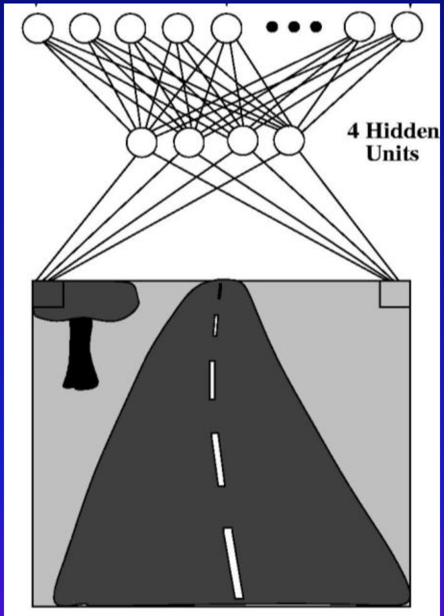
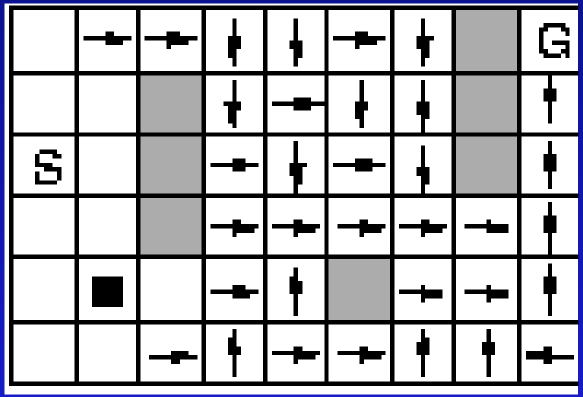
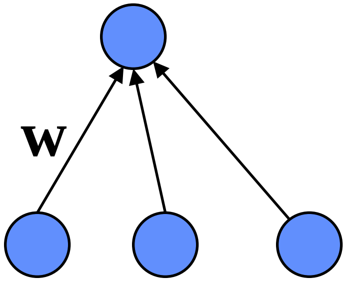
CSE/NB 528
Lecture 14:
From Supervised to Reinforcement Learning
(Chapter 9)





Recall from last time: Sigmoid Networks

Output $v = g(\mathbf{w}^T\mathbf{u}) = g(\sum_i w_i u_i)$

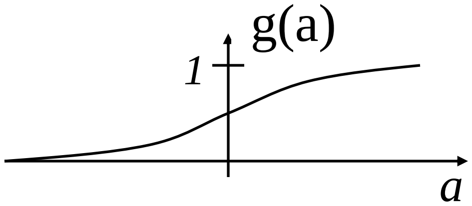


Input nodes

$\mathbf{u} = (u_1 \quad u_2 \quad u_3)^T$

Sigmoid output function:

$$g(a) = \frac{1}{1 + e^{-\beta a}}$$



Sigmoid is a non-linear “squashing” function: Squashes input to be between 0 and 1. Parameter β controls the slope.

What should we optimize?

- Given training examples ($\mathbf{u}^m$,d^m) (m = 1, …, N), define the output error function:

$$E(\mathbf{w}) = \frac{1}{2}(d^m - v^m)^2$$

where $$v^m = g(\mathbf{w}^T \mathbf{u}^m)$$

How would you change **w** so that $E(\mathbf{w})$ is minimized?

Learning the Synaptic Weights

- How would you change **w** so that $E(\mathbf{w})$ is minimized?
  - Gradient Descent: Change **w** in proportion to $-dE/d\mathbf{w}$ (why?)

$$\mathbf{w} \rightarrow \mathbf{w} - \varepsilon \frac{dE}{d\mathbf{w}} \qquad E(\mathbf{w}) = \frac{1}{2}(d^m - v^m)^2$$

$$\frac{dE}{d\mathbf{w}} = -\underbrace{(d^m - v^m)}_{\text{delta = error}} g'(\mathbf{w}^T \mathbf{u}^m)\mathbf{u}^m$$

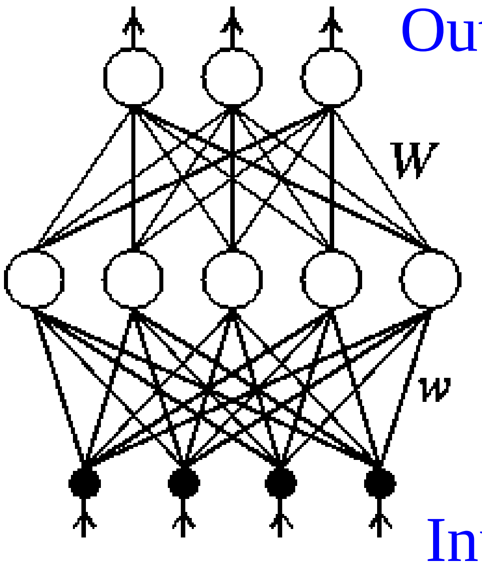
Derivative of sigmoid

Also known as the "delta rule" or "LMS (least mean square) rule"

But wait….

- What if we have multiple layers?

$$v_i = g\left(\sum_j W_{ji}\, g\left(\sum_k w_{kj} u_k\right)\right)$$



Output $\mathbf{v} = (v_1\ v_2\ \ldots\ v_J)^T$; Desired = $\mathbf{d}$

Delta rule can be used to adapt these weights

How do we adapt these?
(no desired output provided here)

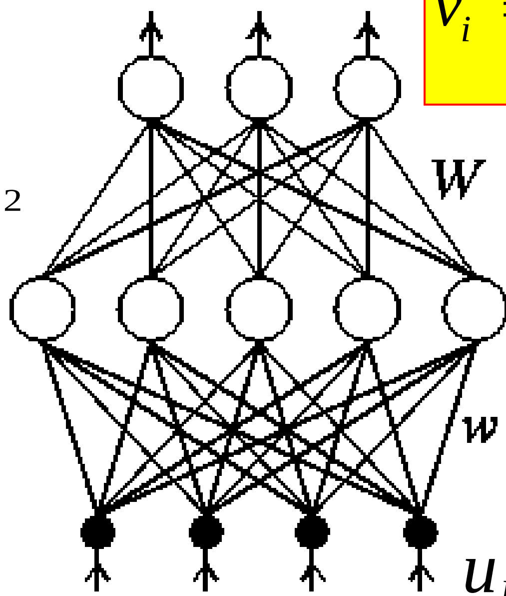
Input $\mathbf{u} = (u_1\ u_2\ \ldots\ u_K)^T$

Enter…the backpropagation algorithm

(Actually, nothing but the chain rule from calculus)

Uppermost layer (delta rule)

$$E(\mathbf{W},\mathbf{w}) = \frac{1}{2}\sum_i (d_i - v_i)^2$$



$$v_i = g(\sum_j W_{ji} x_j)$$

$$x_j$$

$$u_k$$

Learning rule for hidden-output weights **W**:

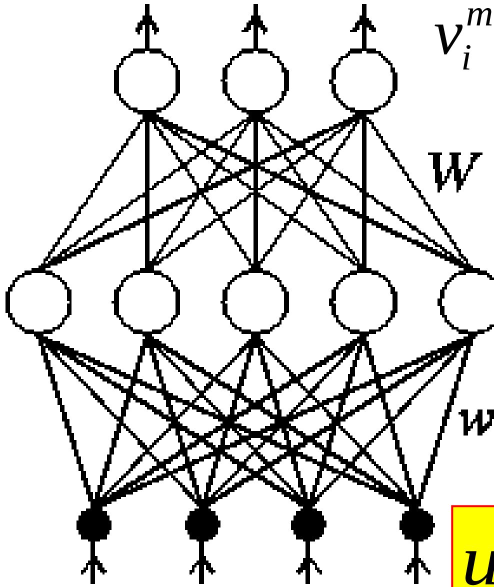
$$W_{ji} \rightarrow W_{ji} - \varepsilon \frac{dE}{dW_{ji}}$$ {*gradient descent*}

$$\frac{dE}{dW_{ji}} = -(d_i - v_i) g'(\sum_j W_{ji} x_j) x_j$$ {*delta rule*}

Backpropagation: Inner layer (chain rule)

$$E(\mathbf{W},\mathbf{w}) = \frac{1}{2}\sum_i (d_i - v_i)^2$$

$$v_i^m = g(\sum_j W_{ji} x_j)$$



$$\boxed{x_j^m = g(\sum_k w_{kj} u_k^m)}$$

$$\boxed{u_k^m}$$

Learning rule for input-hidden weights w:

$$w_{kj} \to w_{kj} - \varepsilon \frac{dE}{dw_{kj}} \qquad \text{But}: \frac{dE}{dw_{kj}} = \frac{dE}{dx_j} \cdot \frac{dx_j}{dw_{kj}}$$ **{*chain rule*}**

$$\frac{dE}{dw_{kj}} = \left[-\sum_{m,i} (d_i^m - v_i^m) g'(\sum_j W_{ji} x_j^m) W_{ji} \right] \cdot \left[g'(\sum_k w_{kj} u_k^m) u_k^m \right]$$

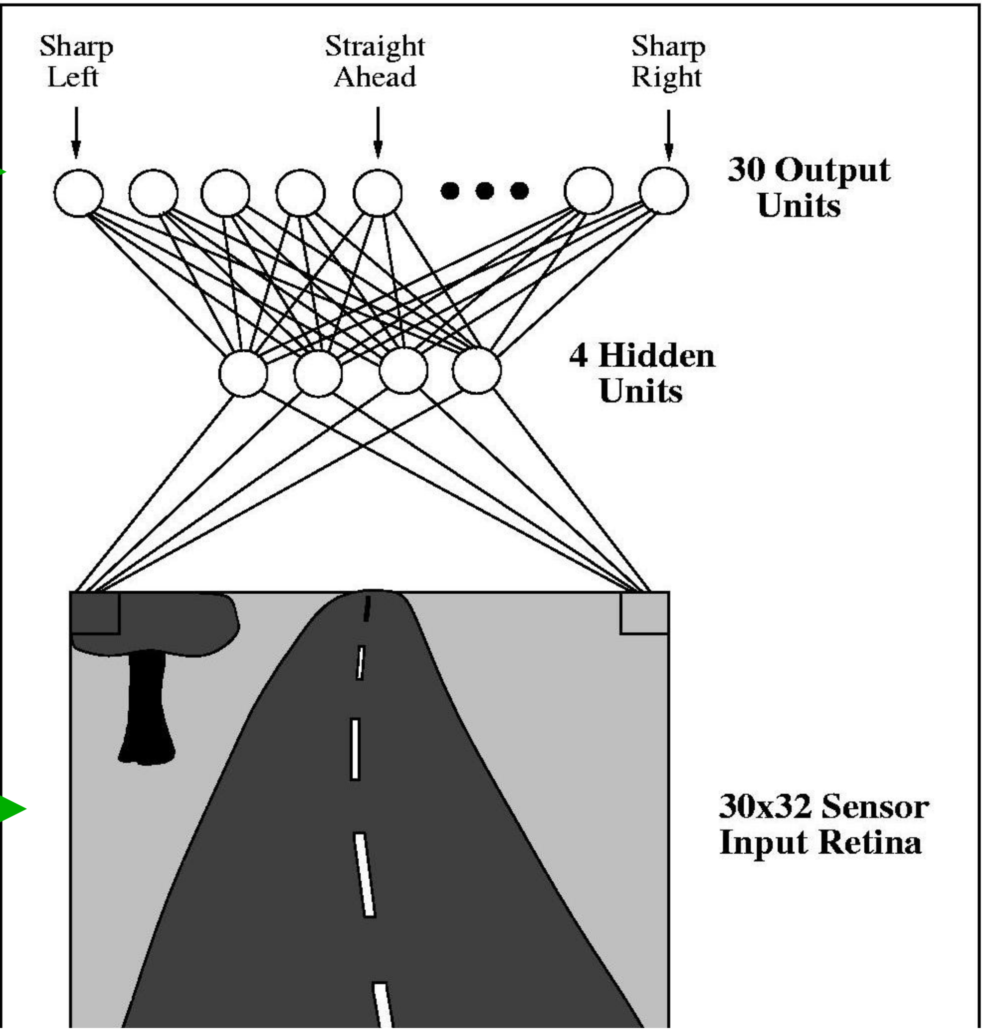
Example: Learning to Drive

Example Network

Get steering angle

Training Output:

$\mathbf{d} = (d_1 \ d_2 \ \ldots \ d_{30})$

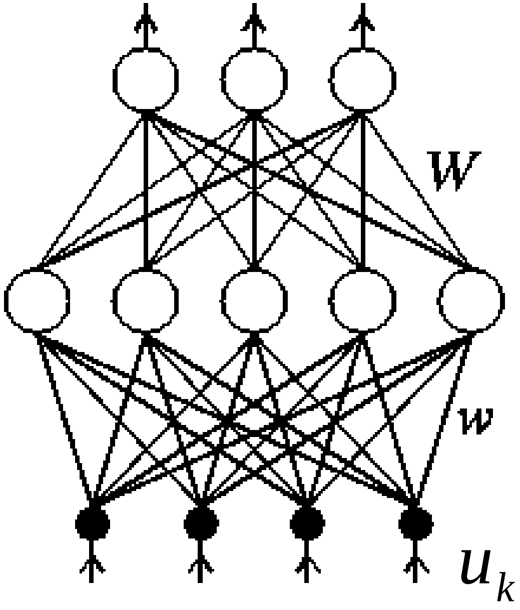


Get current camera image

Training Input $\mathbf{u} = (u_1 \ u_2 \ \ldots \ u_{960})$ = image pixels

(Pomerleau, 1992)

Training the network using backprop



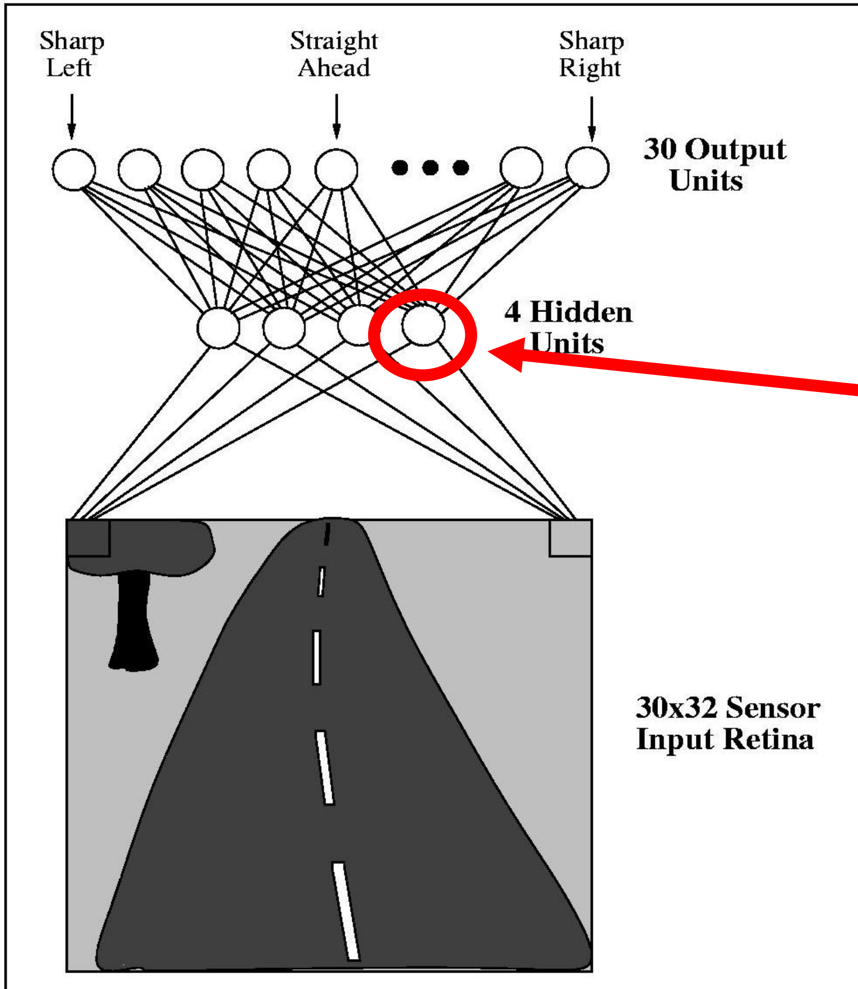
Start with random weights **W**, **w**

Given input **u**, network produces output **v**

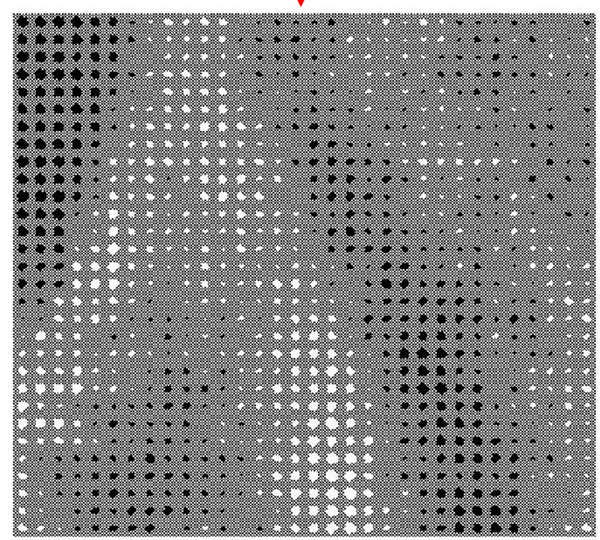
Use backprop to learn **W** and **w** that minimize total error over all output units (labeled *i*):

$$E(\mathbf{W},\mathbf{w}) = \frac{1}{2}\sum_i (d_i - v_i)^2$$

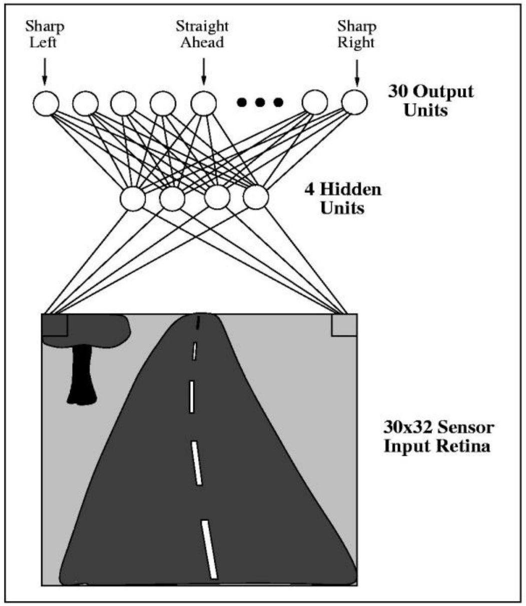
Learning to Drive using Backprop



One of the learned "road features" w_i



ALVINN (Autonomous Land Vehicle in a Neural Network)



CMU Navlab

Trained using human driver + camera images

After learning:

- Drove up to 70 mph on highway
- Up to 22 miles without intervention
- Drove cross-country largely autonomously

(Pomerleau, 1992)

But that doesn’t help me find food in a maze

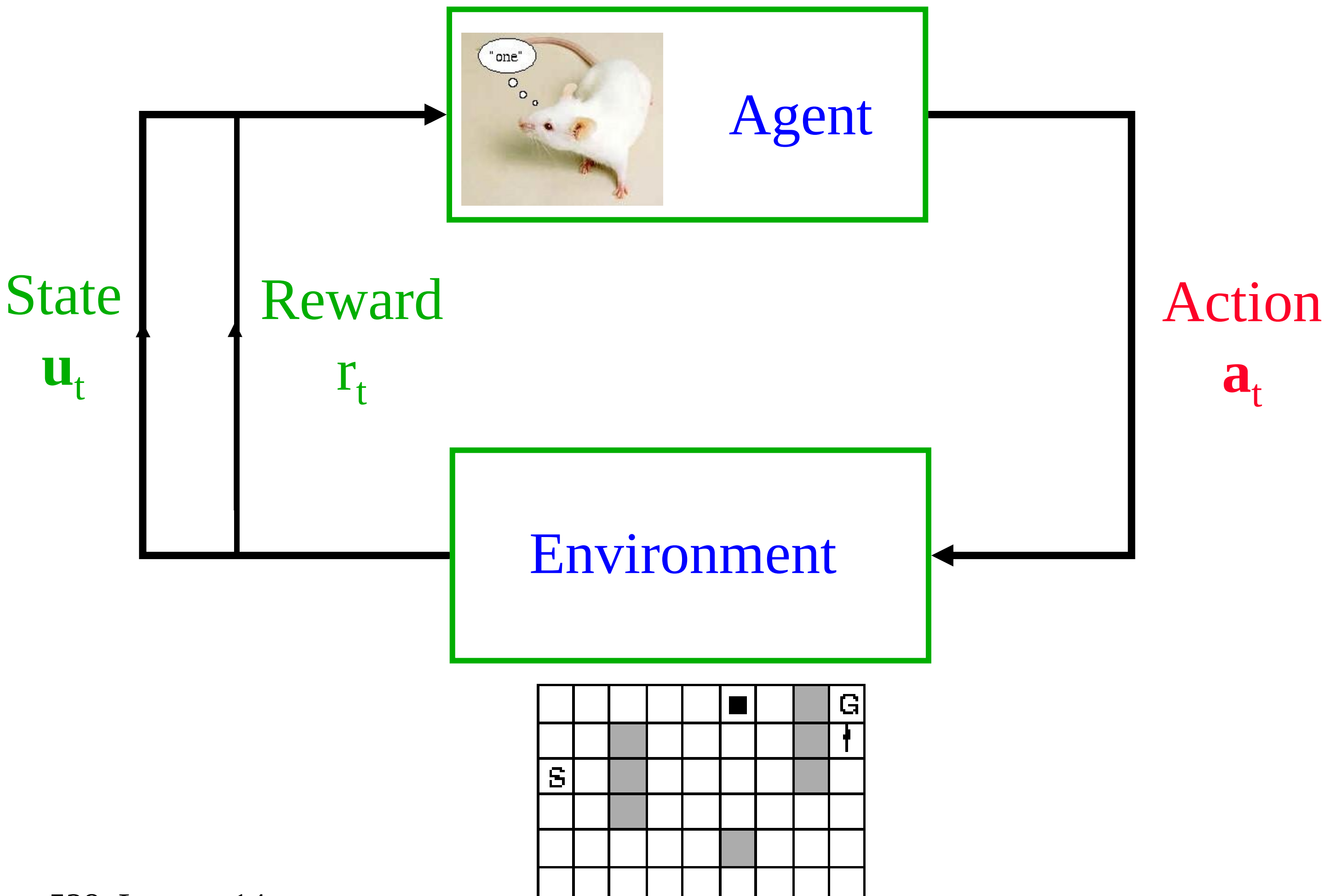
Humans (and animals in general) don't get exact supervisory signals (commands for muscles) for learning to talk, walk, ride a bicycle, play the piano, drive, etc.

We learn by trial-and-error
(with hints from others)

Might get "rewards and punishments" along the way

Enter…Reinforcement Learning

The Reinforcement Learning "Agent"

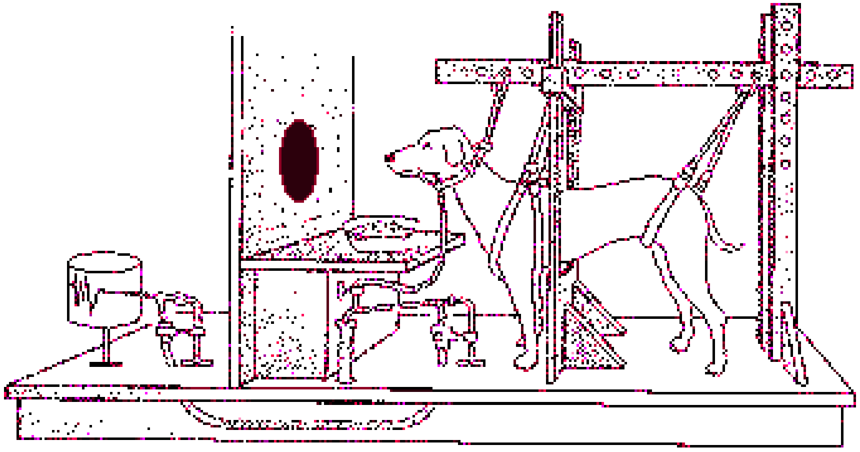


The Reinforcement Learning Framework

- **Unsupervised learning:** Learn the hidden causes of inputs
- **Supervised learning:** Learn a function based on training examples of (input, desired output) pairs
- **Reinforcement Learning:** Learn the best action for any given state so as to maximize total expected (future) reward
  - Intermediate between unsupervised and supervised learning Instead of explicit teaching signal (or desired output), you get *rewards or punishments*
  - Inspired by classical conditioning experiments

Early Results: Pavlov and his Dog

- Classical (Pavlovian) conditioning experiments
- Training: Bell → Food
- After: Bell → Salivate
- Conditioned stimulus (bell) predicts future reward (food)



(http://employees.csbsju.edu/tcreed/pb/pdoganim.html)

Predicting Delayed Rewards

- Reward is typically delivered at the end (when you know whether you succeeded or not)
- Time: $0 \le t \le T$ with stimulus $u(t)$ and reward $r(t)$ at each time step t (Note: $r(t)$ can be zero at some time points)
- Key Idea: Make the output $v(t)$ predict *total expected future reward* starting from time t

$$v(t) \approx \left\langle \sum_{\tau=0}^{T-t} r(t+\tau) \right\rangle$$

Learning to Predict Delayed Rewards

- Use a set of modifiable weights $w(t)$ and *predict based on all past stimuli* $u(t)$:

$$v(t) = \sum_{\tau=0}^{t} w(\tau) u(t-\tau)$$

- Would like to find the weights (or filter) $w(\tau)$ that minimize:

$$\left(\sum_{\tau=0}^{T-t} r(t+\tau) - v(t) \right)^2$$

(Can we minimize this using gradient descent and delta rule?)

Yes, BUT…not yet available are the future rewards

Temporal Difference (TD) Learning

- **Key Idea**: Rewrite squared error to get rid of future terms:

$$\left(\sum_{\tau=0}^{T-t} r(t+\tau) - v(t)\right)^2 = \left(r(t) + \sum_{\tau=0}^{T-t-1} r(t+1+\tau) - v(t)\right)^2$$

$$\approx \left(r(t) + v(t+1) - v(t)\right)^2$$

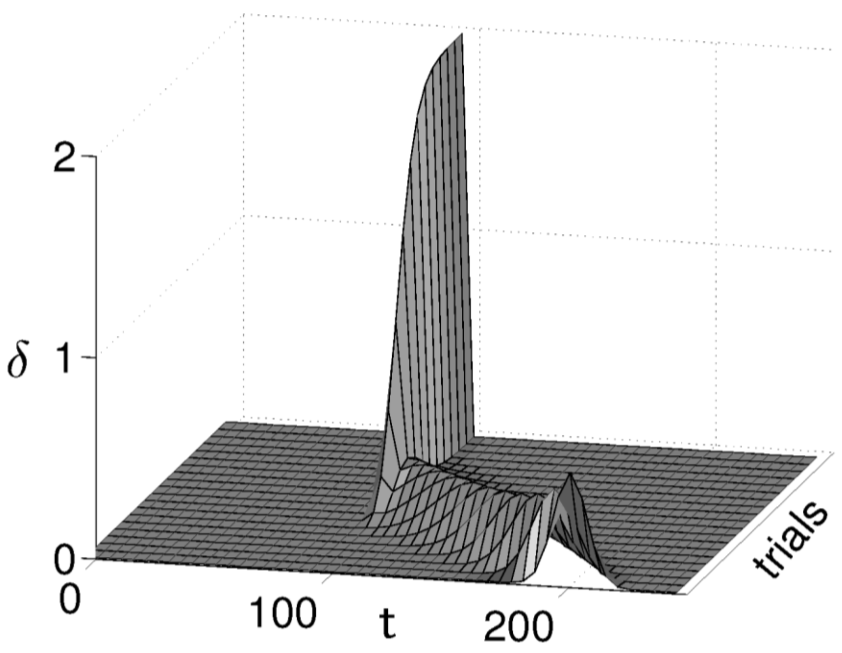
- **Temporal Difference (TD) Learning**:

$$w(\tau) \rightarrow w(\tau) + \varepsilon\left[\overbrace{\underbrace{r(t) + v(t+1)}_{\text{Expected future reward}} - v(t)}^{\delta}\right] u(t-\tau)$$

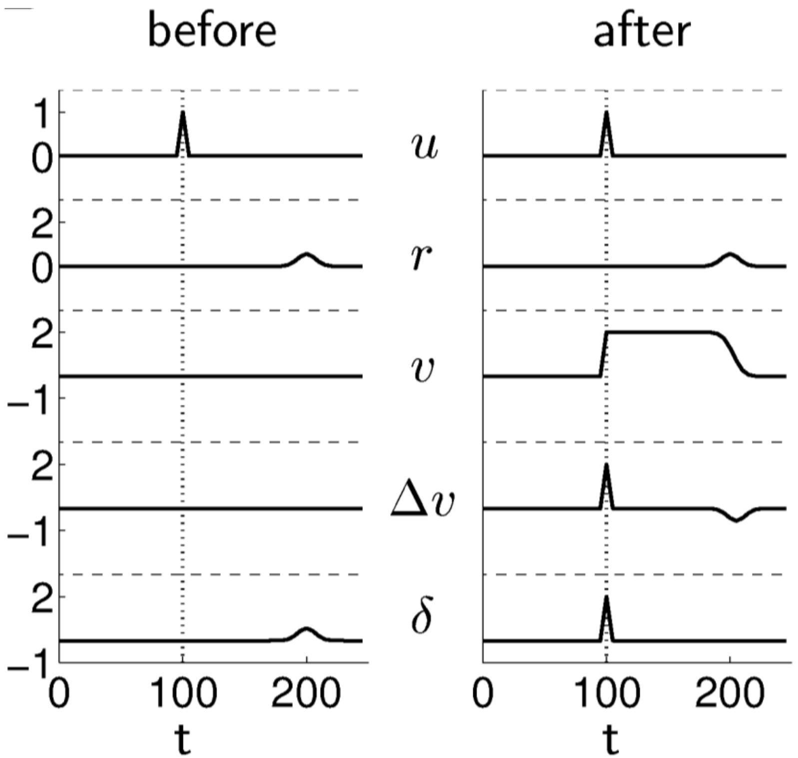
Prediction

Predicting Delayed Reward: TD Learning

Stimulus at t = 100 and reward at t = 200



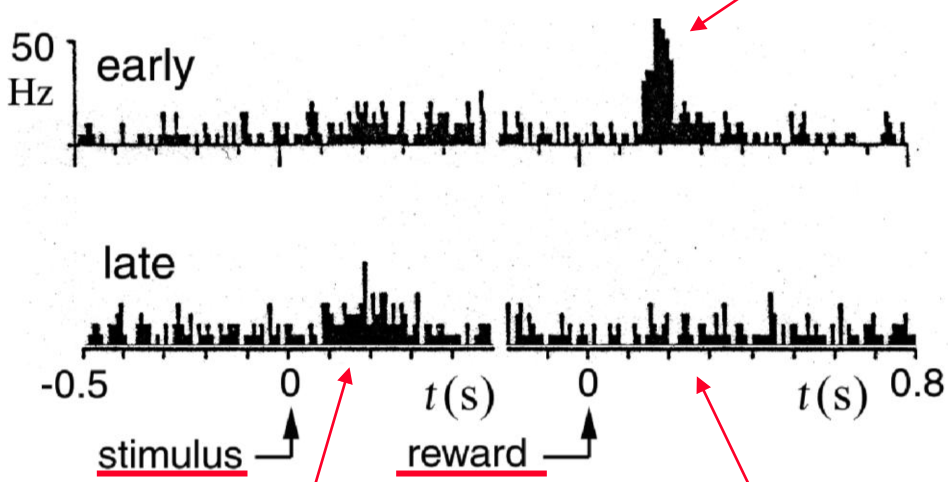
Prediction error δ for each time step
(over many trials)



Reward Prediction Error in the Primate Brain?

Dopaminergic cells in Ventral Tegmental Area (VTA)

Reward Prediction error? $[r(t)+v(t+1)-v(t)]$



Before Training

After Training

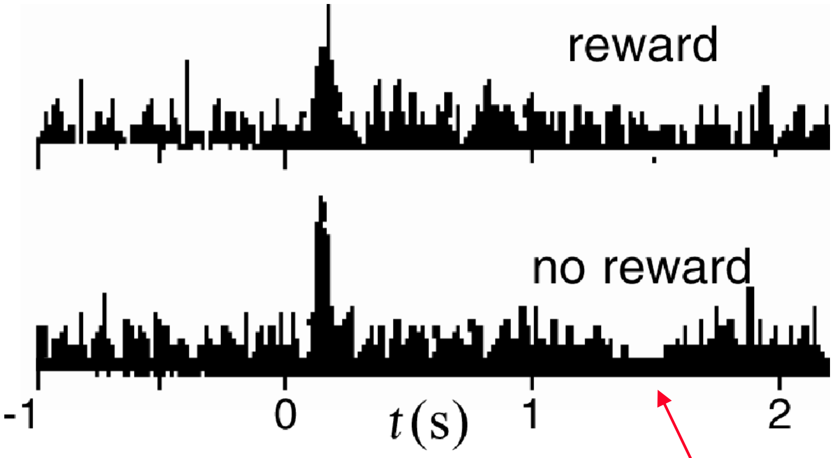
$[0+v(t+1)-v(t)]$

No error

$v(t)\approx r(t)+v(t+1)$

More Evidence for Prediction Error Signals

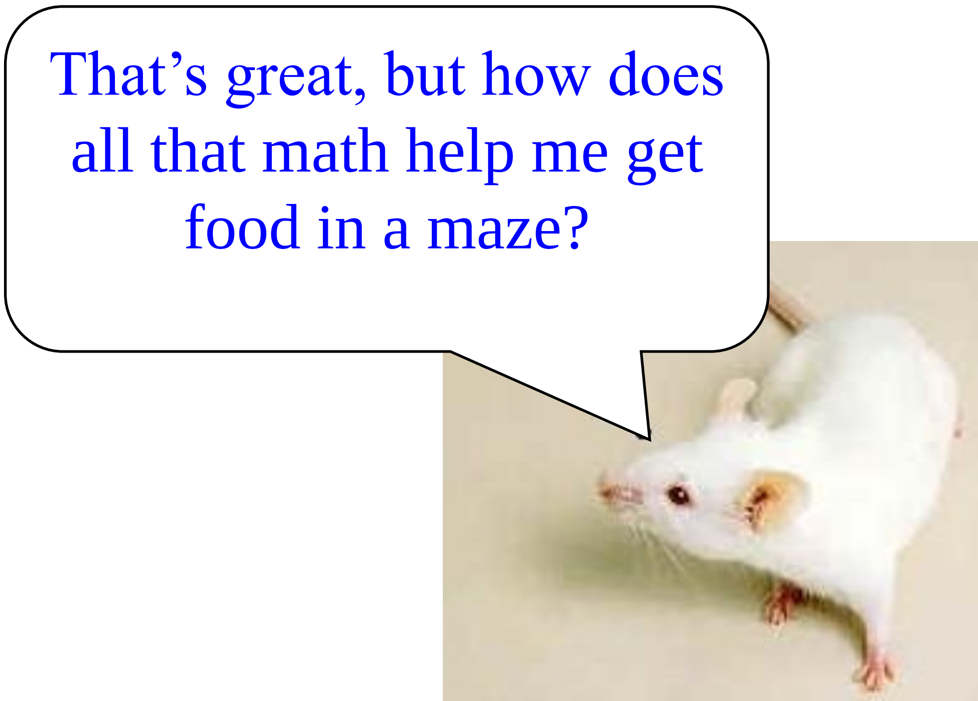
Dopaminergic cells in VTA



Negative error

$$r(t) = 0, v(t+1) = 0$$

$$[r(t) + v(t+1) - v(t)] = -v(t)$$


That's great, but how does all that math help me get food in a maze?

Using Reward Predictions to Select Actions

- Suppose you have computed a "Value" for each action
- $Q(a)$ = value (predicted reward) for executing action a
  - Higher if action yields more reward, lower otherwise
- Can select actions probabilistically according to their value:

$$P(a) = \frac{\exp(\beta Q(a))}{\sum_{a'} \exp(\beta Q(a'))}$$

(High β selects actions with highest Q value. Low β selects more uniformly)

Simple Example: Bee Foraging

- Experiment: Bees select either yellow (y) or blue (b) flowers based on nectar reward

- Idea: Value of yellow/blue = average reward obtained so far

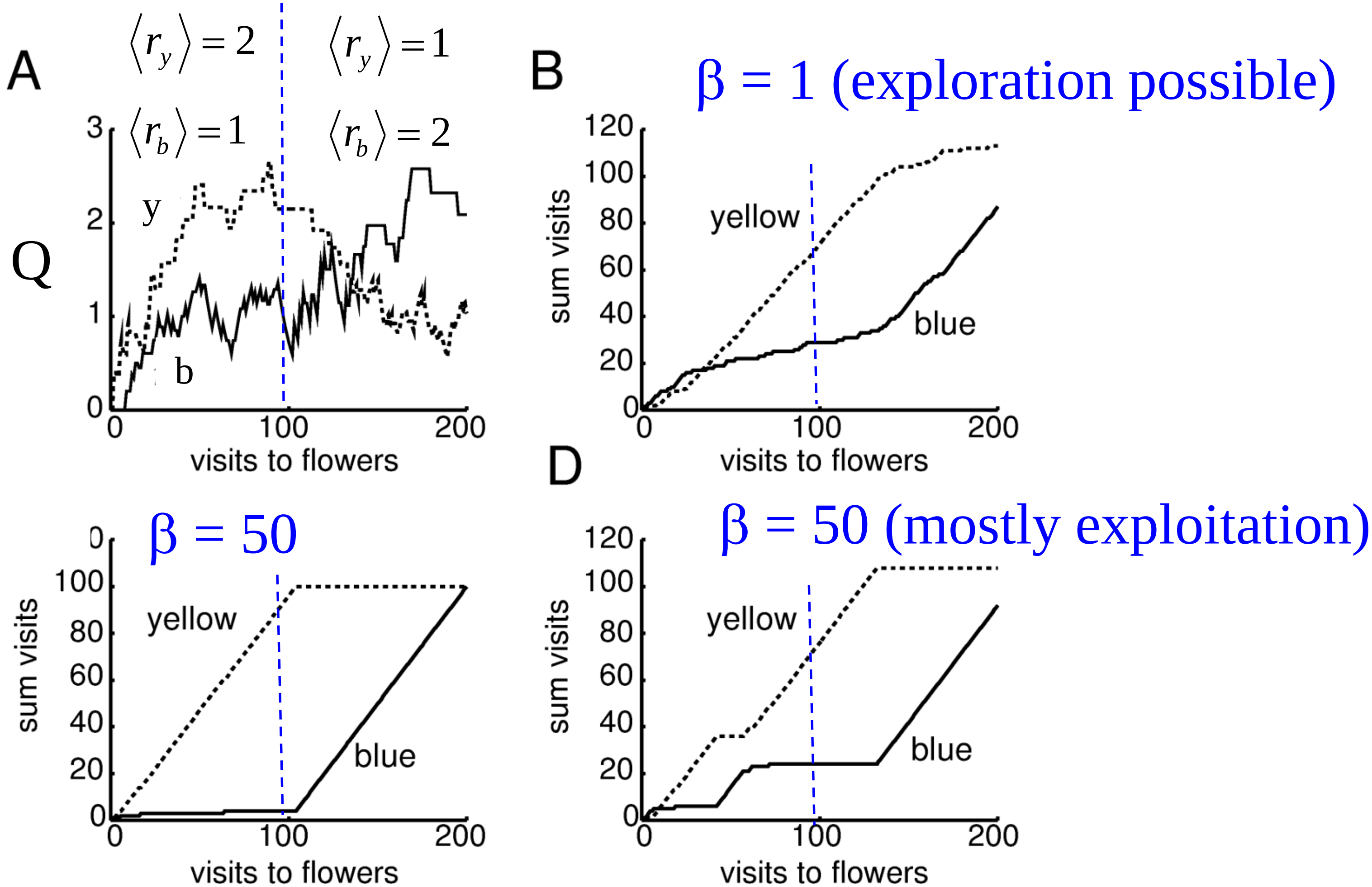
$$\left.\begin{aligned} Q(y) &\to Q(y) + \varepsilon(r_y - Q(y)) \\ Q(b) &\to Q(b) + \varepsilon(r_b - Q(b)) \end{aligned}\right\}$$

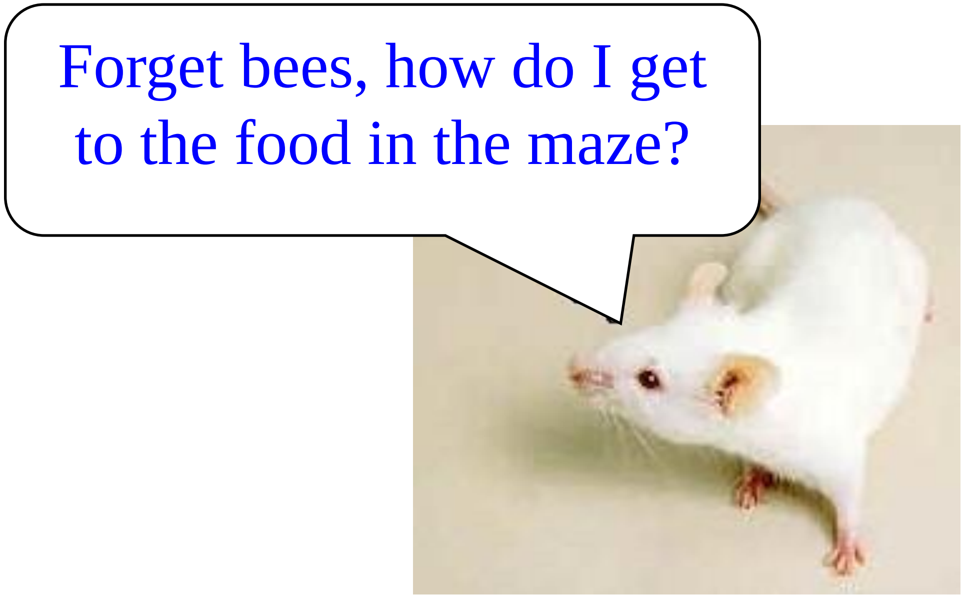
delta rule (running average)

$$P(y) = \frac{\exp(\beta Q(y))}{\exp(\beta Q(y)) + \exp(\beta Q(b))}$$

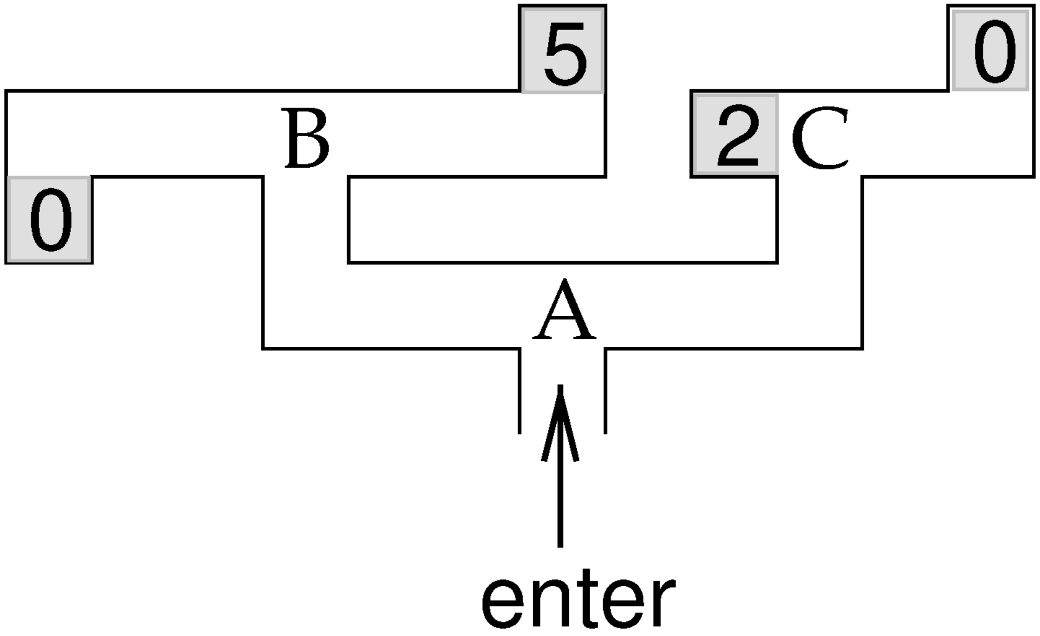
$$P(b) = 1 - P(y)$$

Simulating Bees




Forget bees, how do I get to the food in the maze?

Selecting Actions when Reward is Delayed

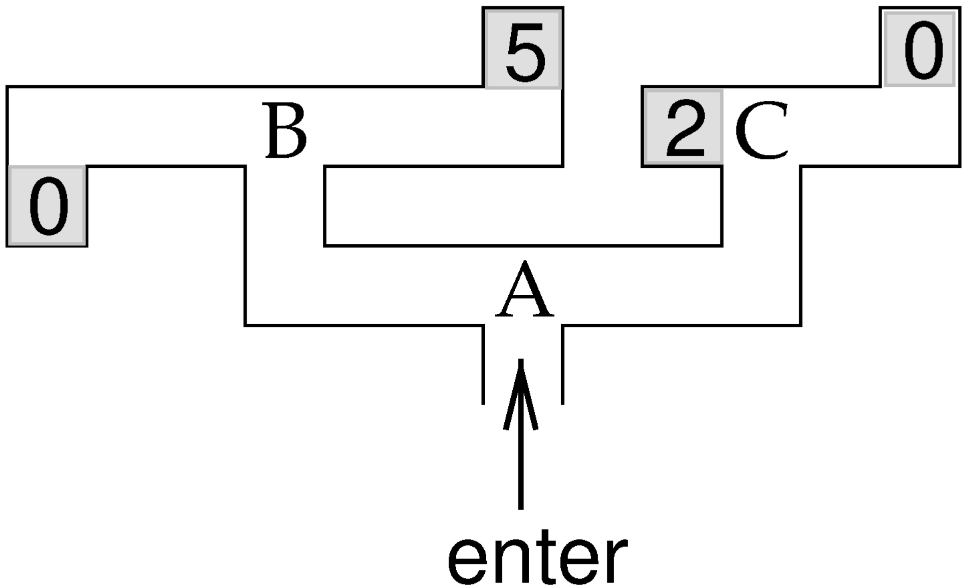


States: A, B, or C

Possible actions at any state: Left (L) or Right (R)

If you randomly choose to go L or R (random "policy"), what is the *value v of each state*?

Policy Evaluation



For random policy:

$$v(B) = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 5 = 2.5$$

$$v(C) = \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 0 = 1$$

$$v(A) = \frac{1}{2} \cdot v(B) + \frac{1}{2} \cdot v(C) = 1.75$$

(Location, action) ⇨ new location

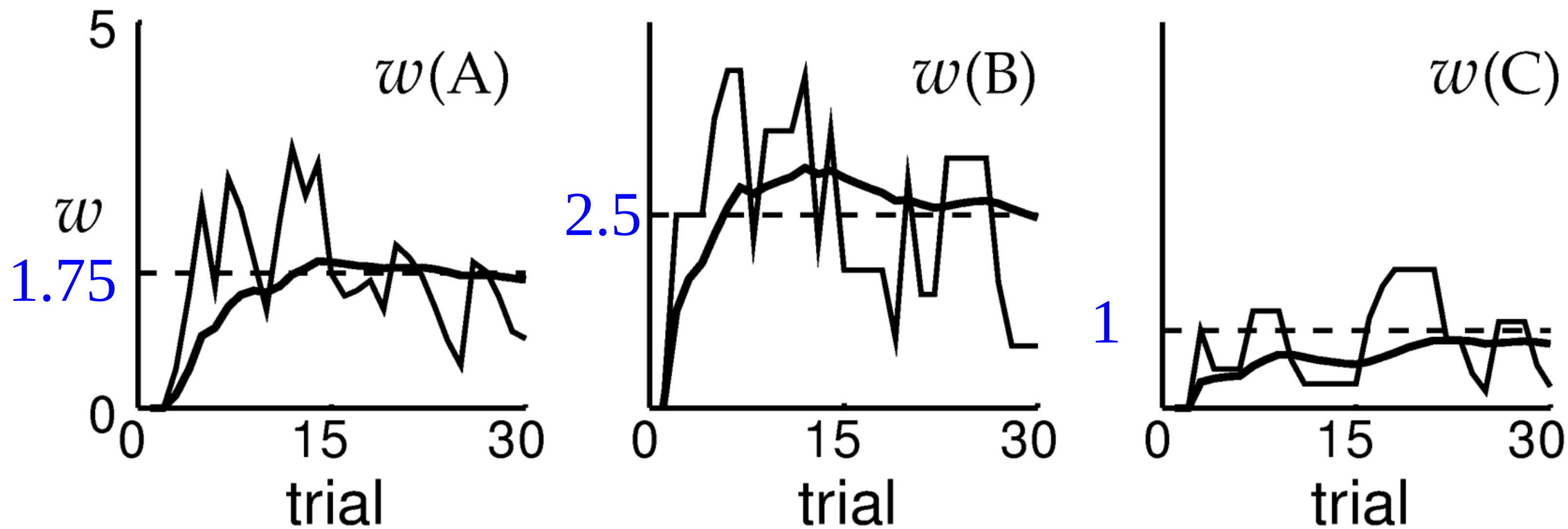
(u,a) ⇨ u'

Let $v(u) = w(u)$

Can learn this using TD learning:

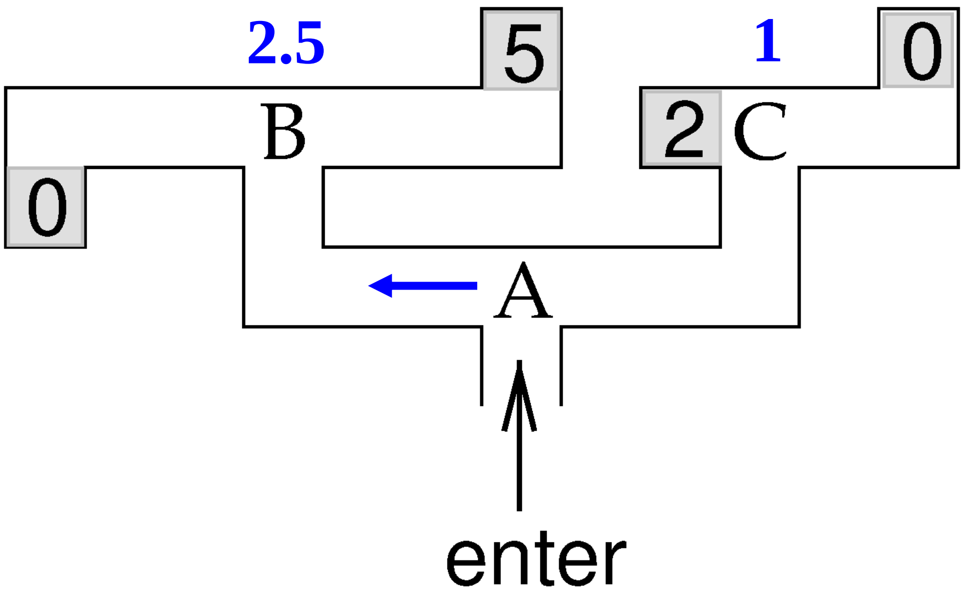
$$w(u) \rightarrow w(u) + \varepsilon [r_a(u) + v(u') - v(u)]$$

Maze Value Learning for Random Policy



Once I know the values, I can pick the action that leads to the higher valued state!

Selecting Actions based on Values



Values act as surrogate immediate rewards → Locally optimal choice leads to globally optimal policy (for Markov environments)

Related to *Dynamic Programming* in CS (see appendix in text)

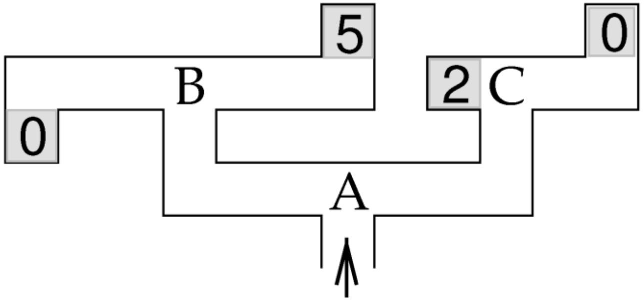
Actor-Critic Learning

- Two separate components: Actor (maintains policy) and Critic (maintains value of each state)

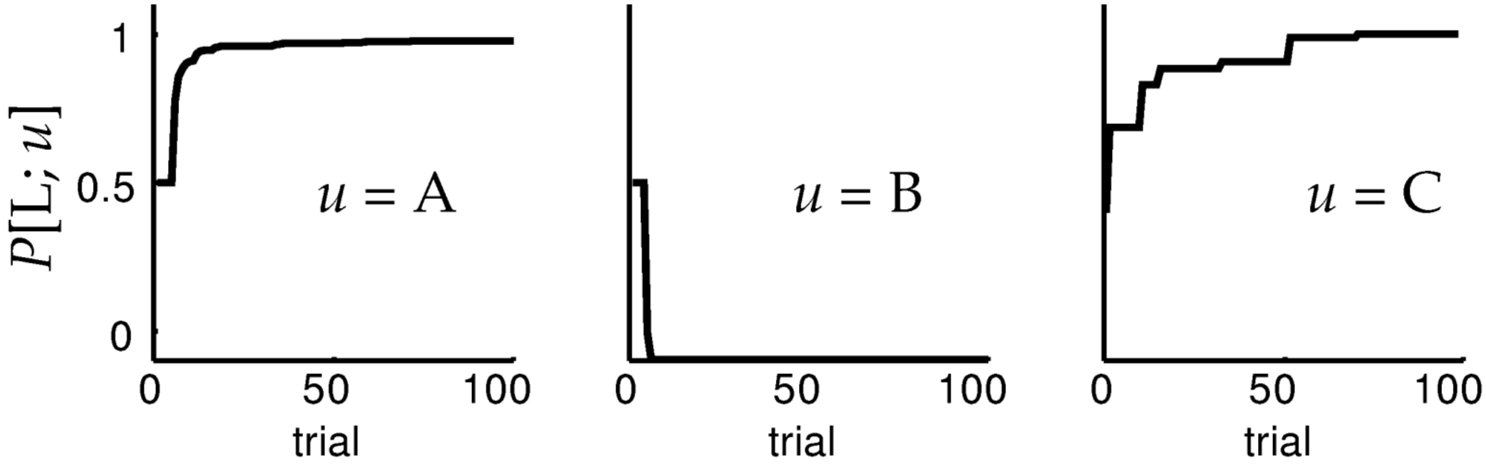
1. Critic Learning ("Policy Evaluation"):
   Value of state $u = v(u) = w(u)$
   $$w(u) \to w(u) + \varepsilon [r_a(u) + v(u') - v(u)]$$ (same as TD rule)
2. Actor Learning ("Policy Improvement"):
   $$P(a;u) = \frac{\exp(\beta Q_a(u))}{\sum_b \exp(\beta Q_b(u))}$$ Use this to select an action *a* in *u*

   For all *a*':
   $$Q_{a'}(u) \to Q_{a'}(u) + \varepsilon [r_a(u) + v(u') - v(u)](\delta_{aa'} - P(a';u))$$
3. Interleave 1 and 2

Actor-Critic Learning in the Maze Task

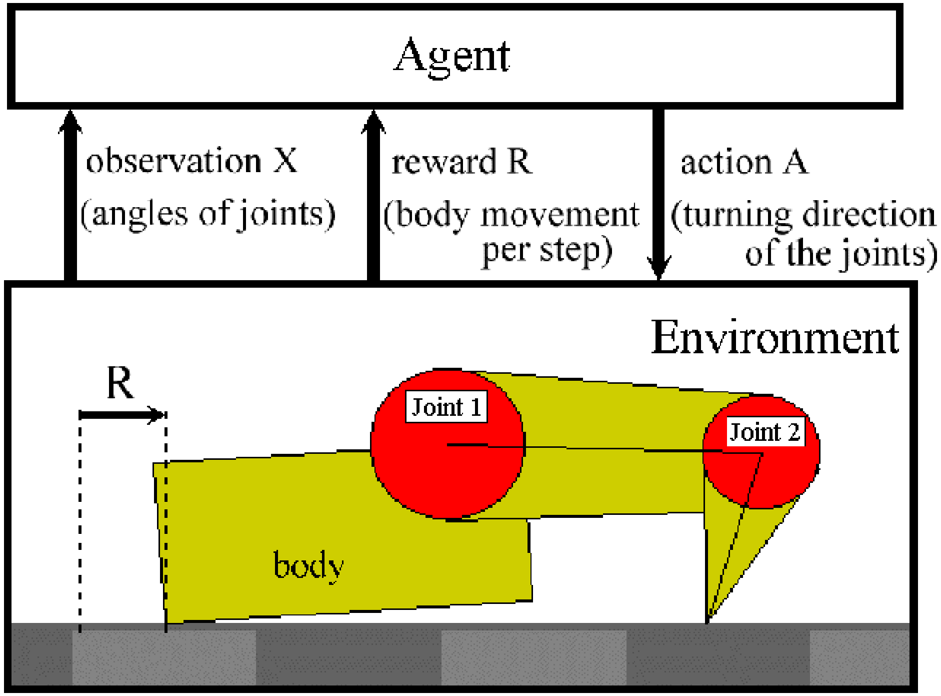


Probability of going Left at a location



Demo of Reinforcement Learning in a Robot

(from http://sysplan.nams.kyushu-u.ac.jp/gen/papers/JavaDemoML97/robodemo.html)



Things to do:

Finish homework 3
Work on group project

Next week: Prof. Emo Todorov on motor control