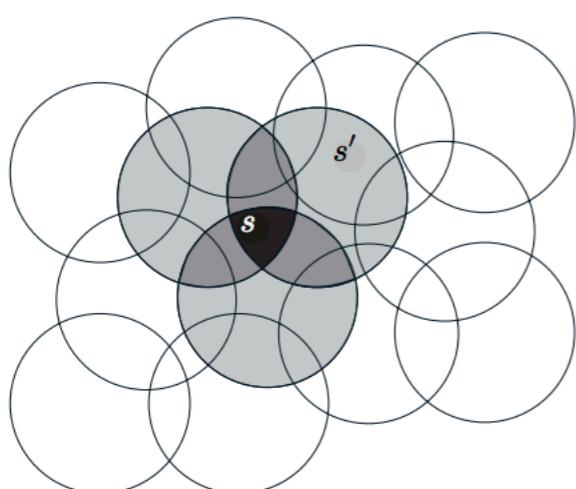
- Instead of directly looking up the states in a kind of addressable-memory, we represent different features of the states

$$\vec{\phi}_s = \begin{pmatrix} x \\ y \\ \text{heading} \\ \text{batterypower} \end{pmatrix}$$

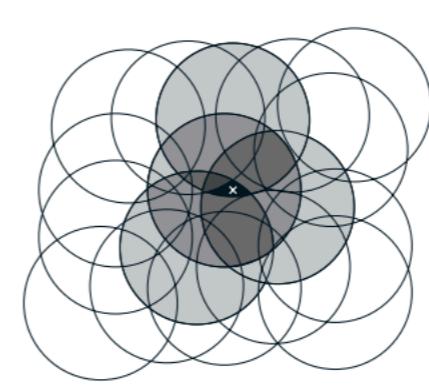


- Position in x, y coordinates (real numbers)
- Heading in degrees w.r.t. north (real number or quantised)
- Battery power = some real number

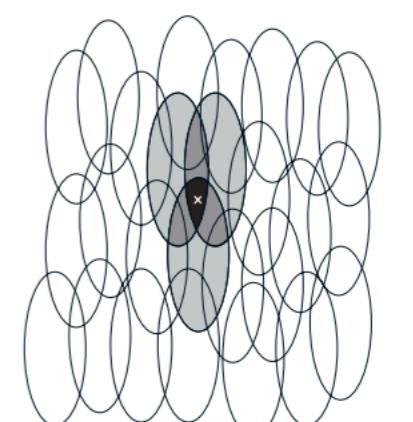
## Coarse coding/tiling



Narrow generalization



Broad generalization



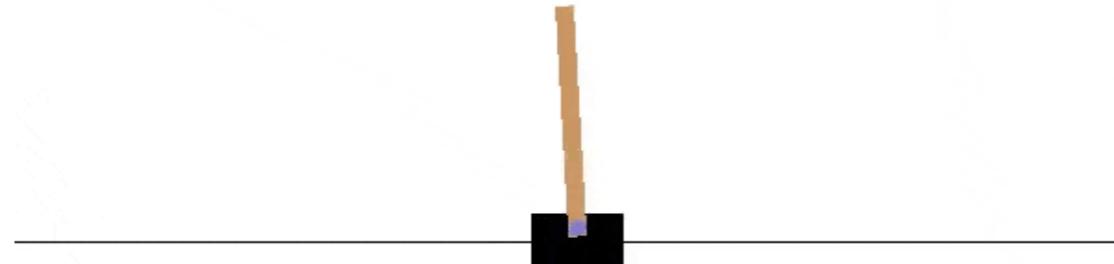
Asymmetric generalization

Figure 9.7: Generalization in linear function approximation methods is determined by the sizes and shapes of the features' receptive fields. All three of these cases have roughly the same number and density of features.

# Generalization and Function Approximation

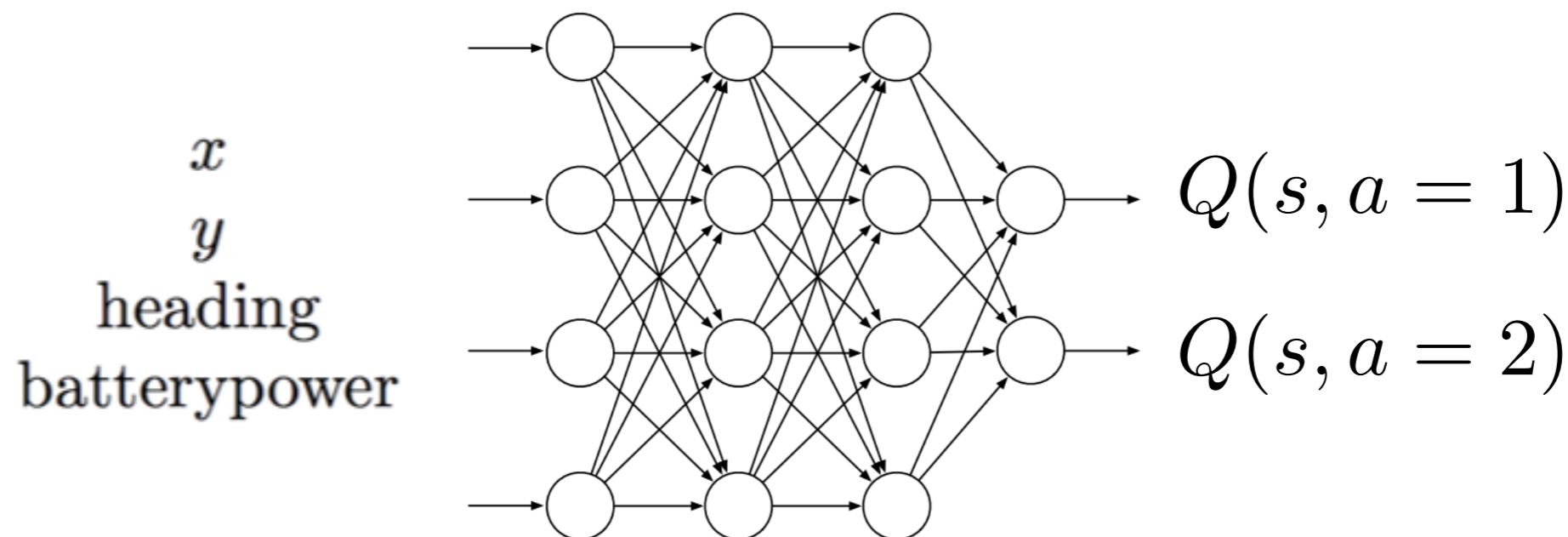
- Instead of directly looking up the states in a kind of addressable-memory, we represent different features of the the states

- position ( $x$ )
- velocity ( $x_{dot}$ )
- angle ( $\theta$ )
- angular velocity ( $\theta_{dot}$ )



# Generalization and Function Approximation

- Neural networks make a useful approximation scheme due to the universal approximation properties



- Train with backprop as in past stuff but the loss function is the Q-learning update rule

$$L = \frac{1}{2} [r + \max_{a'} Q(s', a') - Q(s, a)]^2$$

**Reminder: Backpropagation algorithm for computing gradient**

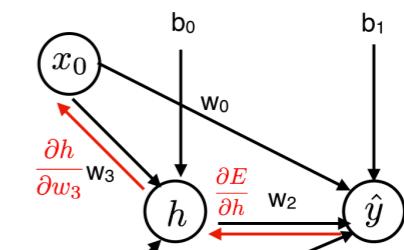
Multi-step strategy:

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial h} \frac{\partial h}{\partial w_3}$$

Step 1) Compute how error changes as a function of hidden unit activation

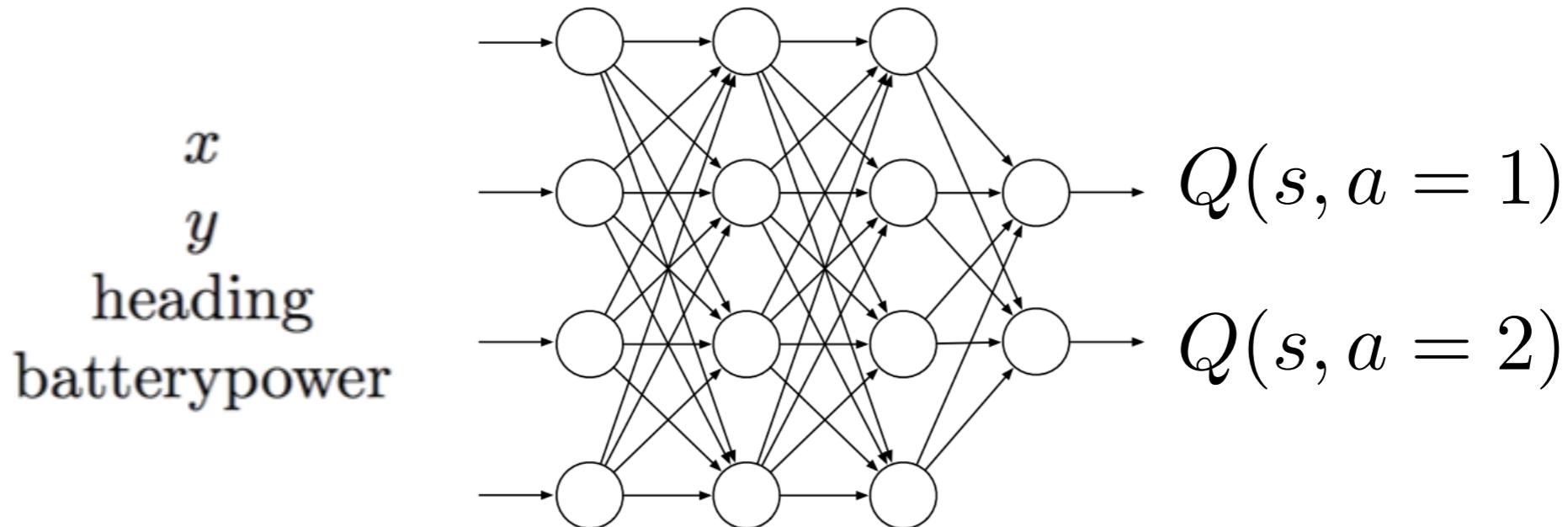
Step 2) Compute how hidden unit activation changes as a function of weight

$$E(w, b) = (\hat{y} - y)^2 = (g(\text{net}) - y)^2$$

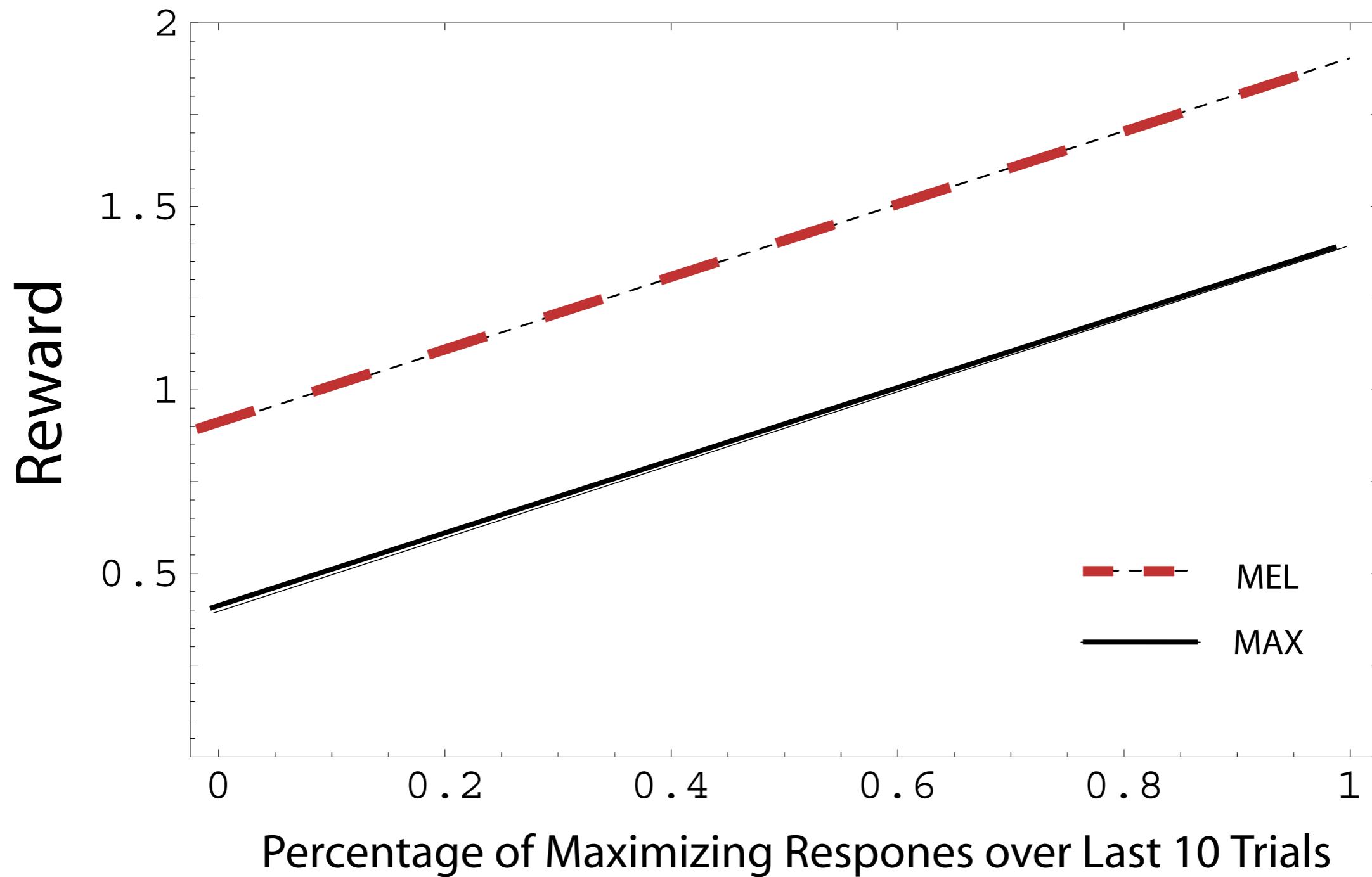


# Q-learning gradient updates

- Do a feedforward pass of the network for the current state  $s$  and get predicted Q values for all actions (output nodes)
- Choose an action using e.g., softmax or epsilon greedy to encourage exploration
- Do a feedforward pass for the network for the next state ( $s'$ ) and get the maximum possible Q-value
- Set Q-value target for the chosen action to  $r + \max_{a'} Q(s', a')$  for the chosen action and zero for all others
- Backpropagate that error as for normal network



$$L = \frac{1}{2} [r + \max_{a'} Q(s', a') - Q(s, a)]^2$$





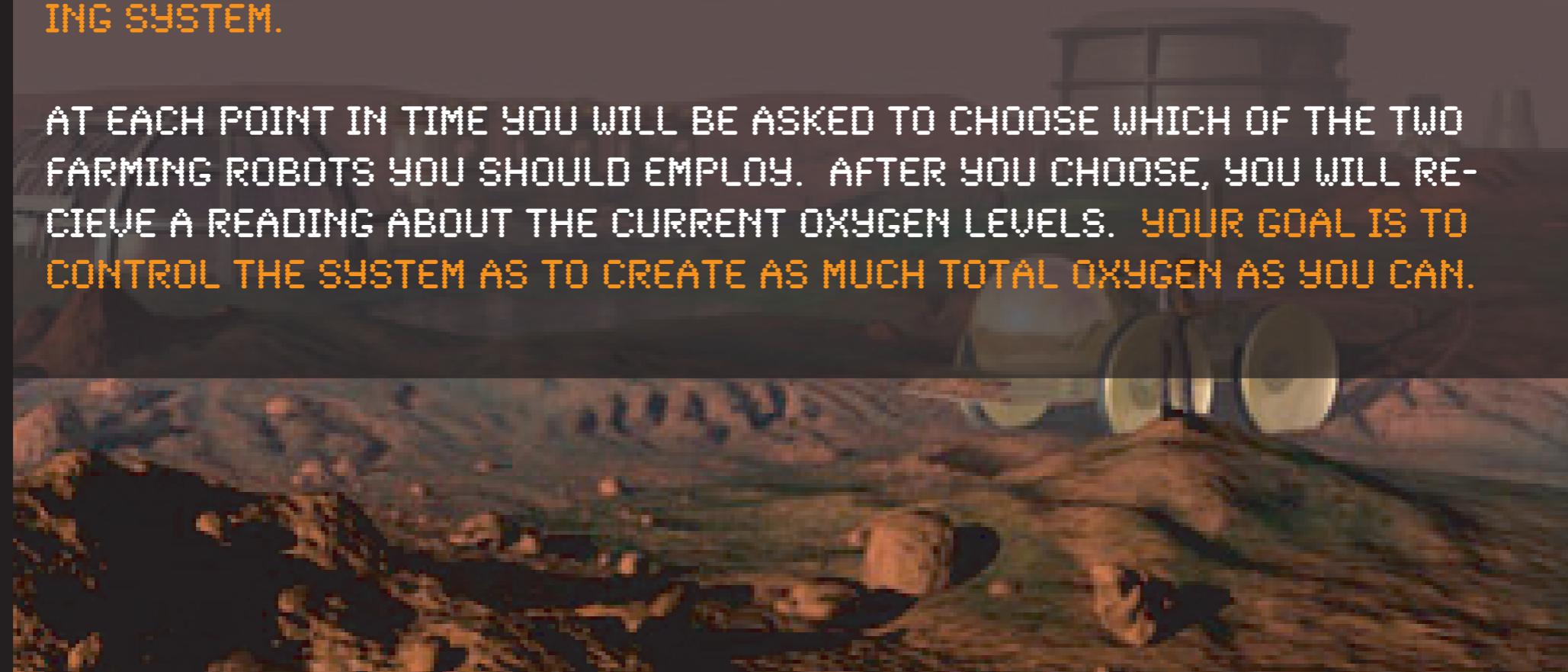
WELCOME TO THE N.A.S.A. MARS FARMING PROJECT

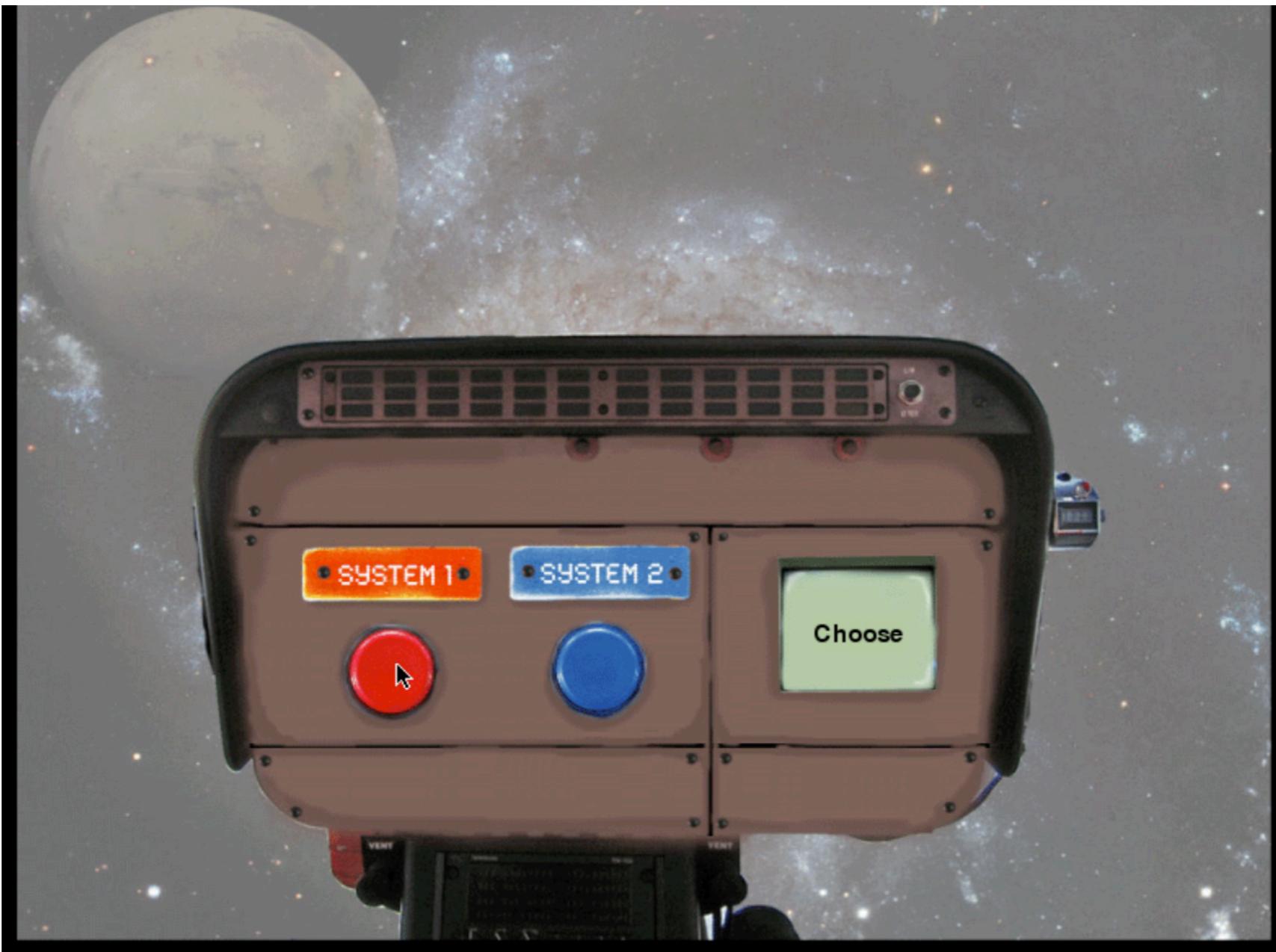
N.A.S.A. HAS ESTABLISHED AN EXPERIMENTAL FARMING BASE ON THE PLANET MARS. CURRENTLY, THIS BASE IS MANNED ENTIRELY BY ROBOT SYSTEMS CONTROLLED FROM EARTH. HOWEVER, N.A.S.A. HOPES THAT THIS EXPERIMENTAL FACILITY MAY SOMEDAY SUPPORT HUMAN OCCUPANTS. THE GOAL OF THE CURRENT PHASE OF THE MISSION IS TO ESTABLISH A BIOLOGICAL FARMING SYSTEM WHICH CAN PROVIDE A SOURCE OF OXYGEN ( $O_2$ ) FOR FUTURE OCCUPANTS.

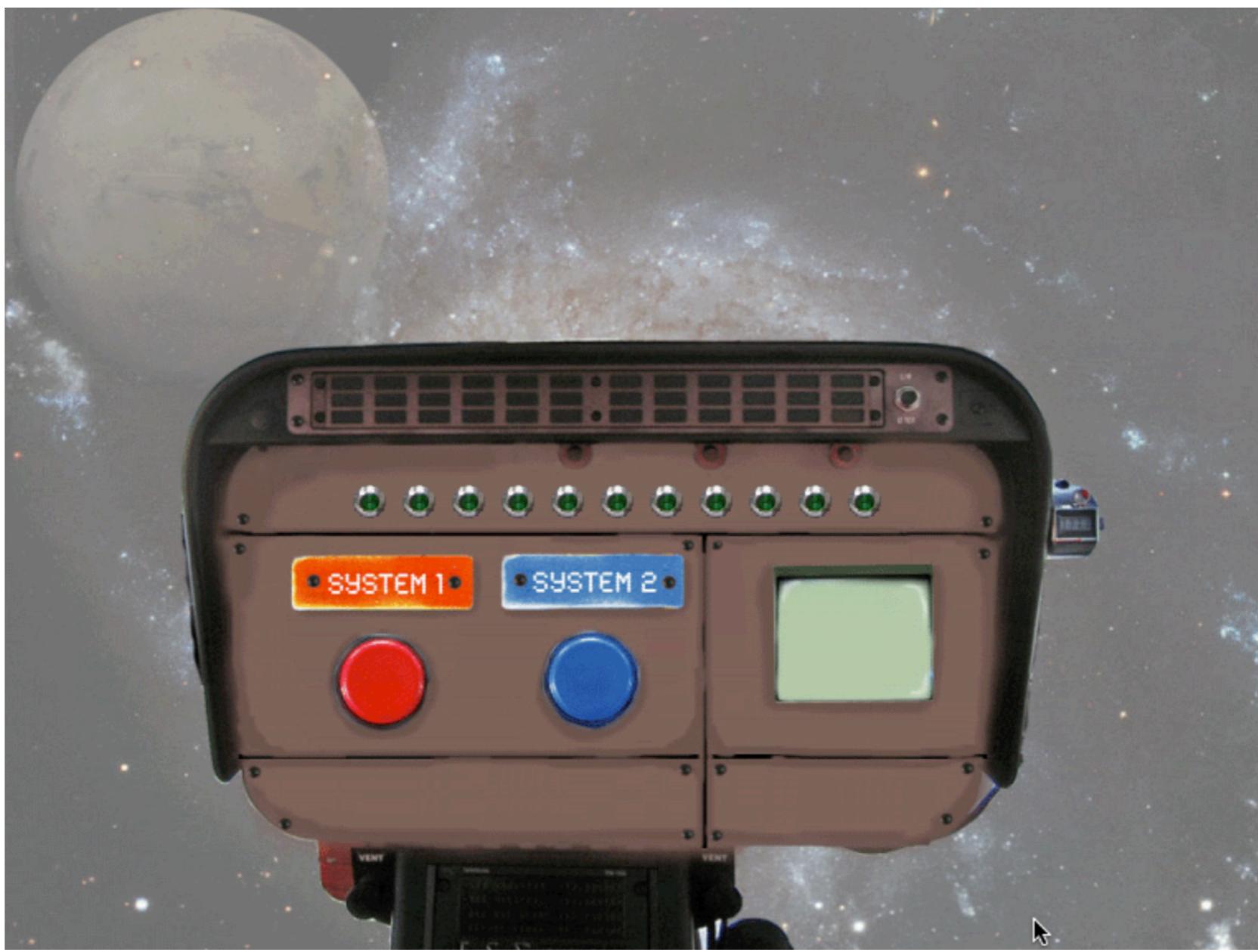


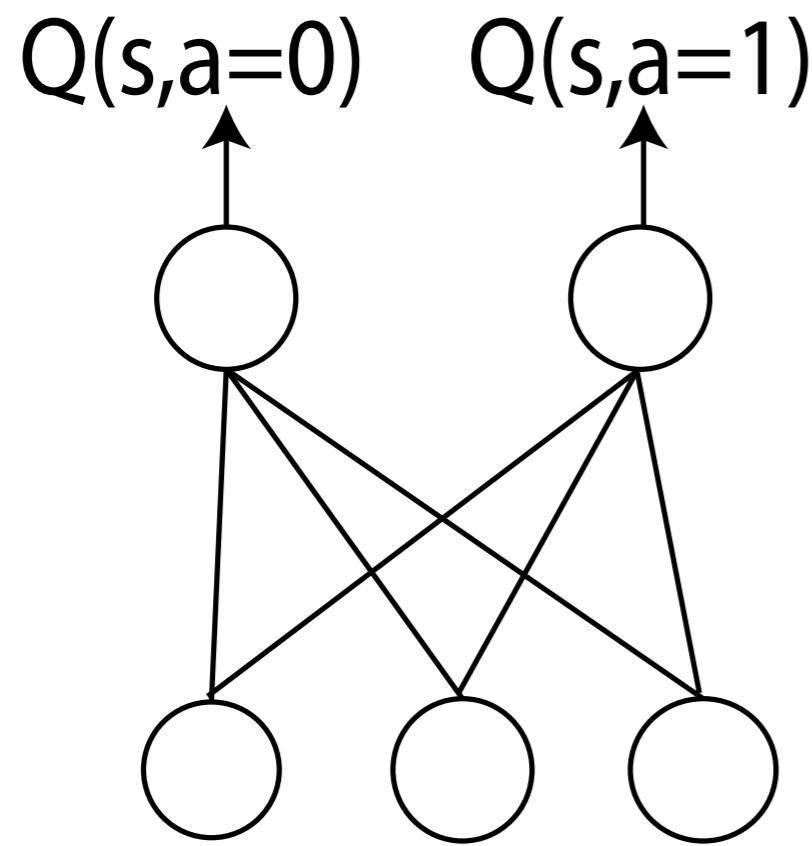
IT IS CURRENTLY UNKNOWN WHICH FARMING METHODS WORK BEST WITH THE MARTIAN SOIL, THUS NASA DEVELOPED TWO INDEPENDENT ROBOT SYSTEMS EACH WHICH SPECIALIZE IN A DIFFERENT SETS OF FARMING TECHNIQUES (SUCH AS THE AMOUNT OF WATER THEY USE, WHAT SPECIES OF PLANTS THEY PREFER, ETC...). YOUR JOB AS MISSION CONTROLLER IS TO EMPLOY THESE ROBOT SYSTEMS IN ORDER TO MAXIMIZE THE TOTAL OUTPUT OF OXYGEN FROM THE FARMING SYSTEM.

AT EACH POINT IN TIME YOU WILL BE ASKED TO CHOOSE WHICH OF THE TWO FARMING ROBOTS YOU SHOULD EMPLOY. AFTER YOU CHOOSE, YOU WILL RECEIVE A READING ABOUT THE CURRENT OXYGEN LEVELS. YOUR GOAL IS TO CONTROL THE SYSTEM AS TO CREATE AS MUCH TOTAL OXYGEN AS YOU CAN.

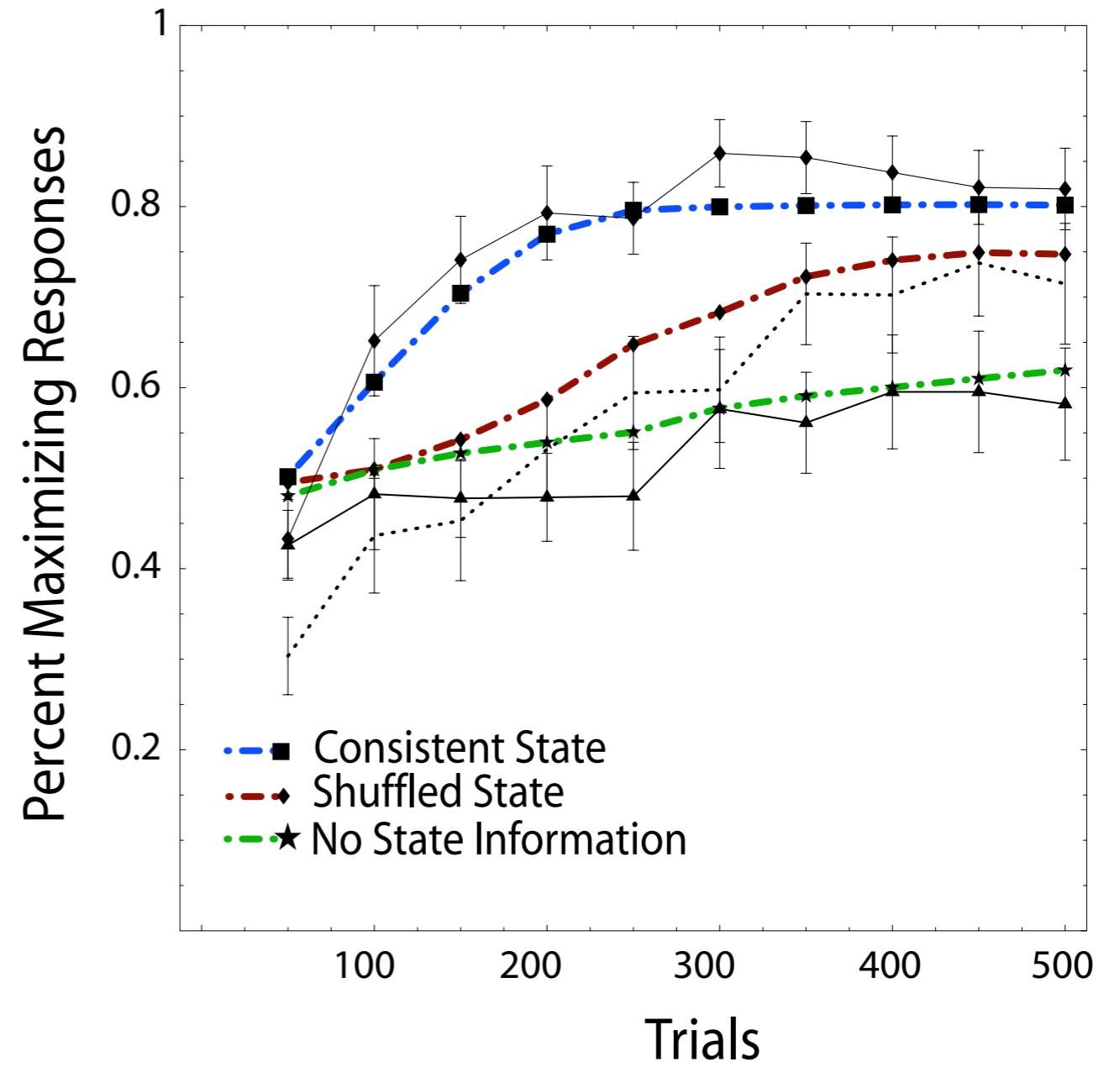








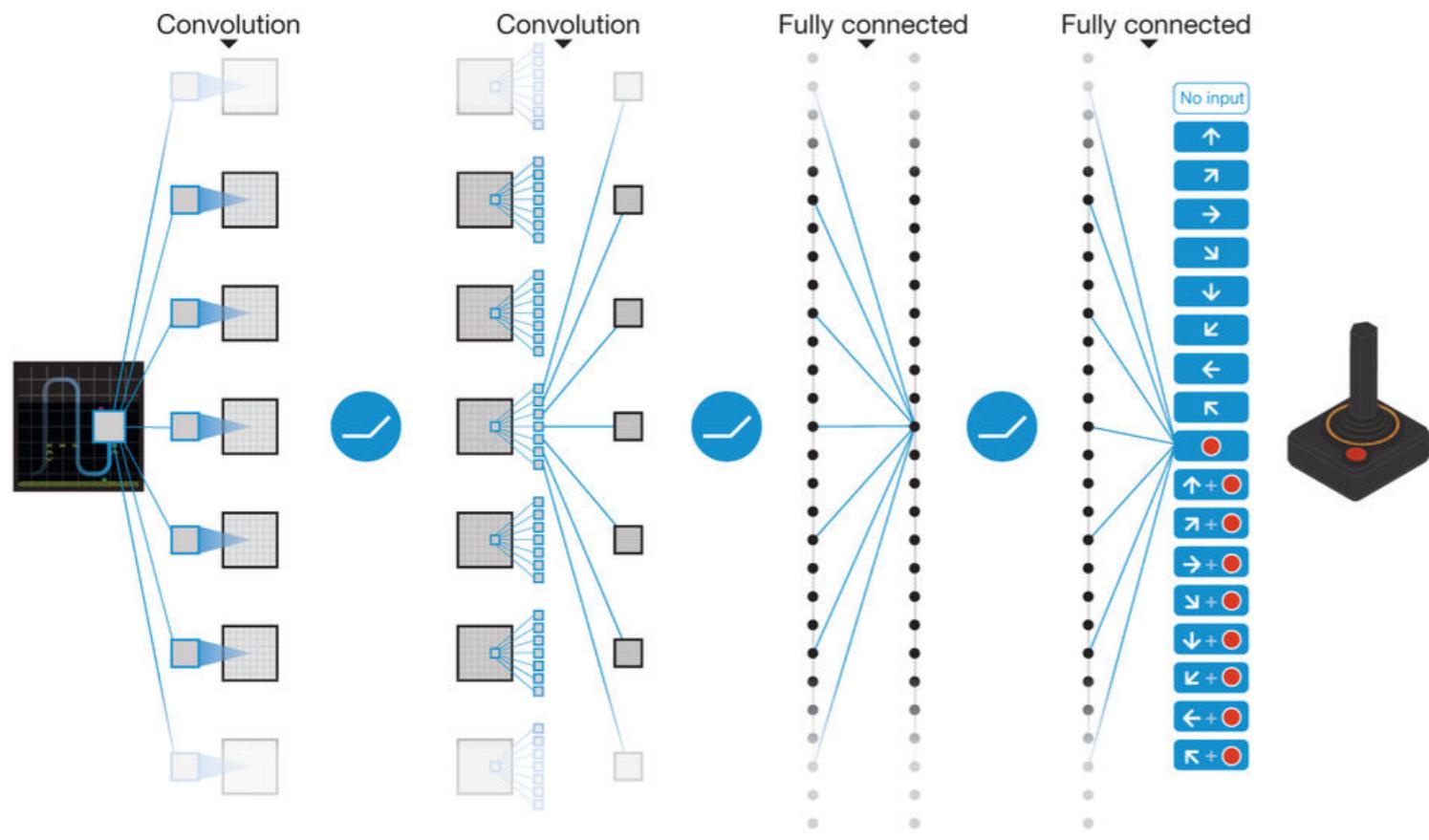
| <b>Position</b> | <b>Last Choice</b> | <b>Bias Unit</b> |
|-----------------|--------------------|------------------|
| of<br>Light     | [0/1]              |                  |
|                 |                    | [0-10]           |



$$Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

$$L = \frac{1}{2} [r + \max_{a'} Q(s', a') - Q(s, a)]^2$$

The way the problem is represented to the network makes it easier to learn the state representation matters!



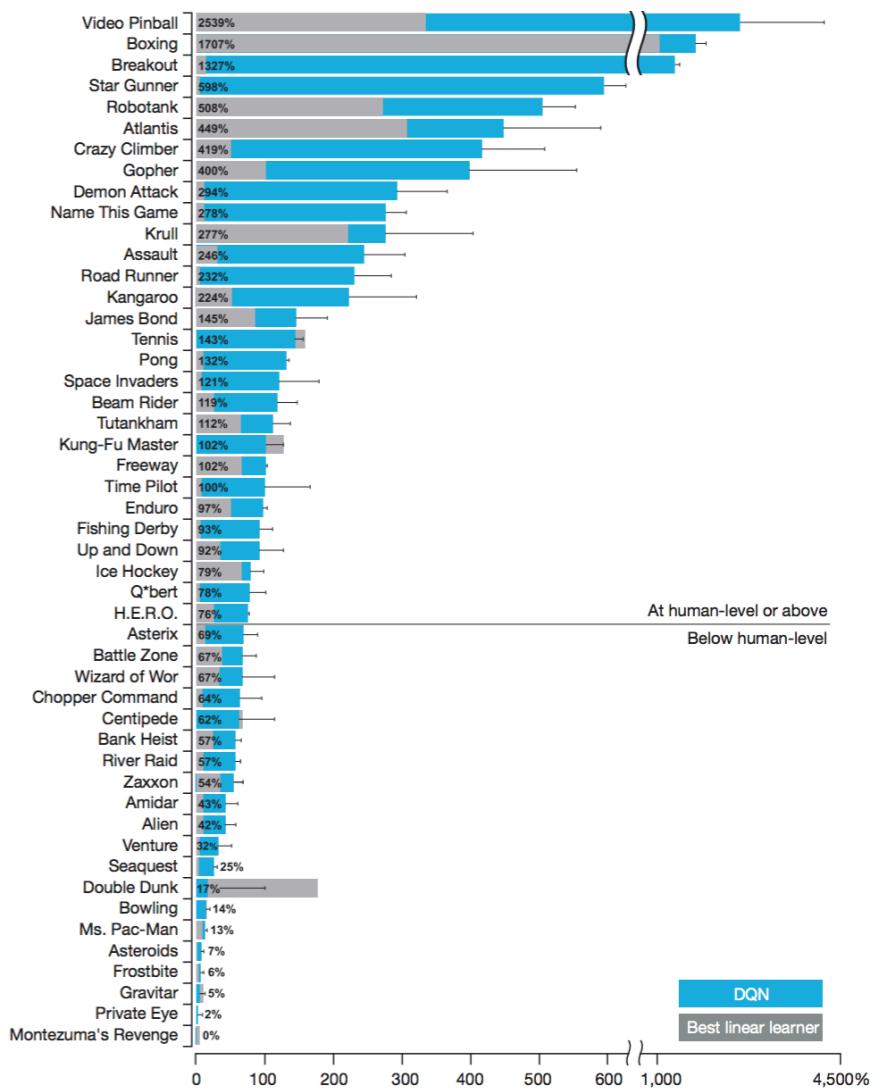
# LETTER

doi:10.1038/nature14236

## Human-level control through deep reinforcement learning

Volodymyr Mnih<sup>1\*</sup>, Koray Kavukcuoglu<sup>1\*</sup>, David Silver<sup>1\*</sup>, Andrei A. Rusu<sup>1</sup>, Joel Veness<sup>1</sup>, Marc G. Bellemare<sup>1</sup>, Alex Graves<sup>1</sup>, Martin Riedmiller<sup>1</sup>, Andreas K. Fidjeland<sup>1</sup>, Georg Ostrovski<sup>1</sup>, Stig Petersen<sup>1</sup>, Charles Beattie<sup>1</sup>, Amir Sadik<sup>1</sup>, Ioannis Antonoglou<sup>1</sup>, Helen King<sup>1</sup>, Dharshan Kumaran<sup>1</sup>, Daan Wierstra<sup>1</sup>, Shane Legg<sup>1</sup> & Demis Hassabis<sup>1</sup>

**DQN!**



# Experience Replay

- Combining neural networks with TD methods means you are combining two sample-intensive methods together. The neural network needs lots of training to converge, but also you need lots of samples to estimate the value function or q-value function.
- One way to “help” the neural network is through “experience replay” where you store the  $\langle s, a, r, s' \rangle$  tuples you experience through your actions in the environment.
- Using batch updates to the backprop weights (meaning compute the gradients for a bunch of training patterns all at once accumulating them at each point in the network and applying the update rule once).
- Can help network pop out of local minima by getting rid of some correlations in the training data

I  
HAD  
THE  
DREAM  
AGAIN.

# Next time

- Model-based RL/Planning

## Three levels of description (*David Marr, 1982*)

### Computational

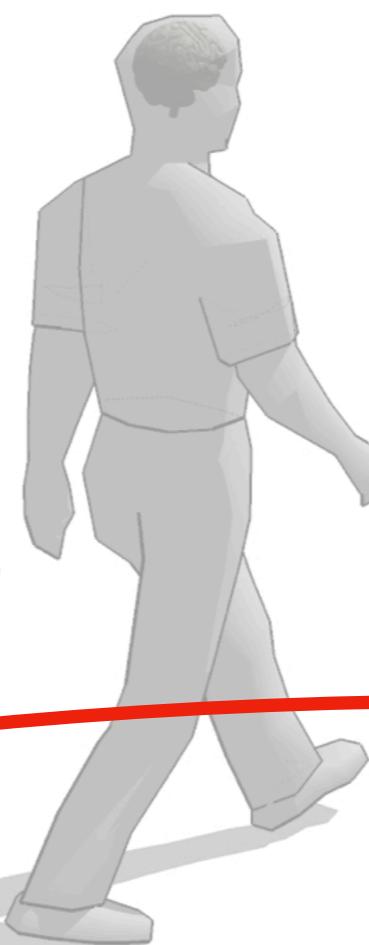
Why do things work the way they do?  
What is the goal of the computation?  
What are the unifying principles?

### Algorithmic

What representations can implement such computations?  
How does the choice of representations determine the algorithm?

### Implementational

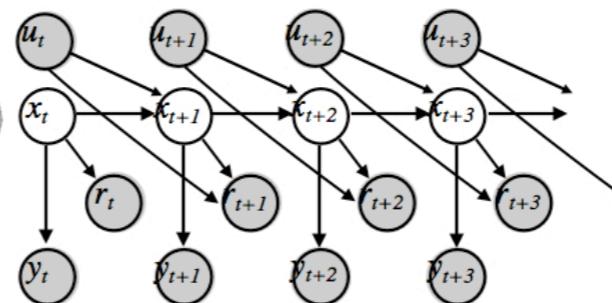
How can such a system be built in hardware?  
How can neurons carry out the computations?



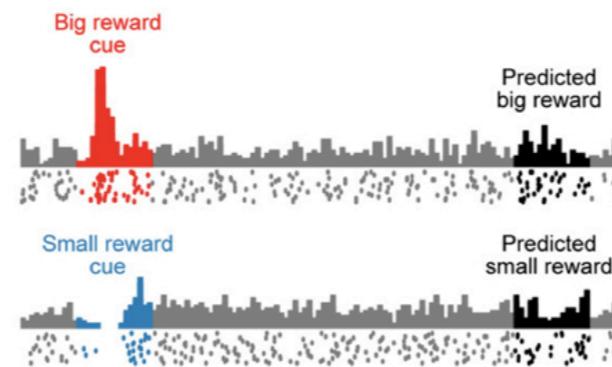
maximize:

$$R_t = r_{t+1} + r_{t+2} + \dots + r_T$$

Bellman



Dynamic programming,  
TD methods, Monte  
Carlo



Neural firing patterns,  
prediction errors,  
system level  
neuroscience

# Slide Credits

Nathaniel Daw (exploration/gittins)  
Alex Rich  
Gillian Hayes (TD methods/explore)  
Rich Sutton (general approach)  
Andy Barto (general approach)