# Reinforcement Learning

Philipp Koehn

17 November 2015

# Rewards

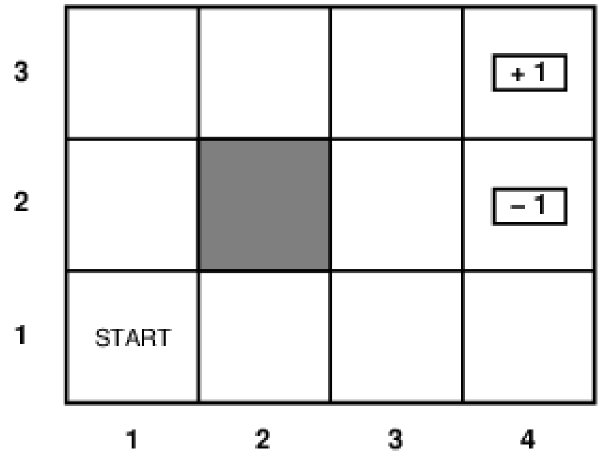


- Agent takes actions
- Agent occasionally receives reward
- Maybe just at the end of the process, e.g., Chess:
  - agent has to decide on individual moves
  - reward only at end: win/lose
- Maybe more frequently
  - ping pong: any point scored
  - learning to crawl: any forward movement

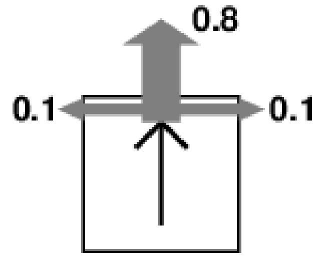
# Markov Decision Process



**State Map**




**Stochastic Movement**




- States $s \in S$, actions $a \in A$
- Model $T(s, a, s') \equiv P(s'|s, a)$ = probability that $a$ in $s$ leads to $s'$
- Reward function $R(s)$ (or $R(s, a)$, $R(s, a, s')$)

$$= \begin{cases} -0.04 & \text{(small penalty) for nonterminal states} \\ \pm 1 & \text{for terminal states} \end{cases}$$

# Agent Designs

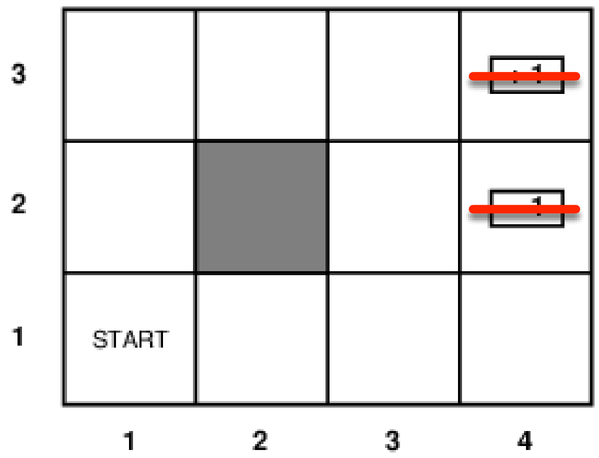


- Utility based agent
  - needs model of environment
  - learns utility function on states
  - selects action that maximize expected outcome utility
- Q-learning
  - learns action-utility function ($Q(s, a)$ function)
  - does not need to model outcomes of actions
  - function provides expected utility of taken a given action at a given step
- Reflex agent
  - learns policy that maps states to actions

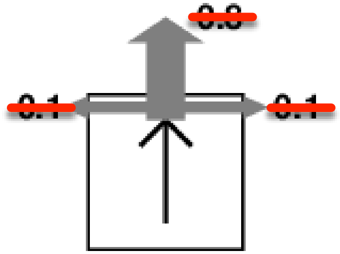
# passive reinforcement learning

# Setup

**State Map**




**Stochastic Movement**



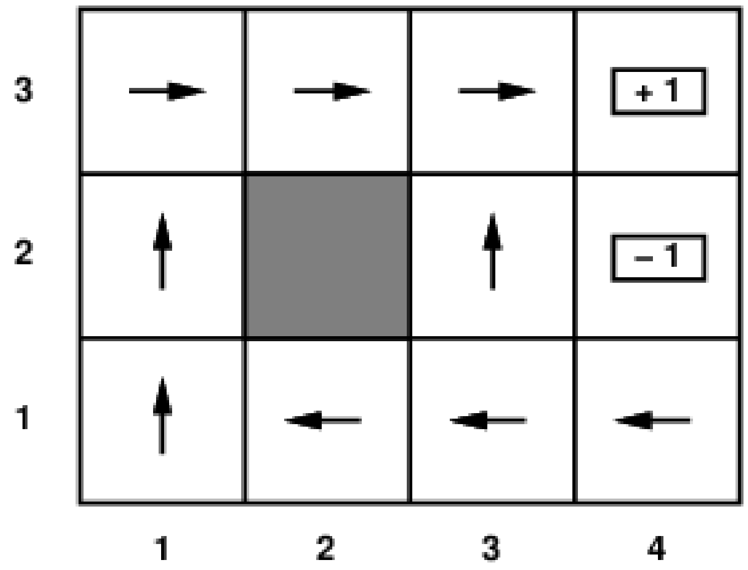
**Reward Function**

$$R(s) = \begin{cases} +1 & \text{for goal} \\ -1 & \text{for pit} \\ -0.04 & \text{for other} \end{cases}$$

- We know which state we are in (= partially observable environment)
- We know which actions we can take
- But only after taking an action
  → new state becomes known
  → reward becomes known

# Passive Reinforcement Learning

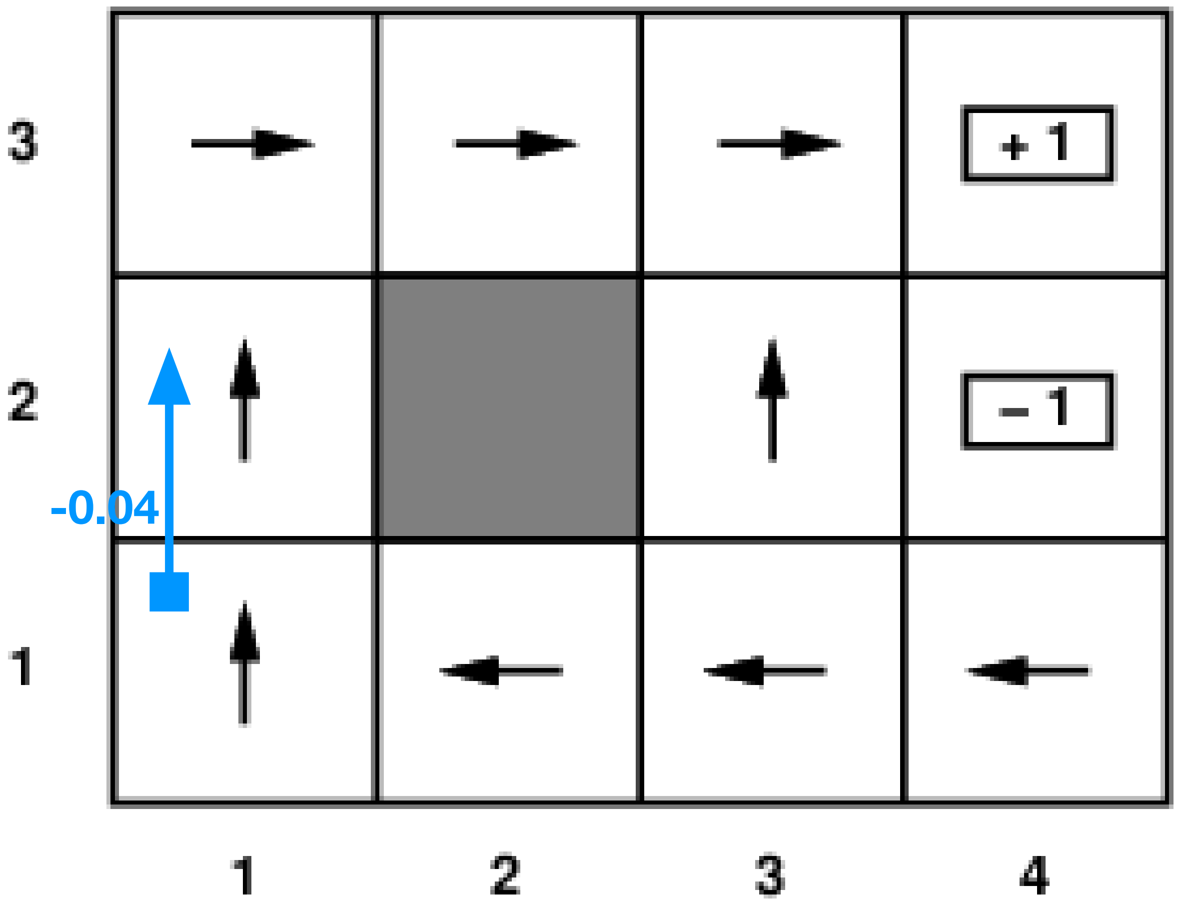
- Given a policy




- Task: compute utility of policy

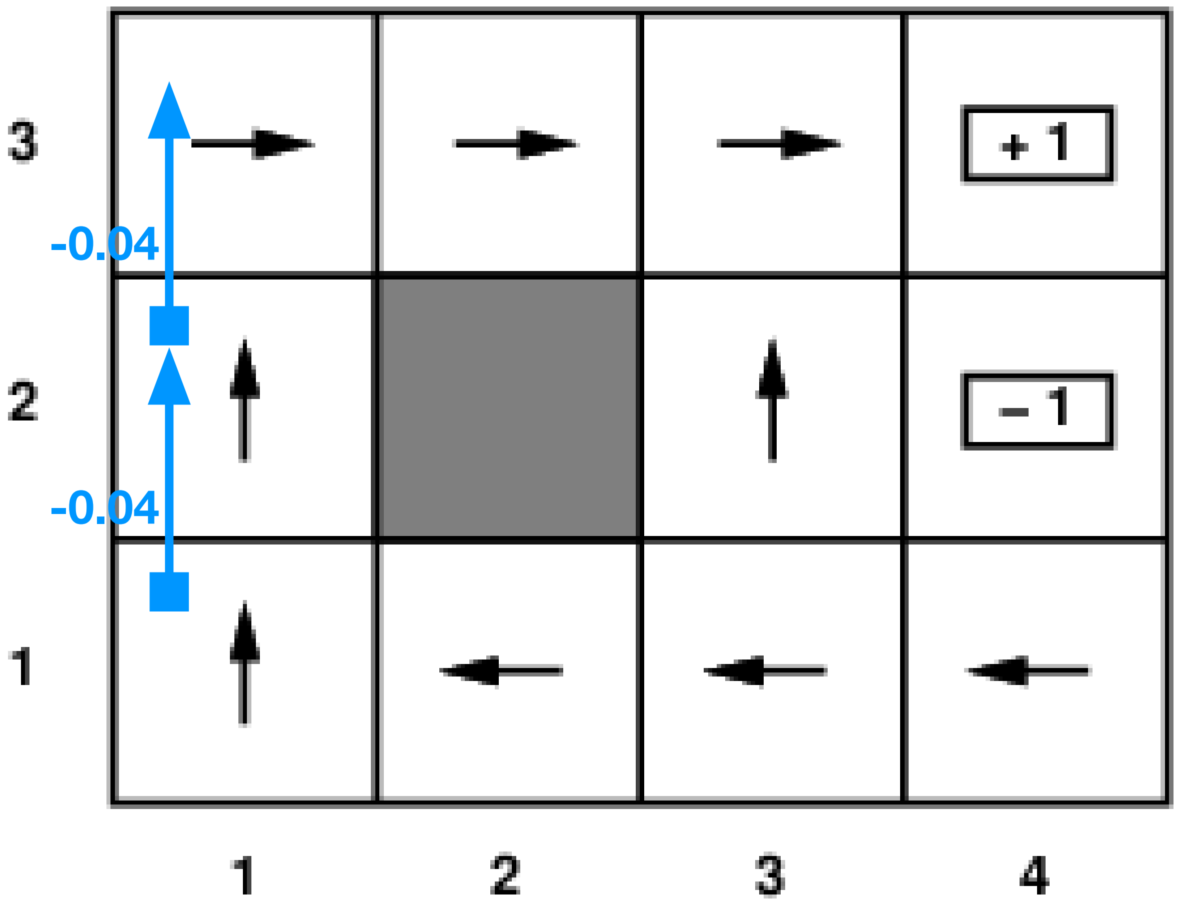
- We will extend this later to **active** reinforcement learning
  (⇒ policy needs to be learned)

# Sampling



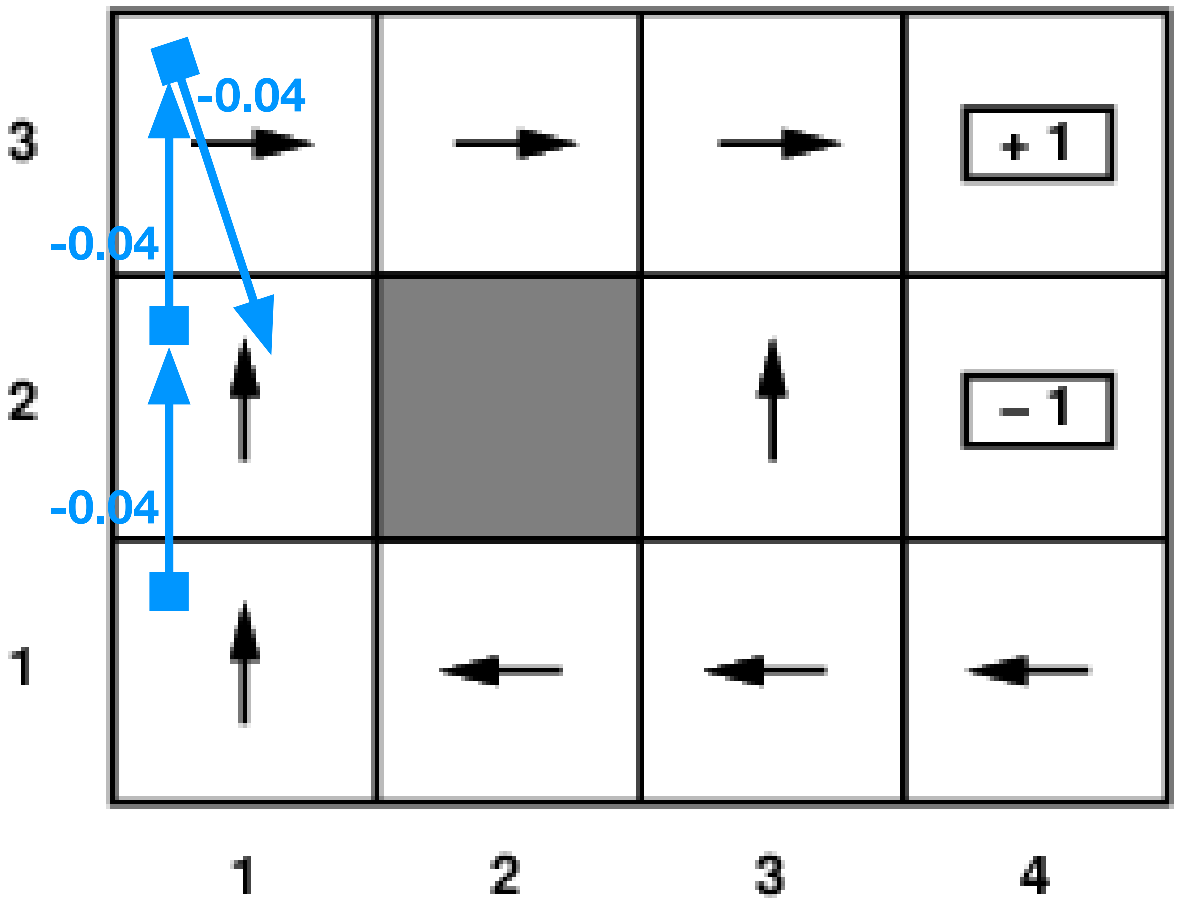
# Sampling



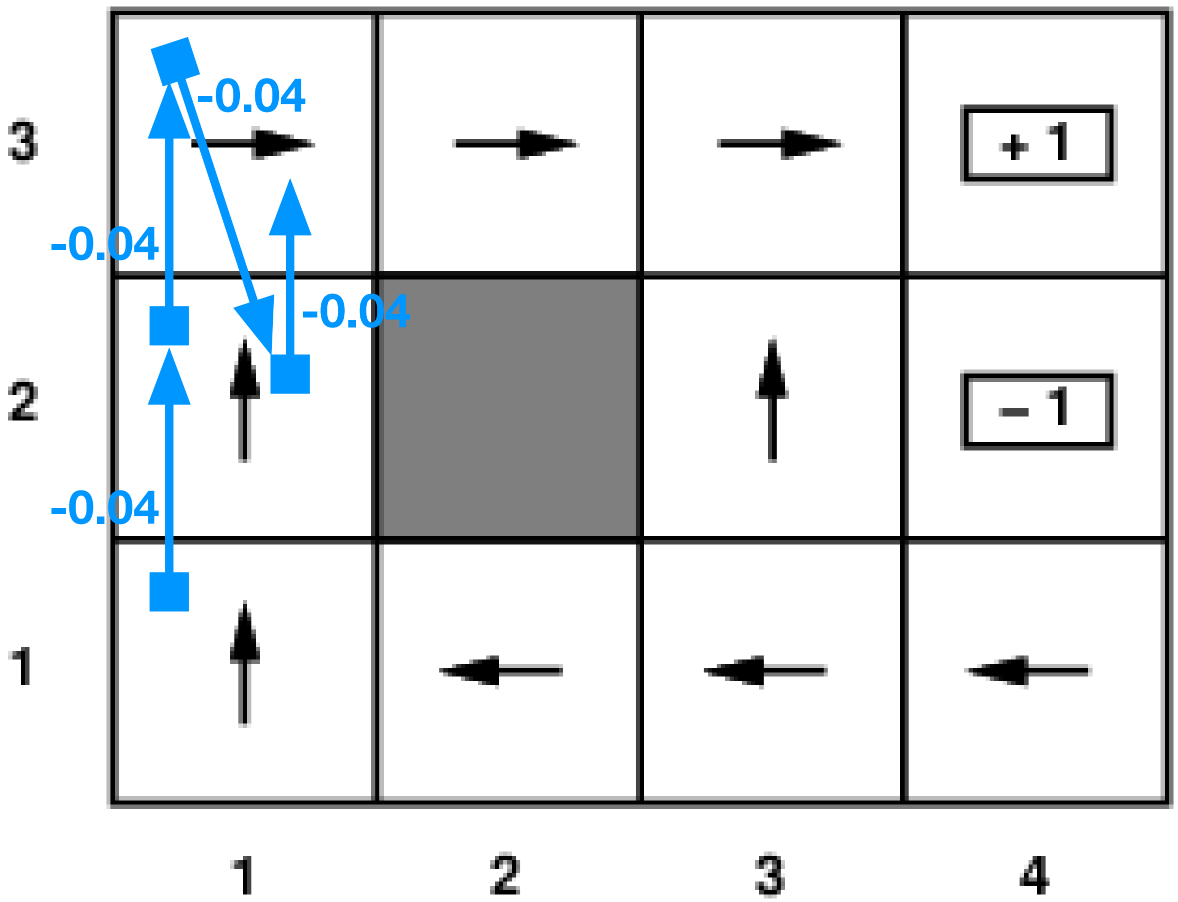


# Sampling



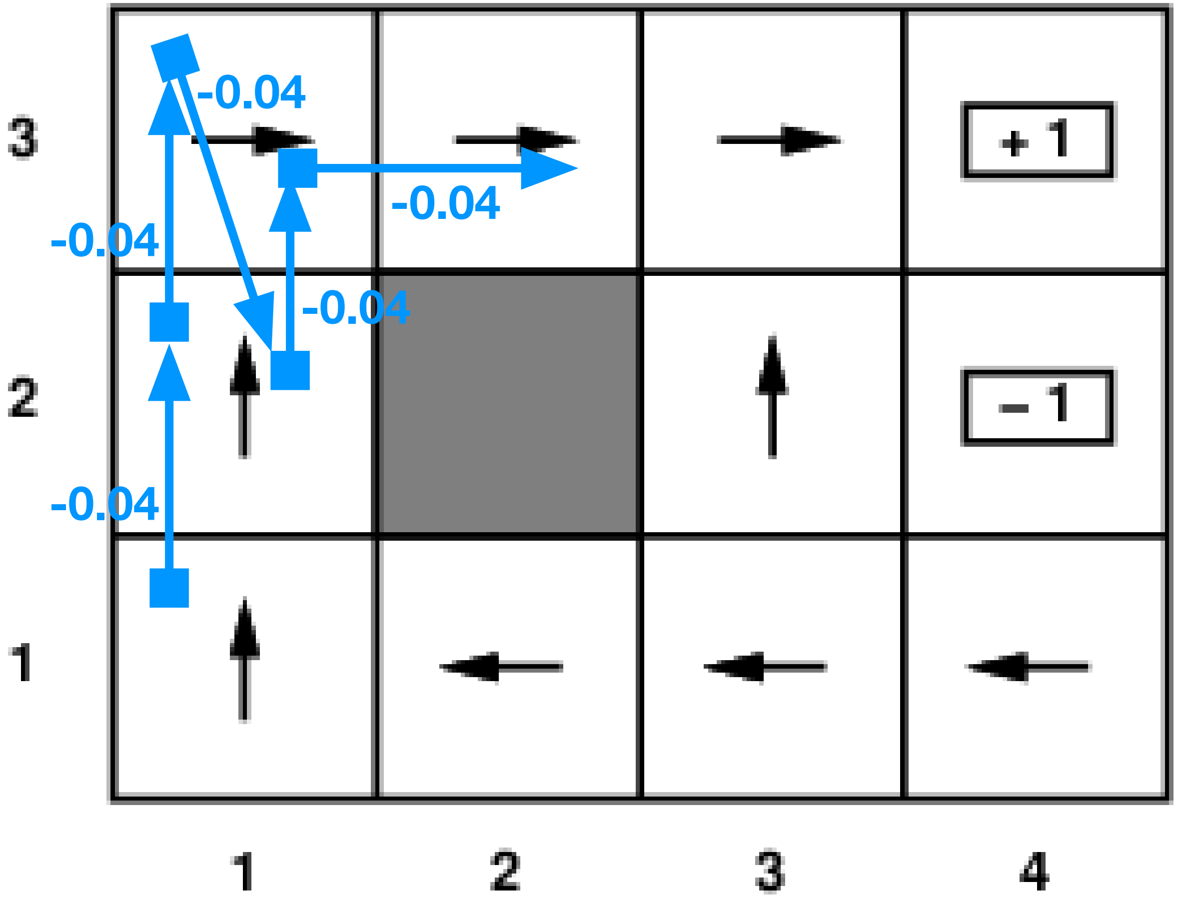


# Sampling



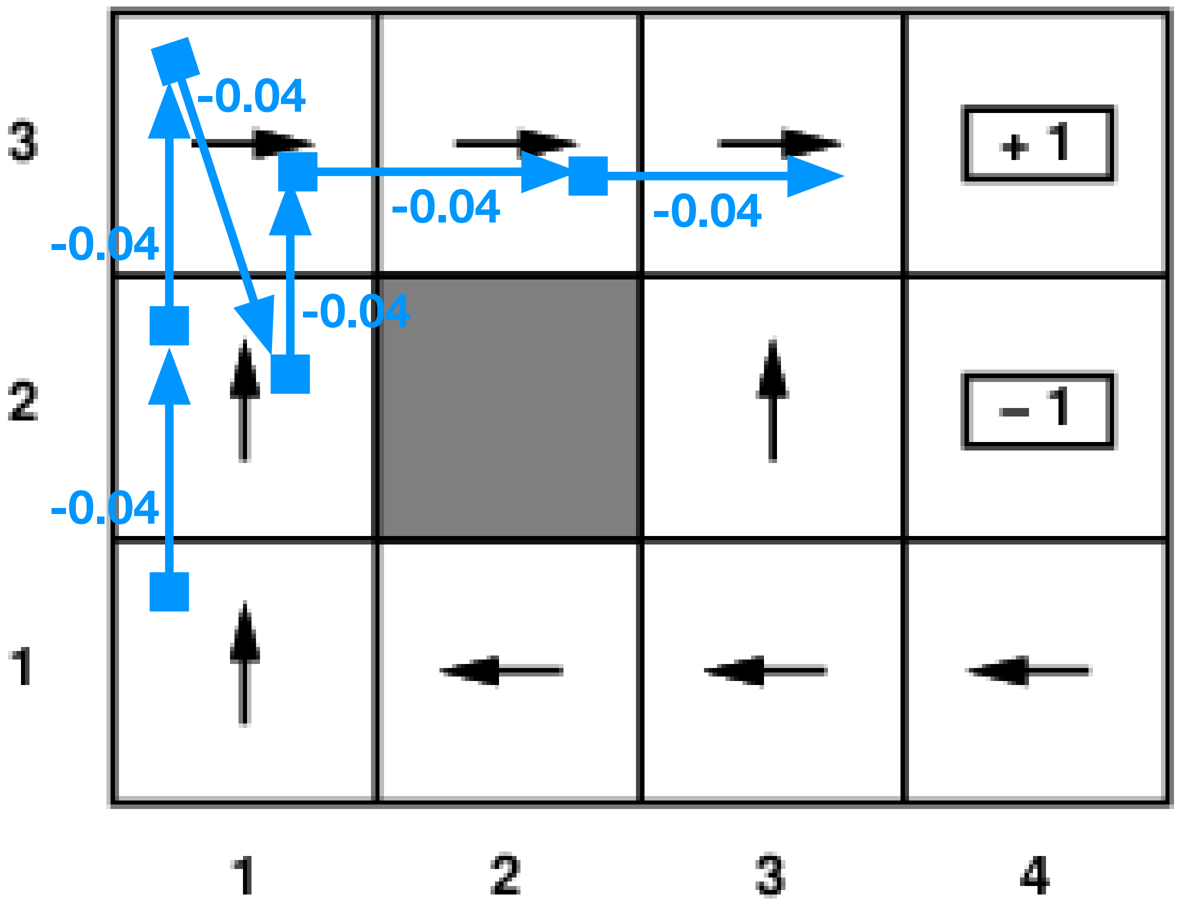
# Sampling



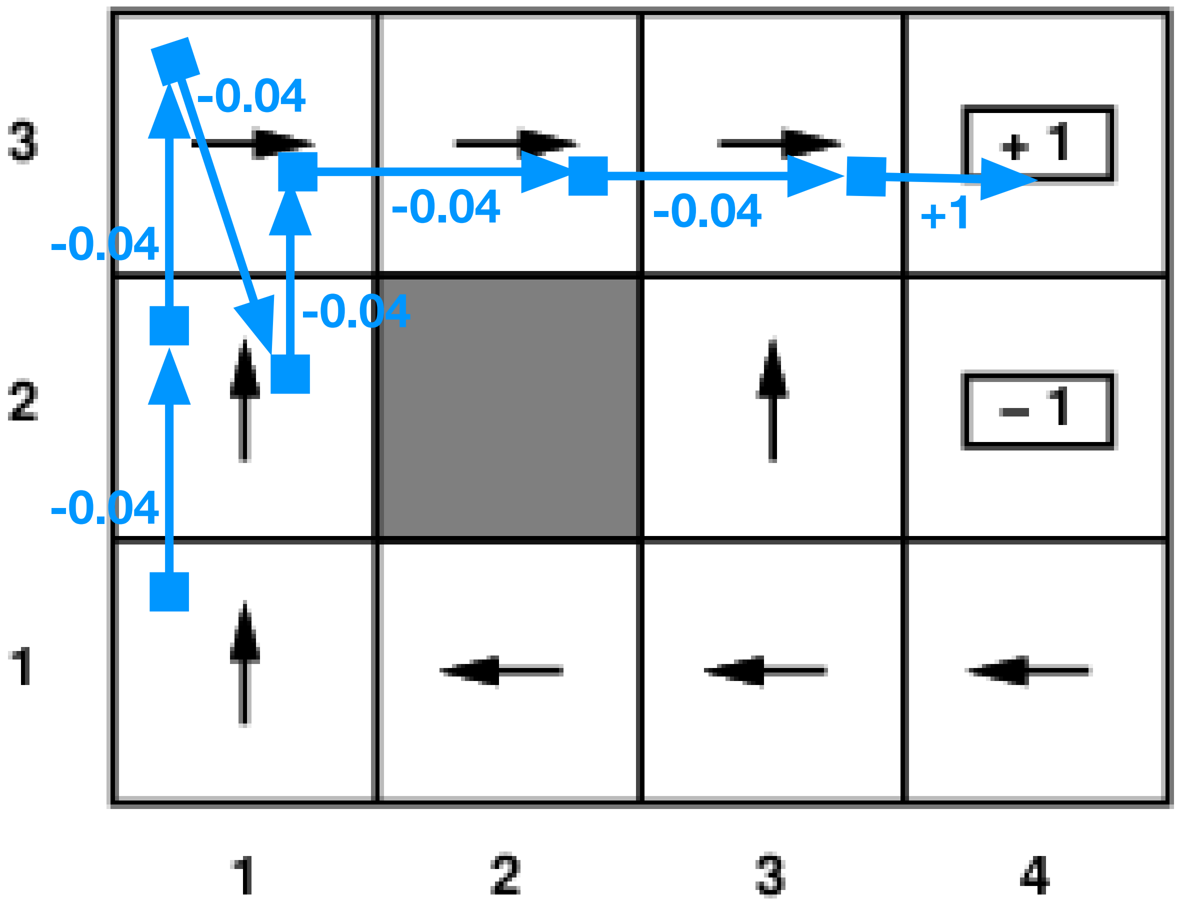


# Sampling

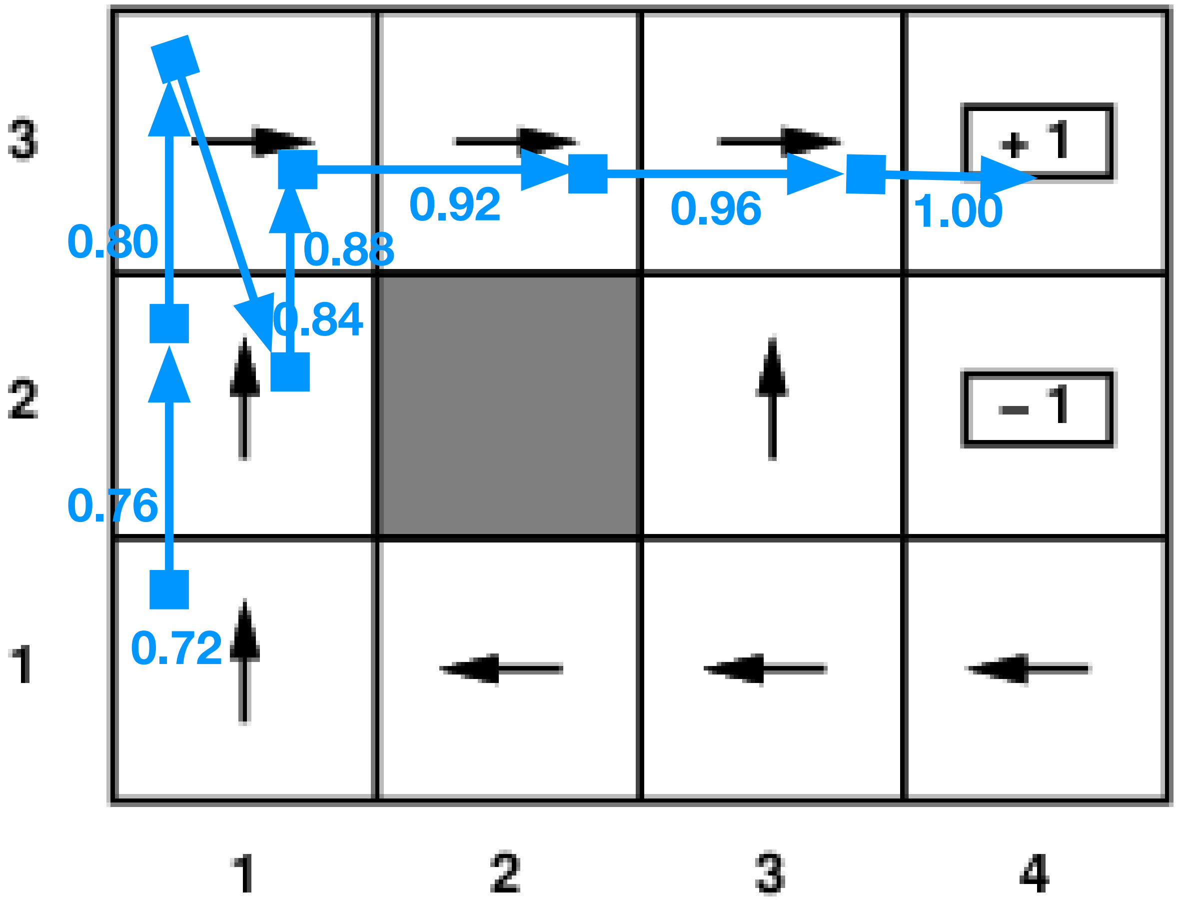




# Sampling

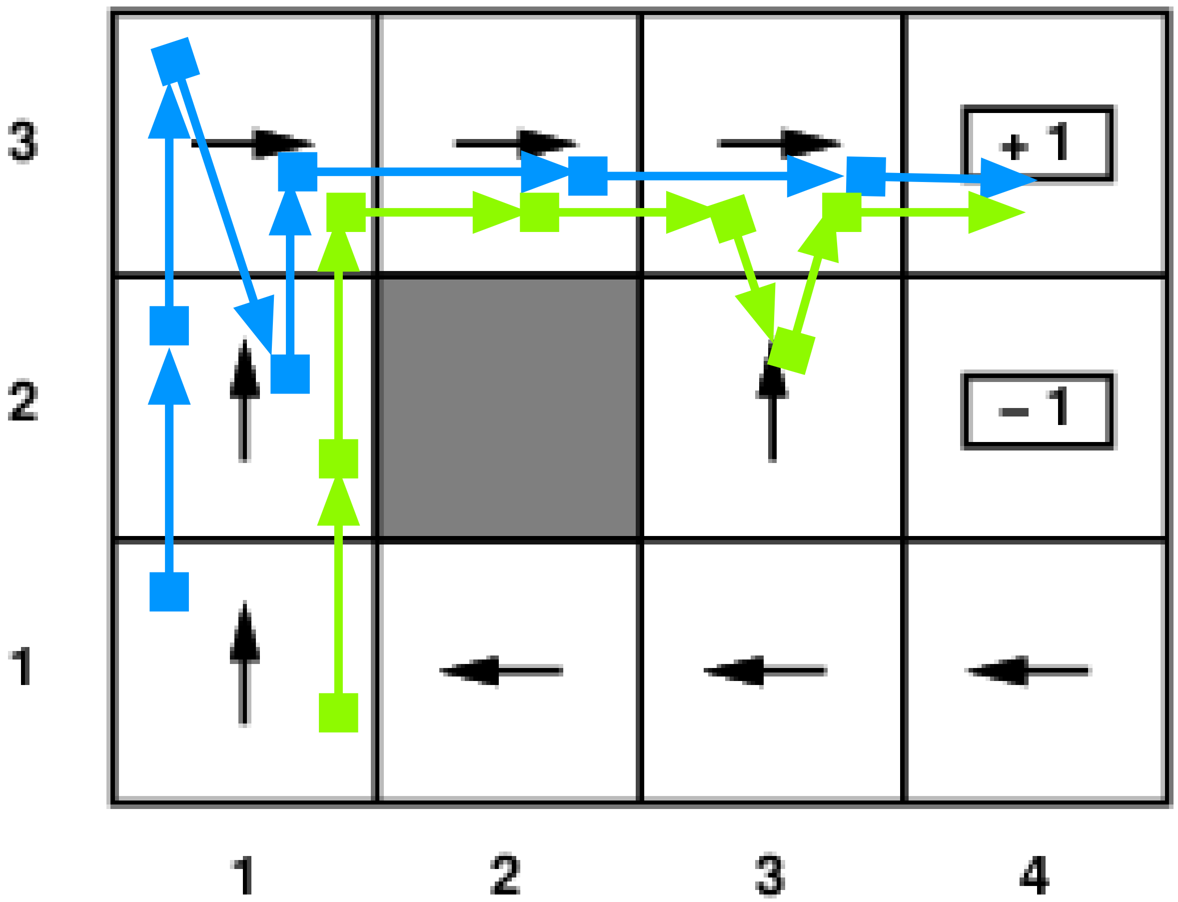


# Sampling



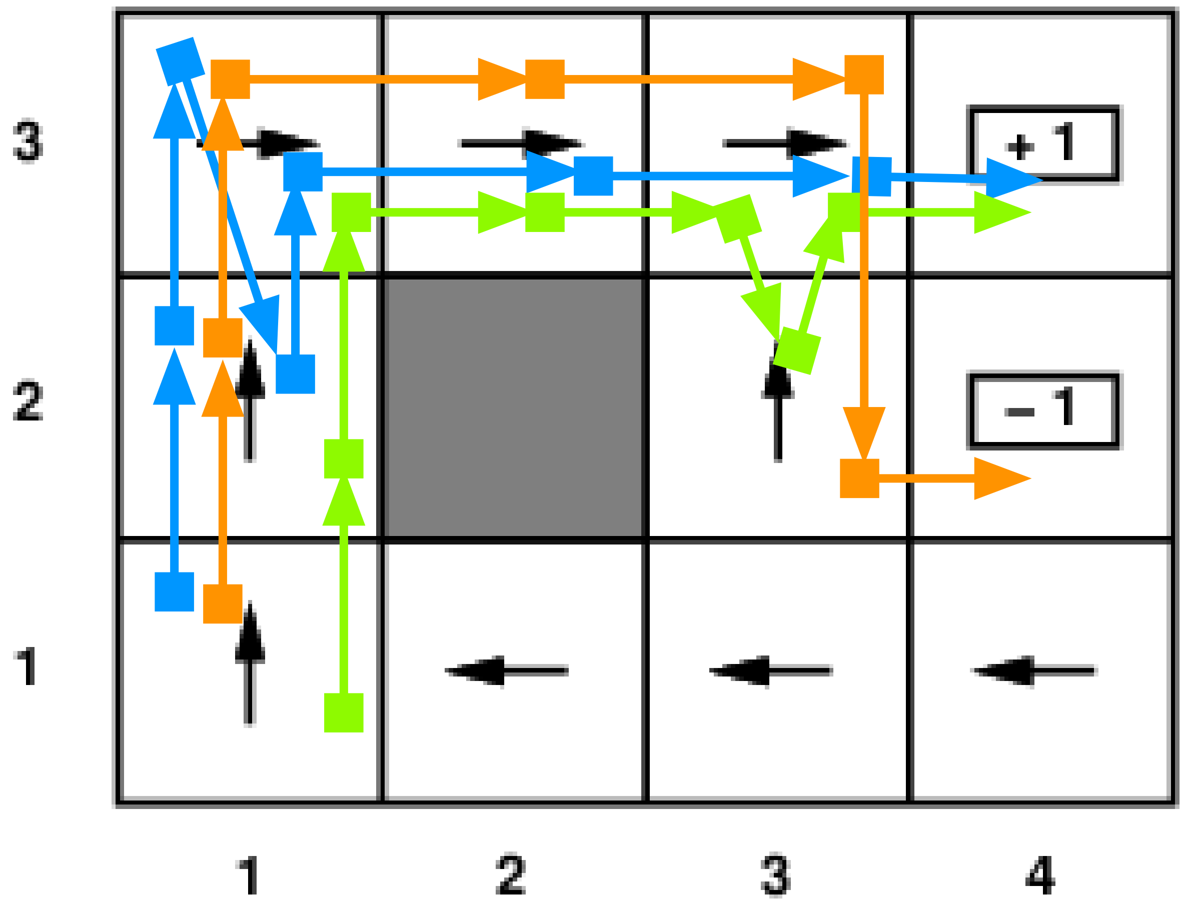

- Sample of reward to go

# Sampling



# Sampling





# Utility of Policy

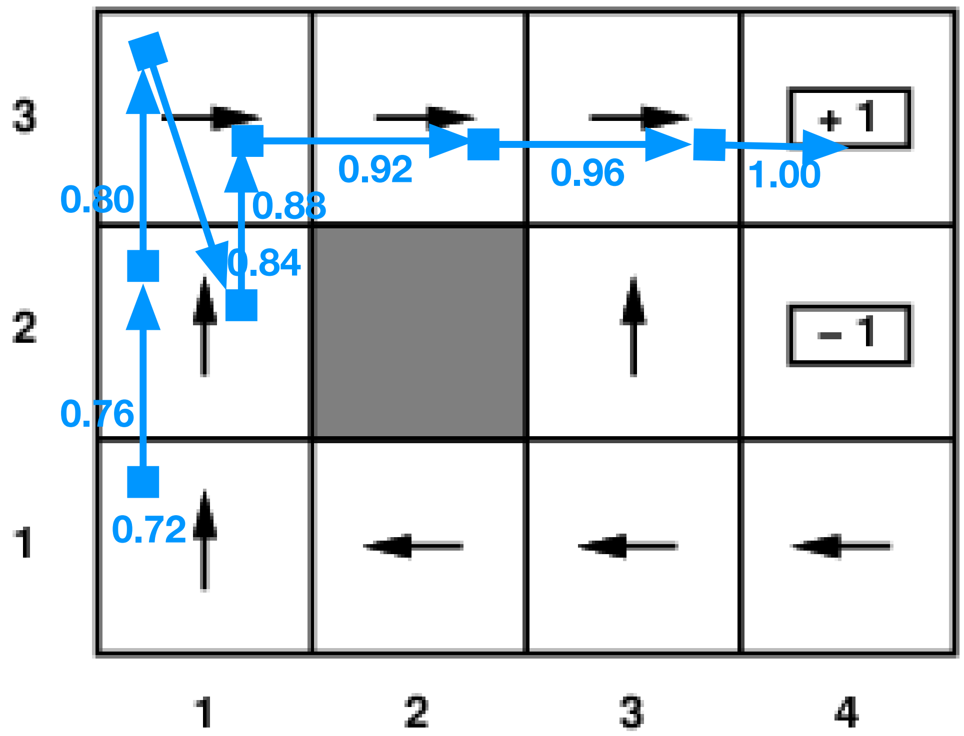
- Definition of utility $U$ of the policy $\pi$ for state $s$

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

- Start at state $S_0 = s$
- Reward for state is $R(s)$
- Discount factor $\gamma$ (we use $\gamma = 1$ in our examples)

# Direct Utility Estimation

- Learning from the samples
- Reward to go:
  - (1,1) one sample: 0.72
  - (1,2) two samples: 0.76, 0.84
  - (1,3) two samples: 0.80, 0.88
- Reward to go
  will converge to utility of state
- But very slowly — can we do better?



# Bellman Equation

- Direct utility estimation ignores dependency between states
- Given by Bellman equation

$$U^\pi(s) = R(s) + \gamma \sum_{s'} P(s'|s,\pi(s))\, U^{\pi(s')}$$

($\gamma$ = reward decay)

- Use of this known dependence can speed up learning
- Requires learning of transition probabilities $P(s'|s,\pi(s))$

# Adaptive Dynamic Programming



Need to learn:

- State rewards $R(s)$
  - whenever a state is visited, record award (deterministic)
- Outcome of action $\pi(s)$ at state $s$ according to policy $\pi$
  - collect statistic $\text{count}(s, s')$ that $s'$ is reached from $s$
  - estimate probability distribution

$$P(s'|s, \pi(s)) = \frac{\text{count}(s, s')}{\sum_{s''} \text{count}(s, s'')}$$

⇒ Ingredients for policy evaluation algorithm

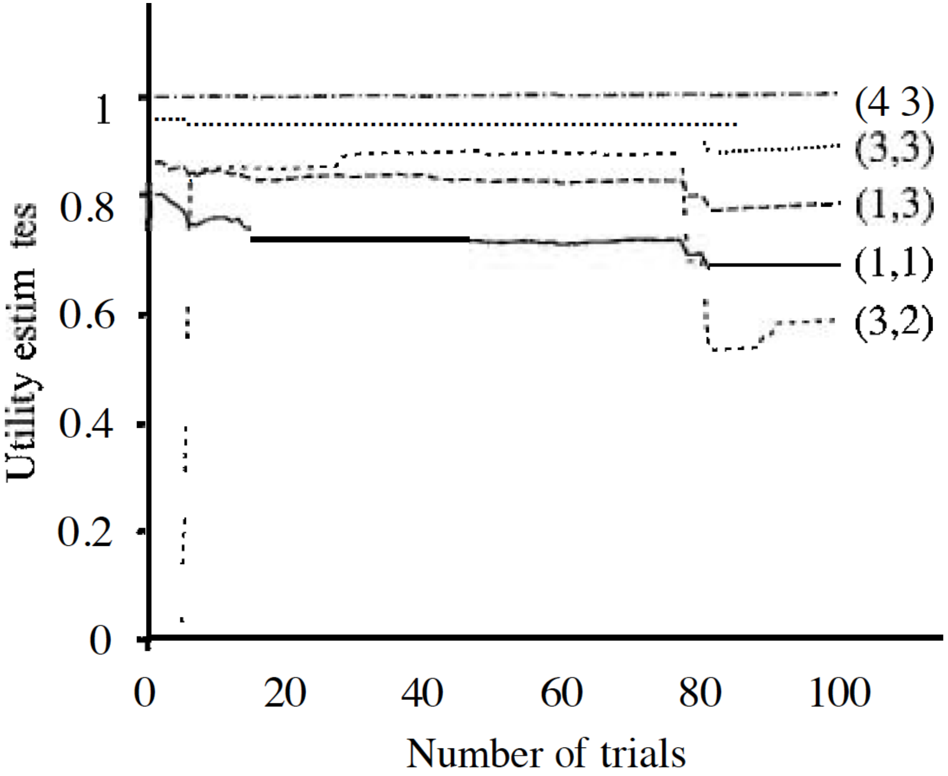
# Adaptive Dynamic Programming

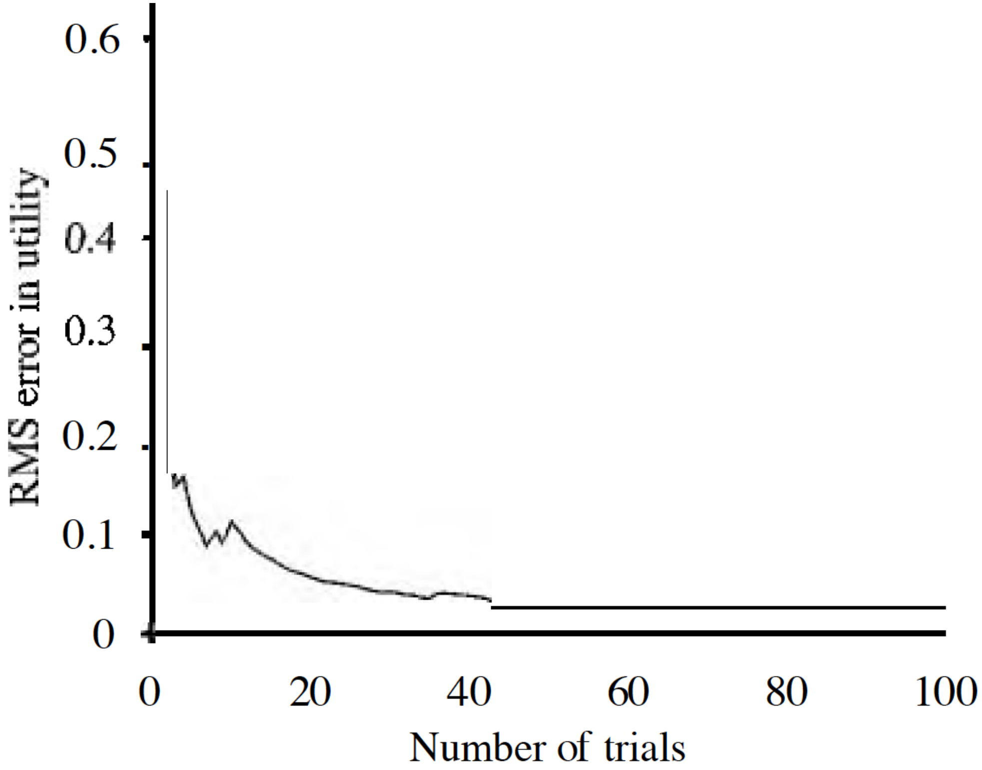


**function** PASSIVE-ADP-AGENT(*percept*) **returns** an action
  **inputs:** percept, a percept indicating the current state $s'$ and reward signal $r'$
  **static:** $\pi$, a fixed policy
    mdp, an MDP with model T, rewards R, discount $\gamma$
    $U$, a table of utilities, initially empty
    $N_{sa}$, a table of frequencies for state-action pairs, initially zero
    $N_{sas'}$, a table of frequencies for state-action-state triples, initially zero
    $s$, a, the previous state and action, initially null

  **if** $s$ is new **then do** $U[s] \leftarrow r$ ; $R[s] \leftarrow r'$
  **if** $s$ is not null **then do**
    increment $N_{sa}[s, \mathbf{a}]$ and $N_{sas'}[s, a, s]$
    **for each** $t$ such that $N_{sas'}[s, a, t]$ is nonzero **do**
      $T[s, a, t] \leftarrow N_{sas'}[s, a, t] / N_{sa}[s, a]$
  $U \leftarrow$ *POLICY-EVALUATION*(^, $U$, mdp)
  **if** TERMINAL?[$s'$] **then** $s, a \leftarrow$ null **else** $s$, a $\leftarrow s$ , $\pi[s']$
  **return** $a$

# Learning Curve




- Major change at 78th trial: first time terminated in –1 state at (4,2)

# Temporal Difference Learning



- Idea: no model $P(s'|s,\pi(s))$, directly adjust utilities $U(s)$ for all visited states
- Current model expects utility of current state as $R(s) + \gamma U^{\pi}(s')$
- Actually current utility: $U^{\pi}(s)$
- Adjust utility of current state $U^{\pi}(s)$ if they differ

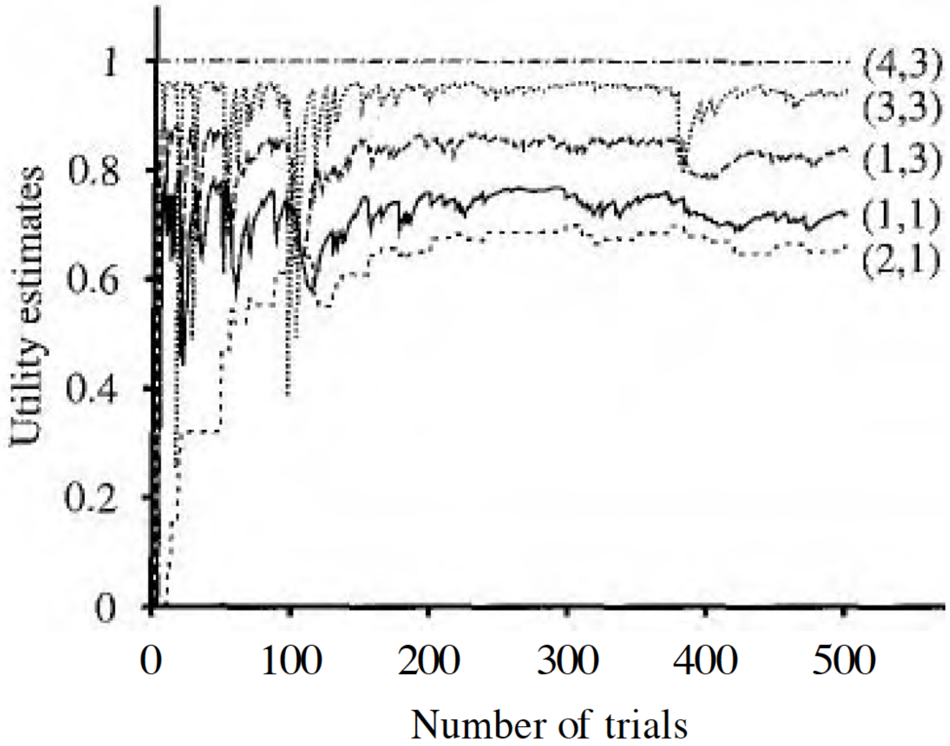
$$\Delta U^{\pi}(s) = \alpha \; ( \; R(s) + \gamma U^{\pi}(s') - U^{\pi}(s))$$

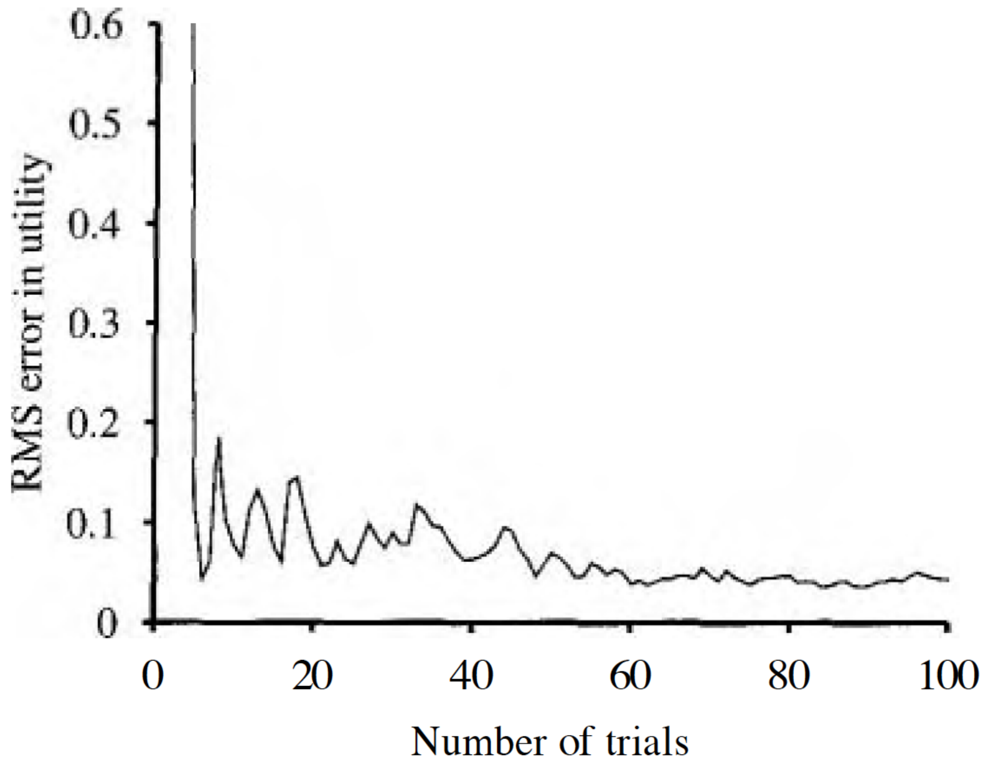
($\alpha$ = learning rate)

- Learning rate may decrease when state has been visited often

# Learning Curve






- Noisier, converging more slowly

# Comparison



- Both eventually converge to correct values
- Adaptive dynamic programming (ADP)
  faster than
  temporal difference learning (TD)
  - both make adjustments to make successors agree
  - but: ADP adjusts all possible successors, TD only observed successor
- ADP computationally more expensive due to policy evaluation algorithm

# active reinforcement learning

# Active Reinforcement Learning



- Passive agent follows prescribed policy

- Active agent decides which action to take
  - following optimal policy (as currently viewed)
  - exploration

- Goal: optimize rewards for a given time frame

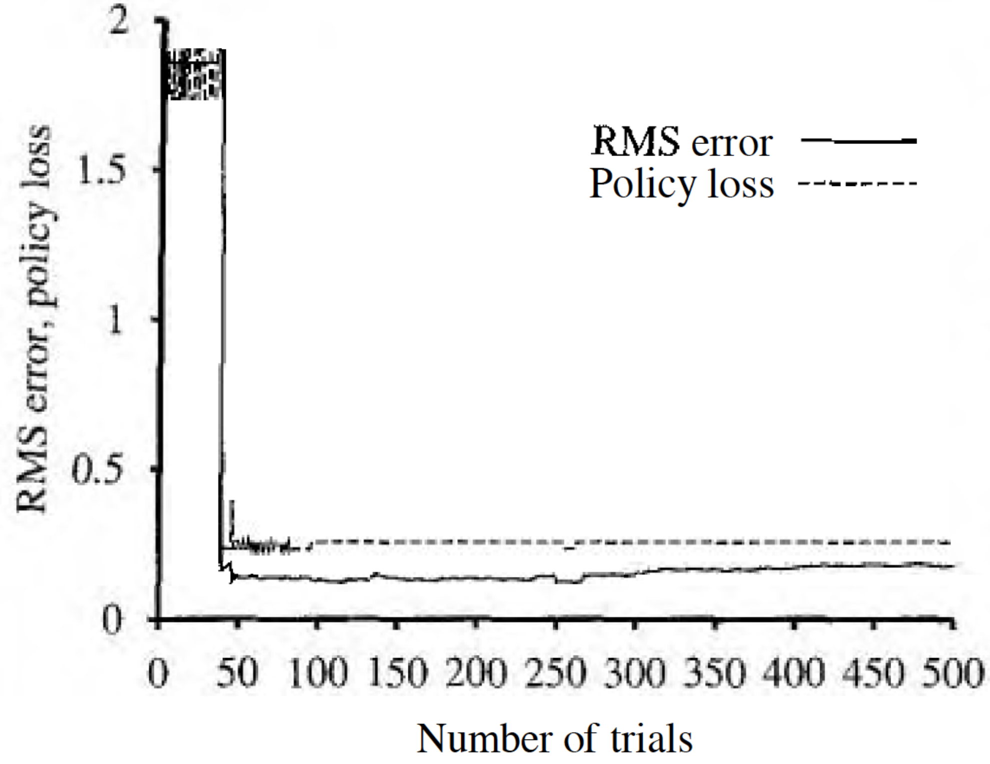
# Greedy Agent

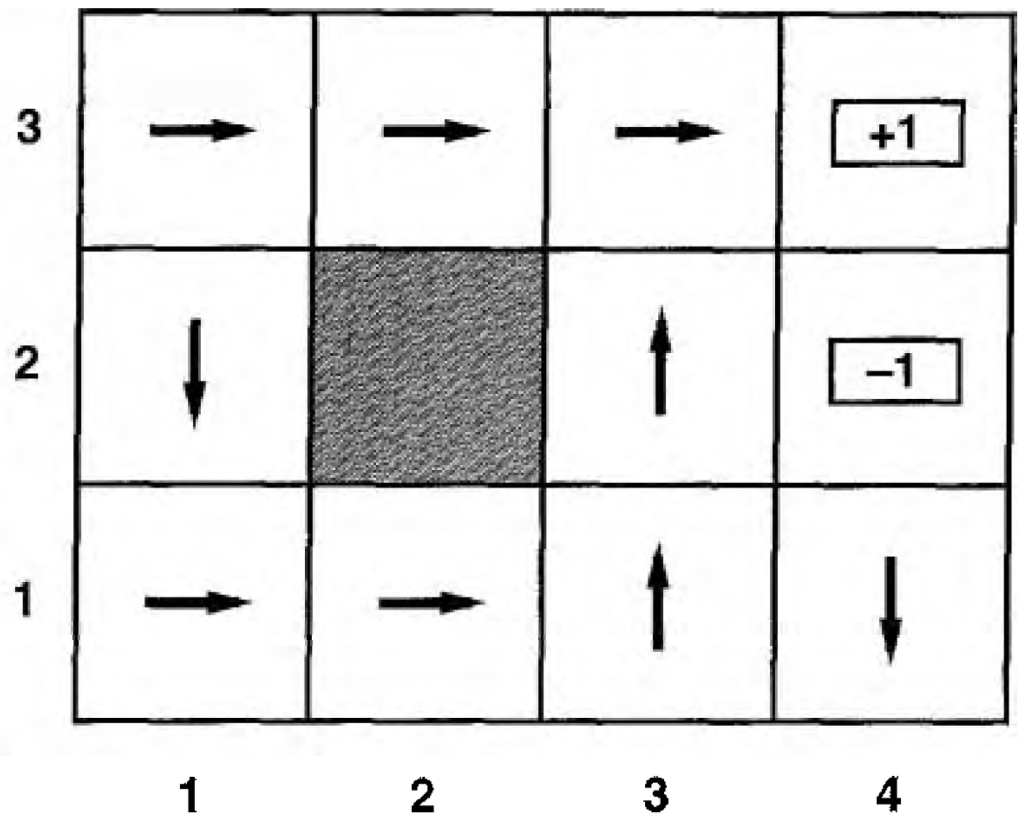


1. Start with initial policy
2. Compute utilities (using ADP)
3. Optimize policy
4. Go to Step 2

- This *very seldom* converges to global optimal policy

# Learning Curve




- Greedy agent stuck in local optimum

# Bandit Problems

- Bandit: slot machine
- N-armed bandit: $n$ levers
- Each has different probability distribution over payoffs
- Spend coin on
  - presume optimal payoff
  - exploration (new lever)
- If independent
  - **Gittins index**: formula for solution
  - uses payoff / number of times used

# Greedy in the Limit of Infinite Exploration



- Explore any action in any state unbounded number of times

- Eventually has to become greedy
  - carry out optimal policy

  ⇒ maximize reward

- Simple strategy
  - with probability $p(1/t)$ take random action
  - initially ($t$ small) focus on exploration
  - later ($t$ big) focus on optimal policy

# Extension of Adaptive Dynamic Programming



- Previous definition of utility calculation

$$U(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)\, U(s')$$

- New utility calculation

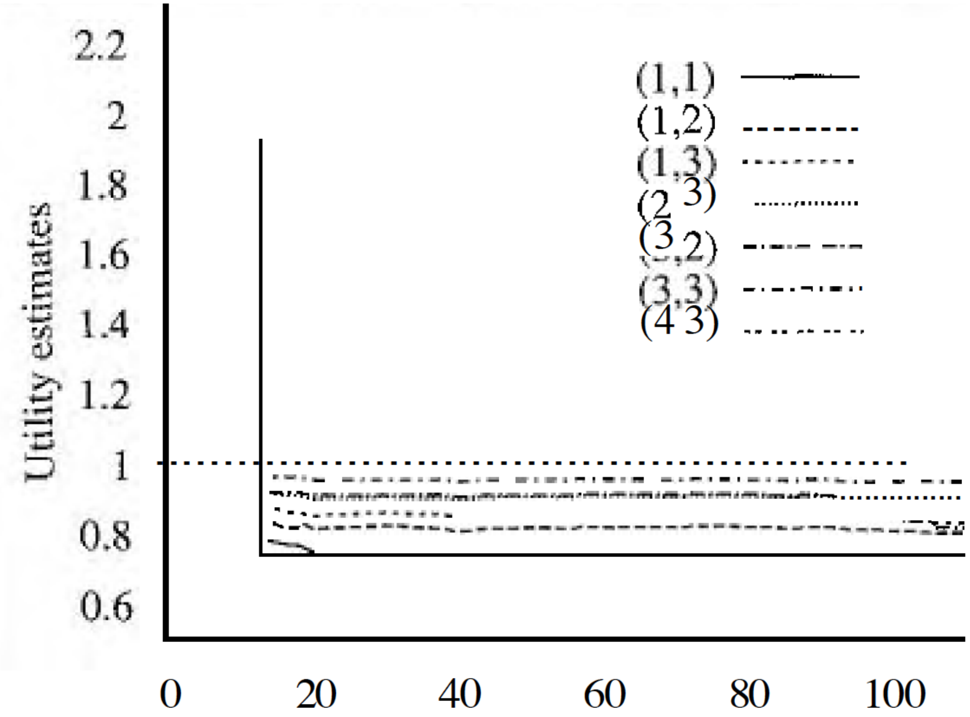
$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s'|s,a)\, U^+(s'), N(s,a)\right)$$

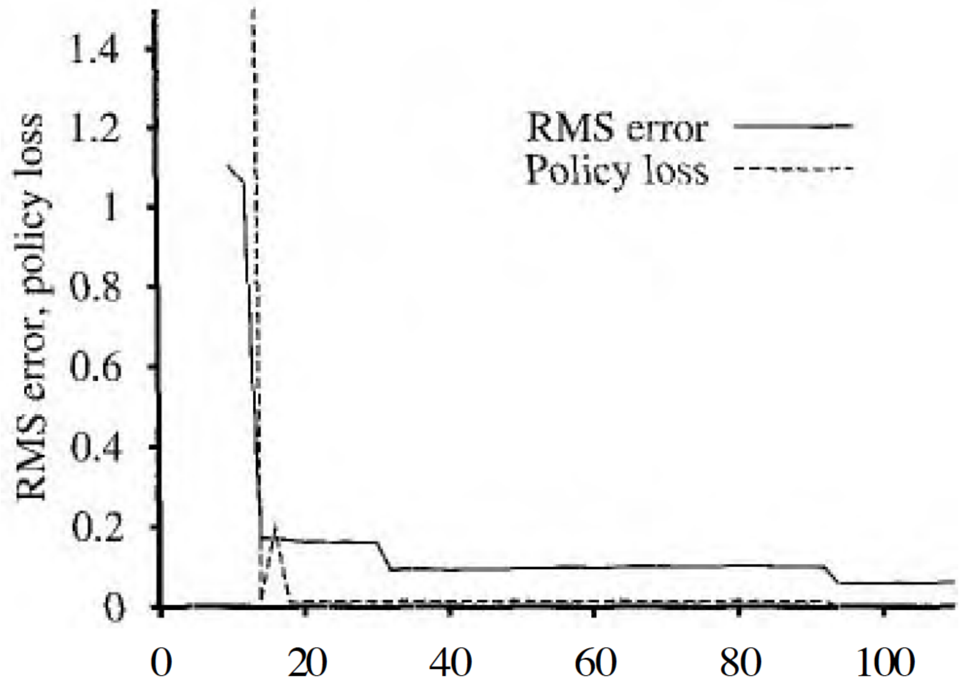
- One possible definition of $f(u,n)$

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_c \\ u & \text{otherwise} \end{cases}$$

$R^+$ is optimistic estimate, best possible award in any state

# Learning Curve




- Performance of exploratory ADP agent
- Parameter settings $R^{+} = 2$ and $N_e = 5$
- Fairly quick convergence to optimal policy

# Q-Learning



- Learning an action utility function $Q(s,a)$

- Allows computation of utilities $U(s) = \max_a Q(s,a)$

- Model-free: no explicit transition model $P(s'|s,a)$

- Theoretically correct Q values

$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a')$$

- Update formula inspired by temporal difference learning

$$\Delta Q(s,a) = \alpha(R(s) + \gamma \max_{a'} Q(s'a') - Q(s,a))$$

- For our example, Q-learning slower, but successful applications (TD-GAMMON)

# generalization in reinforcement learning

# Large Scale Reinforcement Learning

- Adaptive dynamic programming (ASP) scalable to maybe 10,000 states
  - Backgammon has $10^{20}$ states
  - Chess has $10^{40}$ states

- It is not possible to visit all these states multiple times

⇒ Generalization of states needed

# Function Approximation

- Define state utility function as linear combination of features

$$\hat{U}_\theta(s) = \theta_1\ f_1(s) + \theta_2\ f_2(s) + ... + \theta_n\ f_n(s)$$

- Recall: features to assess Chess state
  - $f_1(s)$ = (number of white pawns) – (number of black pawns)
  - $f_2(s)$ = (number of white rooks) – (number of black rooks)
  - $f_3(s)$ = (number of white queens) – (number of black queens)
  - $f_4(s)$ = king safety
  - $f_5(s)$ = good pawn position
  - etc.

⇒ Reduction from $10^{40}$ to, say, 20 parameters

- Main benefit: ability to assess unseen states

# Learning Feature Weights



- Example: 2 features: $x$ and $y$

$$\hat{U}_\theta(f_1, f_2) = \theta_0 + \theta_1 f_1 + \theta_2 f_2$$

- Current feature weights $\theta_0, \theta_1, \theta_2 = (0.5, 0.2, 0.1)$
- Model's prediction of utility of specific state, e.g., $\hat{U}_\theta(1, 1) = 0.8$
- Sample set of trials, found value $u_\theta(1, 1) = 0.4$
- Error $E_\theta = \frac{1}{2}(\hat{U}_\theta(f_1, f_2) - u_\theta(f_1, f_2))^2$
- How do you update the weights $\theta_i$?

# Gradient Descent Training



- Compute gradient of error

$$\frac{dE_\theta}{d\theta_i} = (\hat{U}_\theta(f_1, f_2) - u_\theta(f_1, f_2))\, f_i$$

- Update against gradient

$$\Delta\theta_i = -\mu\, \frac{dE_\theta}{d\theta_i}$$

- Our example
  - $\Delta\theta_1 = -\mu(\hat{U}_\theta(f_1, f_2) - u_\theta(f_1, f_2))\, f_i = -\mu(0.8 - 0.4)\, 1 = -0.4\mu$
  - $\Delta\theta_2 = -\mu(\hat{U}_\theta(f_1, f_2) - u_\theta(f_1, f_2))\, f_i = -\mu(0.8 - 0.4)\, 1 = -0.4\mu$

# Additional Remarks

- If we know something about the problem
  $\Rightarrow$ we may want to use more complex features

- Our toy example: utility related to Manhattan distance from goal $(x_{\text{goal}}, y_{\text{goal}})$

$$f_3(s) = (x - x_{\text{goal}}) + (y - y_{\text{goal}})$$

- Gradient descent training can also be used for temporal distance learning

# policy search

# Policy Search



- Idea: directly optimize policy

- Policy may be parameterized Q functions, hence:

$$\pi(s) = \max_a \hat{Q}_\theta(s,a)$$

- Stochastic policy, e.g., given by softmax function

$$\pi_\theta(s,a) = \frac{1}{Z_s} e^{\hat{Q}_\theta(s,a)}$$

- Policy value $\rho(\theta)$: expected reward if $\pi_\theta$ is carried out

# Hillclimbing



- Deterministic policy, deterministic environment
  ⇒ optimizing policy value $\rho(\theta)$ may be done in closed form
- If $\rho(\theta)$ differentiable
  ⇒ gradient descent by following policy gradient
- Make small changes to parameters
  ⇒ hillclimb if $\rho(\theta)$ improves
- More complex for stochastic environment

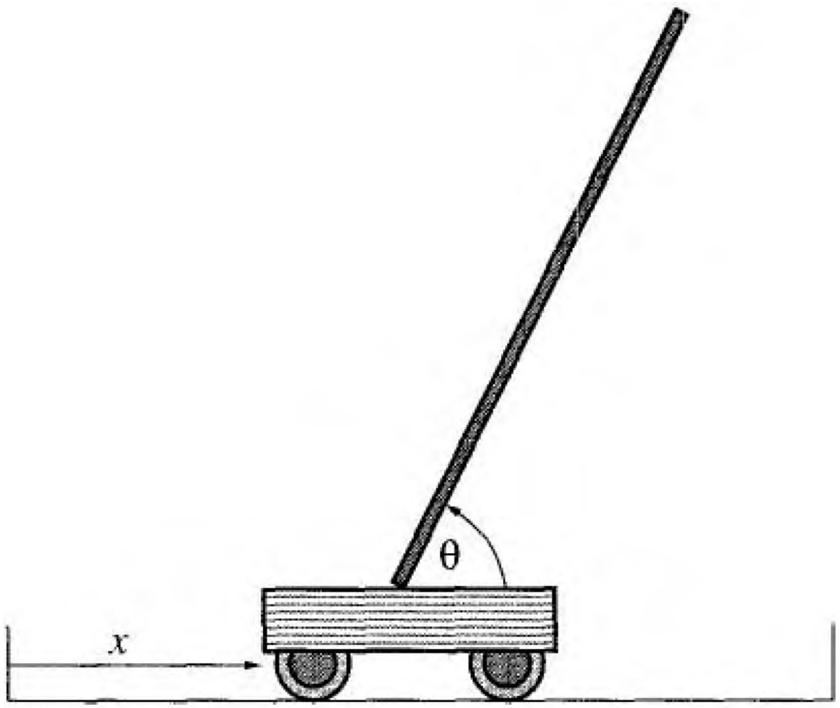
# examples

# Game Playing

- Backgammon: TD-GAMMON (1992)
- Reward only at end of game
- Training with self-play
- 200,000 training games needed


- Competitive with top human players
- Better positional play, worse end game

# Robot Control




- Observe position $x$, vertical speed $\hat{x}$, angle $\theta$, angle speed $\hat{\theta}$
- Action: jerk left or right
- Reward: time balanced until pole falls, or cart out of bounce
- More complex: multiple stacked poles, helicopter flight, walking

# Summary

- Building on Markov decision processes and machine learning

- Passive reinforcement learning
(fixed policy, partially observable environment, stochastic outcomes of actions)
  - sampling (carrying out trials)
  - adaptive dynamic programming
  - temporal difference learning

- Active reinforcement learning
  - greedy in the limit of infinite exploration
  - following optimal policy vs. exploration
  - exploratory adaptive dynamic programming

- Generalization: representing utility function with small set of parameters

- Policy search