

Lecture 1: Introduction to Reinforcement Learning

David Silver

Outline

- 1 Admin
- 2 About Reinforcement Learning
- 3 The Reinforcement Learning Problem
- 4 Inside An RL Agent
- 5 Problems within Reinforcement Learning

Class Information

- Thursdays 9:30 to 11:00am
- Website:
<http://www.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html>
- Group:
<http://groups.google.com/group/csml-advanced-topics>
- Contact me: d.silver@cs.ucl.ac.uk

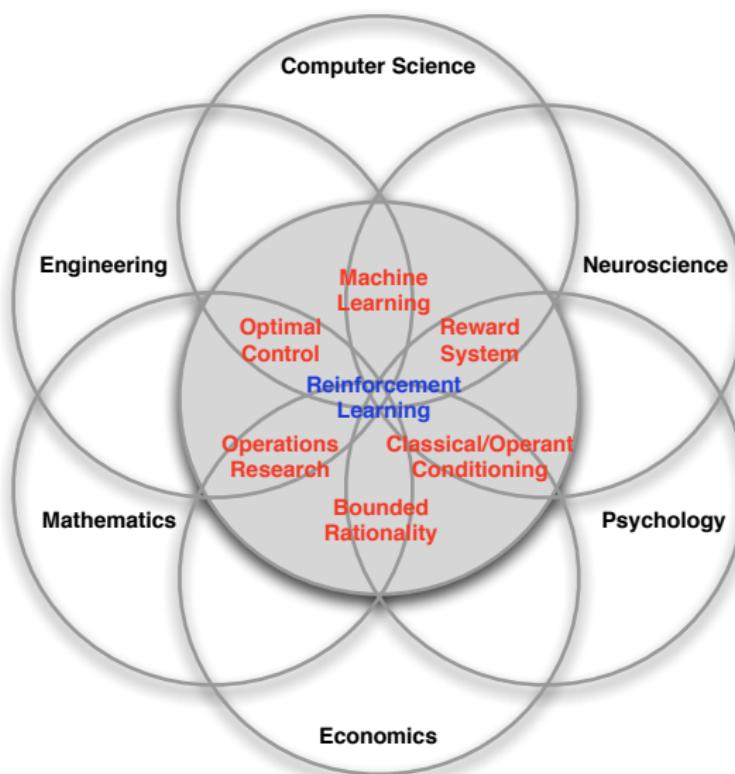
Assessment

- Assessment will be 50% coursework, 50% exam
- Coursework
 - Assignment A: RL problem
 - Assignment B: Kernels problem
 - Assessment = $\max(\text{assignment1}, \text{assignment2})$
- Examination
 - A: 3 RL questions
 - B: 3 kernels questions
 - Answer any 3 questions

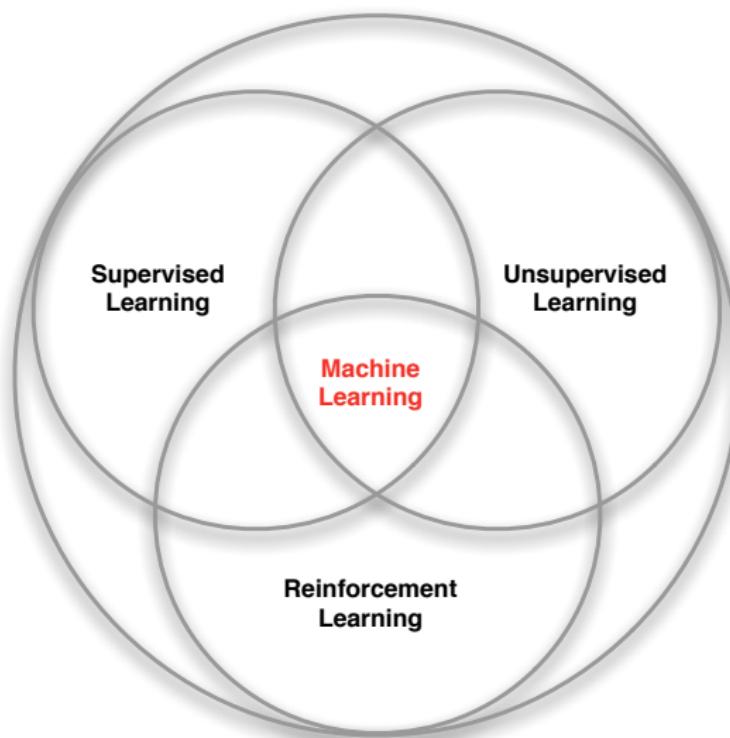
Textbooks

- An Introduction to Reinforcement Learning, Sutton and Barto, 1998
 - MIT Press, 1998
 - ~ 40 pounds
 - Available free online!
 - <http://webdocs.cs.ualberta.ca/~sutton/book/the-book.html>
- Algorithms for Reinforcement Learning, Szepesvari
 - Morgan and Claypool, 2010
 - ~ 20 pounds
 - Available free online!
 - <http://www.ualberta.ca/~szepesva/papers/RLAlgsInMDPs.pdf>

Many Faces of Reinforcement Learning



Branches of Machine Learning



Characteristics of Reinforcement Learning

What makes reinforcement learning different from other machine learning paradigms?

- There is no supervisor, only a *reward* signal
- Feedback is delayed, not instantaneous
- Time really matters (sequential, non i.i.d data)
- Agent's actions affect the subsequent data it receives

Examples of Reinforcement Learning

- Fly stunt manoeuvres in a helicopter
- Defeat the world champion at Backgammon
- Manage an investment portfolio
- Control a power station
- Make a humanoid robot walk
- Play many different Atari games better than humans

Helicopter Manoeuvres

Bipedal Robots

Atari

Rewards

- A **reward** R_t is a scalar feedback signal
- Indicates how well agent is doing at step t
- The agent's job is to maximise cumulative reward

Reinforcement learning is based on the **reward hypothesis**

Definition (Reward Hypothesis)

All goals can be described by the maximisation of expected cumulative reward

Do you agree with this statement?

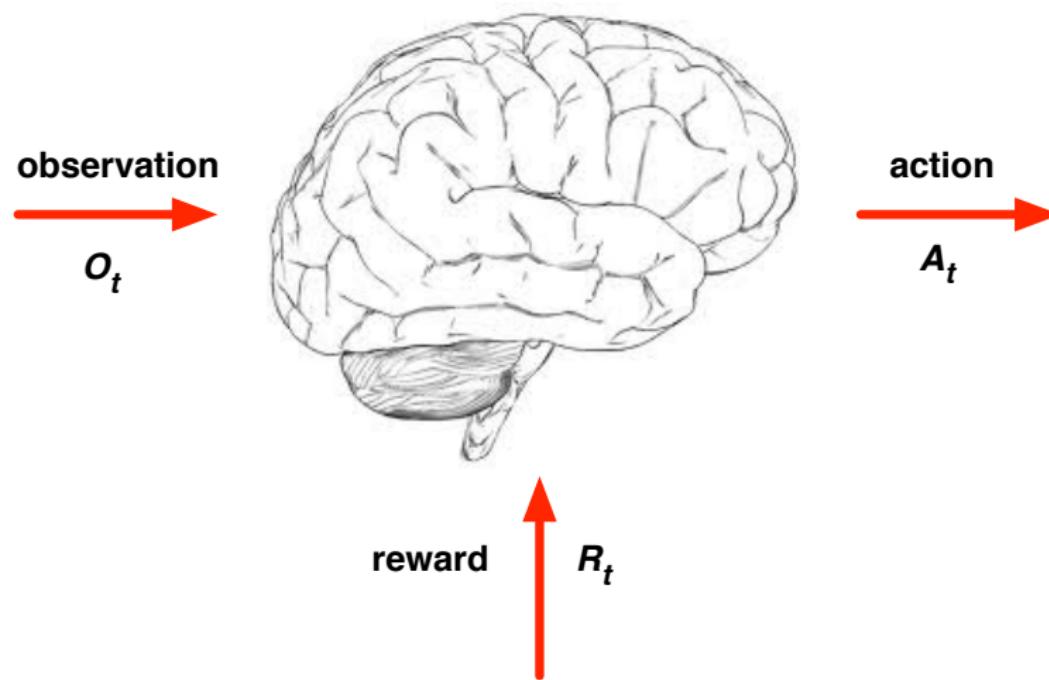
Examples of Rewards

- Fly stunt manoeuvres in a helicopter
 - +ve reward for following desired trajectory
 - -ve reward for crashing
- Defeat the world champion at Backgammon
 - +/-ve reward for winning/losing a game
- Manage an investment portfolio
 - +ve reward for each \$ in bank
- Control a power station
 - +ve reward for producing power
 - -ve reward for exceeding safety thresholds
- Make a humanoid robot walk
 - +ve reward for forward motion
 - -ve reward for falling over
- Play many different Atari games better than humans
 - +/-ve reward for increasing/decreasing score

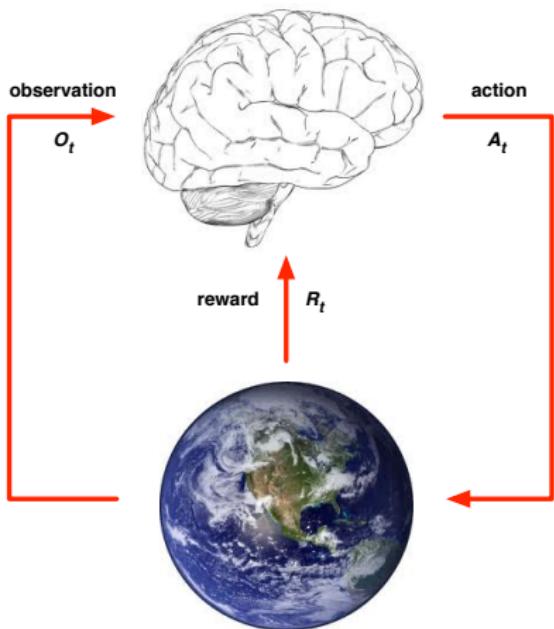
Sequential Decision Making

- Goal: *select actions to maximise total future reward*
- Actions may have long term consequences
- Reward may be delayed
- It may be better to sacrifice immediate reward to gain more long-term reward
- Examples:
 - A financial investment (may take months to mature)
 - Refuelling a helicopter (might prevent a crash in several hours)
 - Blocking opponent moves (might help winning chances many moves from now)

Agent and Environment



Agent and Environment



- At each step t the agent:
 - Executes action A_t
 - Receives observation O_t
 - Receives scalar reward R_t
- The environment:
 - Receives action A_t
 - Emits observation O_{t+1}
 - Emits scalar reward R_{t+1}
- t increments at env. step

History and State

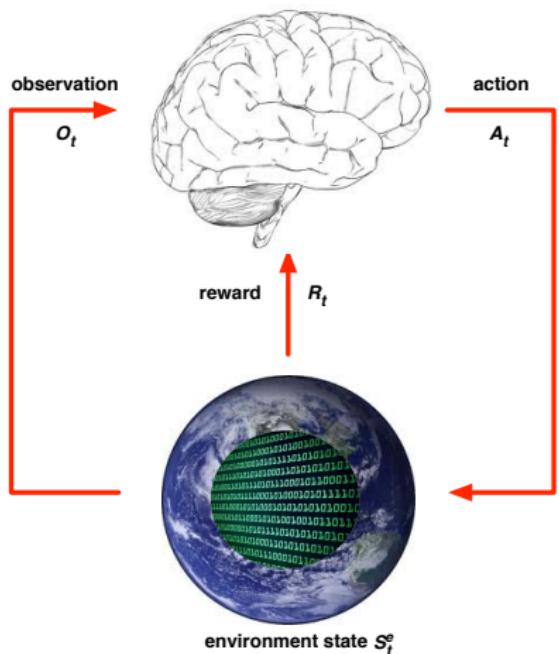
- The **history** is the sequence of observations, actions, rewards

$$H_t = O_1, R_1, A_1, \dots, A_{t-1}, O_t, R_t$$

- i.e. all observable variables up to time t
- i.e. the sensorimotor stream of a robot or embodied agent
- What happens next depends on the history:
 - The agent selects actions
 - The environment selects observations/rewards
- **State** is the information used to determine what happens next
- Formally, state is a function of the history:

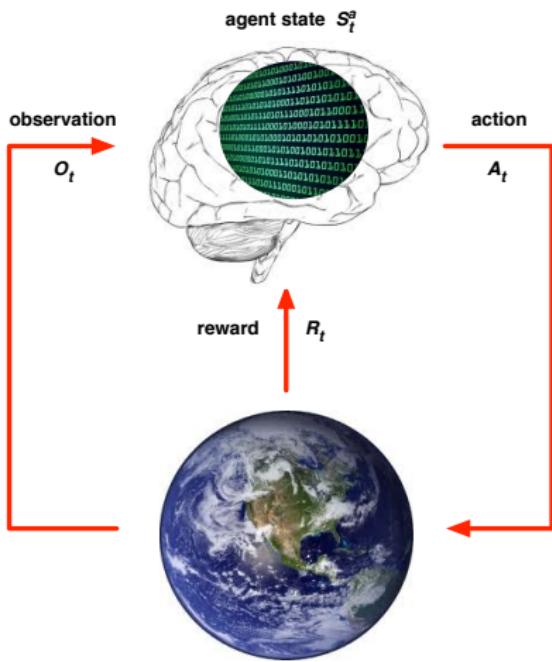
$$S_t = f(H_t)$$

Environment State



- The **environment state** S_t^e is the environment's private representation
- i.e. whatever data the environment uses to pick the next observation/reward
- The environment state is not usually visible to the agent
- Even if S_t^e is visible, it may contain irrelevant information

Agent State



- The **agent state** S_t^a is the agent's internal representation
 - i.e. whatever information the agent uses to pick the next action
 - i.e. it is the information used by reinforcement learning algorithms
- It can be any function of history:

$$S_t^a = f(H_t)$$

Information State

An **information state** (a.k.a. **Markov state**) contains all useful information from the history.

Definition

A state S_t is **Markov** if and only if

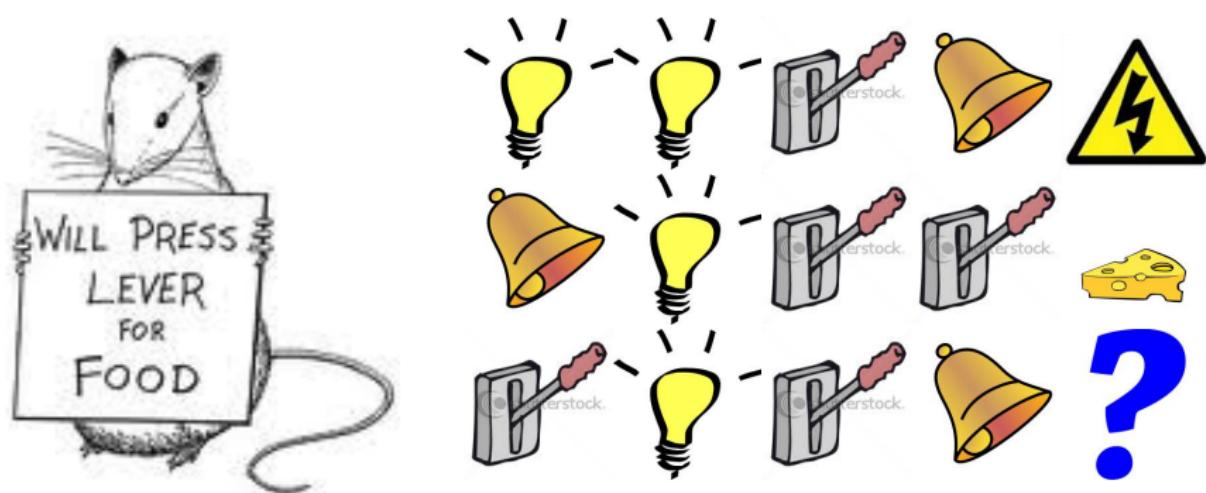
$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, \dots, S_t]$$

- “The future is independent of the past given the present”

$$H_{1:t} \rightarrow S_t \rightarrow H_{t+1:\infty}$$

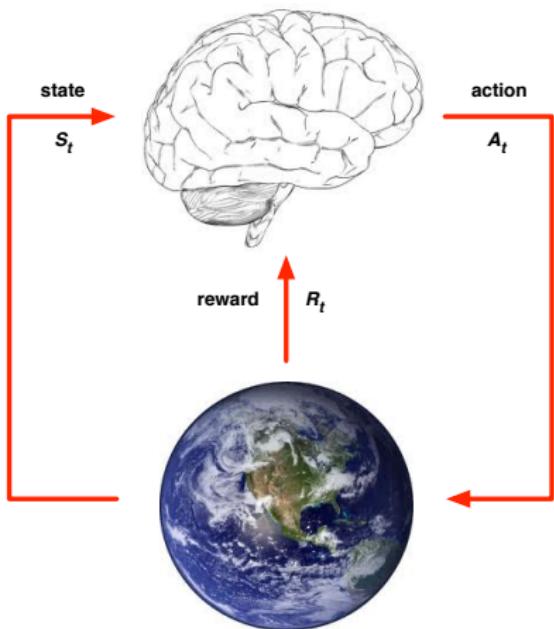
- Once the state is known, the history may be thrown away
- i.e. The state is a sufficient statistic of the future
- The environment state S_t^e is Markov
- The history H_t is Markov

Rat Example



- What if agent state = last 3 items in sequence?
- What if agent state = counts for lights, bells and levers?
- What if agent state = complete sequence?

Fully Observable Environments



Full observability: agent directly observes environment state

$$O_t = S_t^a = S_t^e$$

- Agent state = environment state = information state
- Formally, this is a **Markov decision process (MDP)**
- (Next lecture and the majority of this course)

Partially Observable Environments

- **Partial observability:** agent **indirectly** observes environment:
 - A robot with camera vision isn't told its absolute location
 - A trading agent only observes current prices
 - A poker playing agent only observes public cards
- Now agent state \neq environment state
- Formally this is a **partially observable Markov decision process** (POMDP)
- Agent must construct its own state representation S_t^a , e.g.
 - Complete history: $S_t^a = H_t$
 - **Beliefs** of environment state: $S_t^a = (\mathbb{P}[S_t^e = s^1], \dots, \mathbb{P}[S_t^e = s^n])$
 - Recurrent neural network: $S_t^a = \sigma(S_{t-1}^a W_s + O_t W_o)$

Major Components of an RL Agent

- An RL agent may include one or more of these components:
 - Policy: agent's behaviour function
 - Value function: how good is each state and/or action
 - Model: agent's representation of the environment

Policy

- A **policy** is the agent's behaviour
- It is a map from state to action, e.g.
- Deterministic policy: $a = \pi(s)$
- Stochastic policy: $\pi(a|s) = \mathbb{P}[A_t = a | S_t = s]$

Value Function

- Value function is a prediction of future reward
- Used to evaluate the goodness/badness of states
- And therefore to select between actions, e.g.

$$v_{\pi}(s) = \mathbb{E}_{\pi} [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s]$$

Example: Value Function in Atari

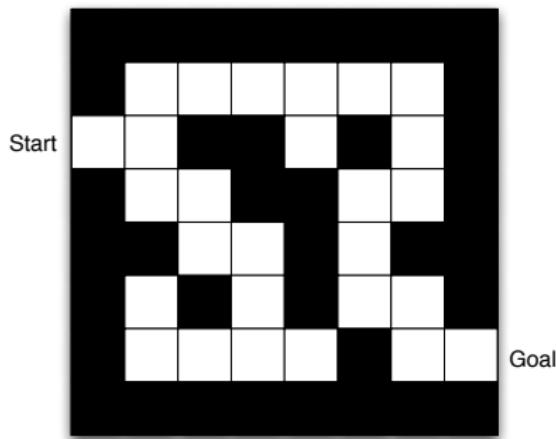
Model

- A **model** predicts what the environment will do next
- \mathcal{P} predicts the next state
- \mathcal{R} predicts the next (immediate) reward, e.g.

$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$$

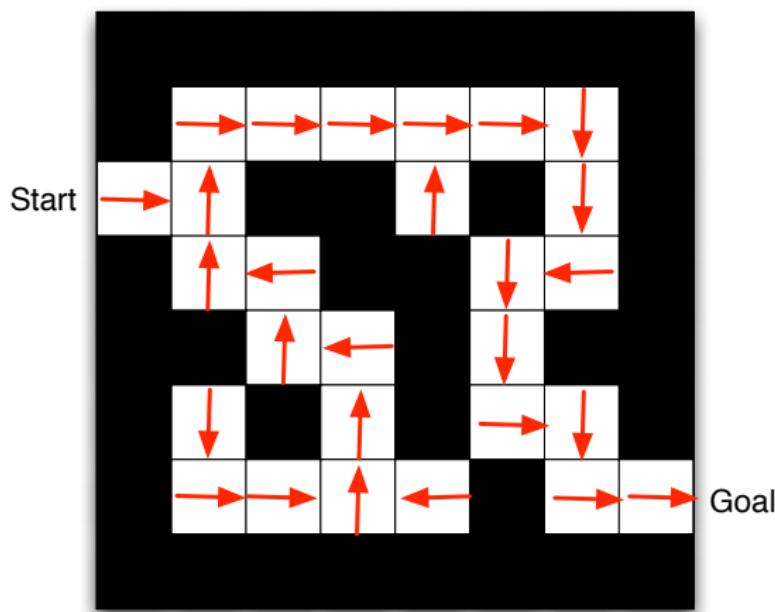
$$\mathcal{R}_s^a = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

Maze Example



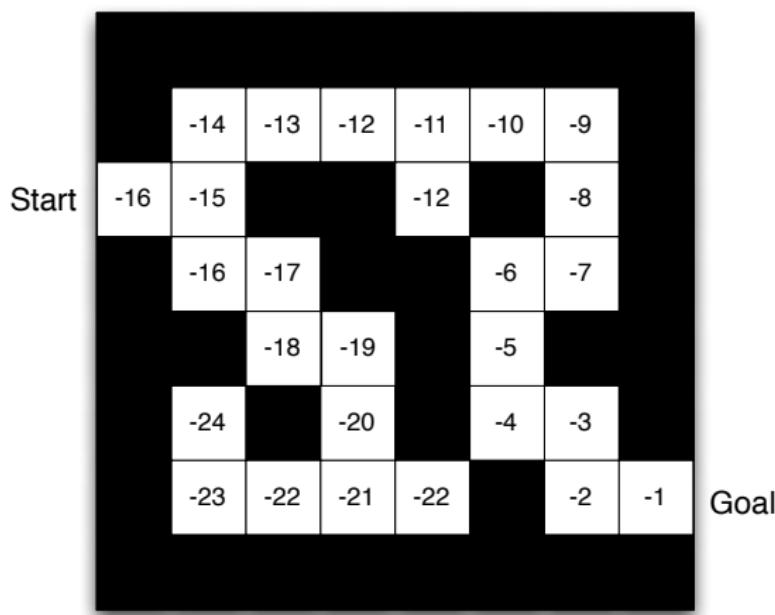
- Rewards: -1 per time-step
- Actions: N, E, S, W
- States: Agent's location

Maze Example: Policy



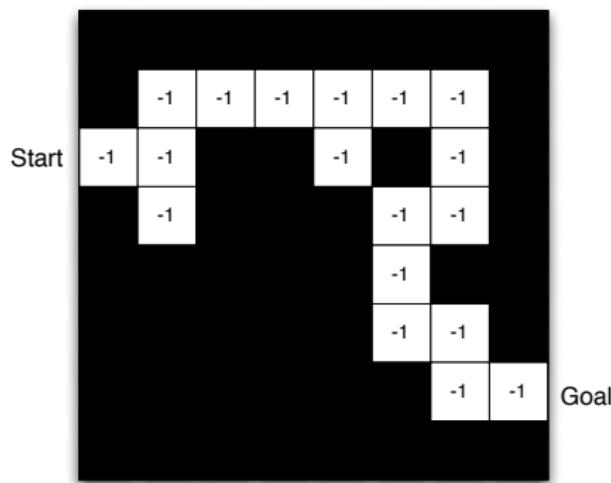
- Arrows represent policy $\pi(s)$ for each state s

Maze Example: Value Function



- Numbers represent value $v_\pi(s)$ of each state s

Maze Example: Model



- Agent may have an internal model of the environment
- Dynamics: how actions change the state
- Rewards: how much reward from each state
- The model may be imperfect

- Grid layout represents transition model $\mathcal{P}_{ss'}^a$
- Numbers represent immediate reward \mathcal{R}_s^a from each state s (same for all a)

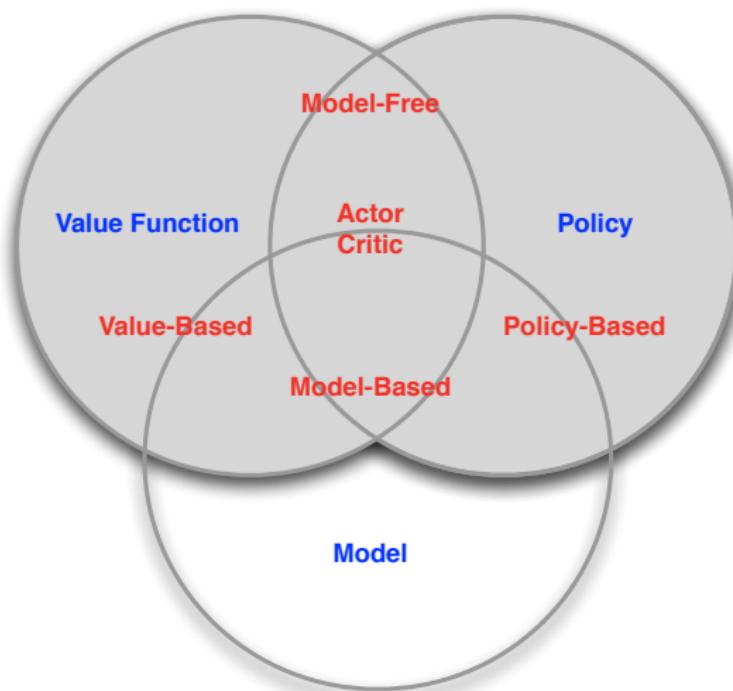
Categorizing RL agents (1)

- Value Based
 - No Policy (Implicit)
 - Value Function
- Policy Based
 - Policy
 - No Value Function
- Actor Critic
 - Policy
 - Value Function

Categorizing RL agents (2)

- Model Free
 - Policy and/or Value Function
 - No Model
- Model Based
 - Policy and/or Value Function
 - Model

RL Agent Taxonomy

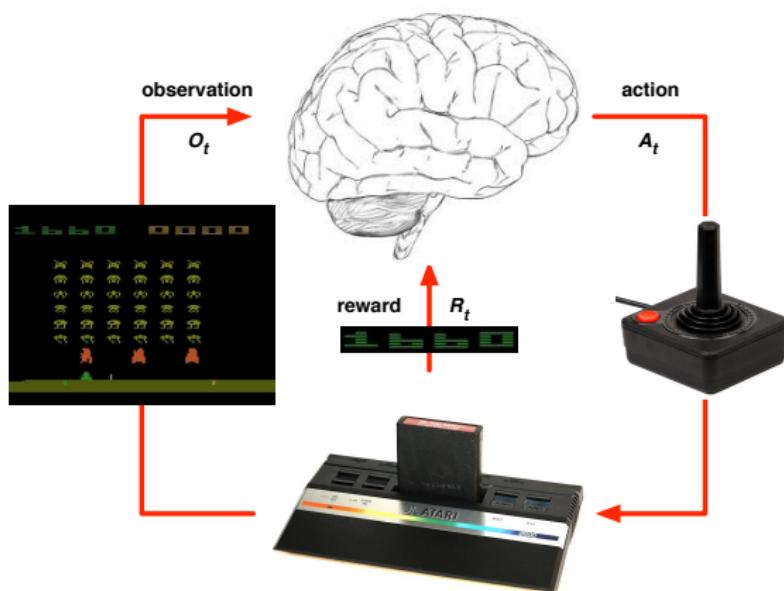


Learning and Planning

Two fundamental problems in sequential decision making

- Reinforcement Learning:
 - The environment is initially unknown
 - The agent interacts with the environment
 - The agent improves its policy
- Planning:
 - A model of the environment is known
 - The agent performs computations with its model (without any external interaction)
 - The agent improves its policy
 - a.k.a. deliberation, reasoning, introspection, pondering, thought, search

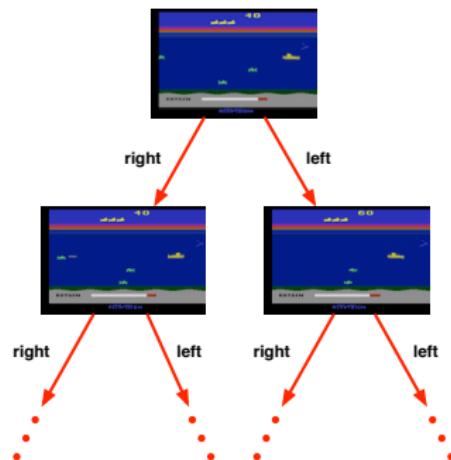
Atari Example: Reinforcement Learning



- Rules of the game are unknown
- Learn directly from interactive game-play
- Pick actions on joystick, see pixels and scores

Atari Example: Planning

- Rules of the game are known
- Can query emulator
 - perfect model inside agent's brain
- If I take action a from state s :
 - what would the next state be?
 - what would the score be?
- Plan ahead to find optimal policy
 - e.g. tree search



Exploration and Exploitation (1)

- Reinforcement learning is like trial-and-error learning
- The agent should discover a good policy
- From its experiences of the environment
- Without losing too much reward along the way

Exploration and Exploitation (2)

- *Exploration* finds more information about the environment
- *Exploitation* exploits known information to maximise reward
- It is usually important to explore as well as exploit

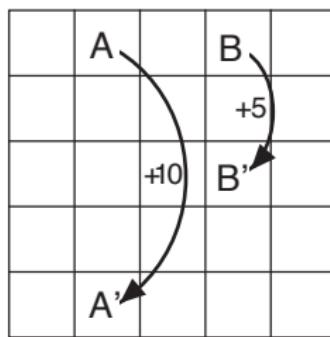
Examples

- Restaurant Selection
 - Exploitation Go to your favourite restaurant
 - Exploration Try a new restaurant
- Online Banner Advertisements
 - Exploitation Show the most successful advert
 - Exploration Show a different advert
- Oil Drilling
 - Exploitation Drill at the best known location
 - Exploration Drill at a new location
- Game Playing
 - Exploitation Play the move you believe is best
 - Exploration Play an experimental move

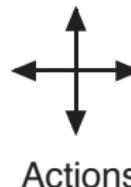
Prediction and Control

- Prediction: evaluate the future
 - Given a policy
- Control: optimise the future
 - Find the best policy

Gridworld Example: Prediction



(a)

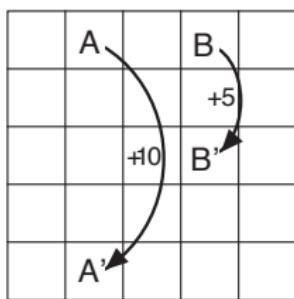


| | | | | |
|------|------|------|------|------|
| 3.3 | 8.8 | 4.4 | 5.3 | 1.5 |
| 1.5 | 3.0 | 2.3 | 1.9 | 0.5 |
| 0.1 | 0.7 | 0.7 | 0.4 | -0.4 |
| -1.0 | -0.4 | -0.4 | -0.6 | -1.2 |
| -1.9 | -1.3 | -1.2 | -1.4 | -2.0 |

(b)

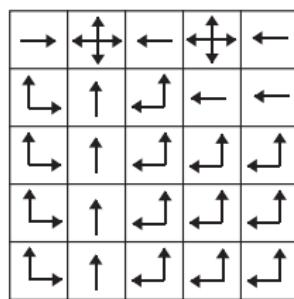
What is the value function for the uniform random policy?

Gridworld Example: Control



a) gridworld

| | | | | |
|------|------|------|------|------|
| 22.0 | 24.4 | 22.0 | 19.4 | 17.5 |
| 19.8 | 22.0 | 19.8 | 17.8 | 16.0 |
| 17.8 | 19.8 | 17.8 | 16.0 | 14.4 |
| 16.0 | 17.8 | 16.0 | 14.4 | 13.0 |
| 14.4 | 16.0 | 14.4 | 13.0 | 11.7 |

b) v_* c) π_*

What is the optimal value function over all possible policies?
 What is the optimal policy?

Course Outline

- Part I: Elementary Reinforcement Learning

- 1 Introduction to RL
- 2 Markov Decision Processes
- 3 Planning by Dynamic Programming
- 4 Model-Free Prediction
- 5 Model-Free Control

- Part II: Reinforcement Learning in Practice

- 1 Value Function Approximation
- 2 Policy Gradient Methods
- 3 Integrating Learning and Planning
- 4 Exploration and Exploitation
- 5 Case study - RL in games

Lecture 2: Markov Decision Processes

David Silver

1 Markov Processes

2 Markov Reward Processes

3 Markov Decision Processes

4 Extensions to MDPs

Introduction to MDPs

- *Markov decision processes* formally describe an environment for reinforcement learning
- Where the environment is *fully observable*
- i.e. The current *state* completely characterises the process
- Almost all RL problems can be formalised as MDPs, e.g.
 - Optimal control primarily deals with continuous MDPs
 - Partially observable problems can be converted into MDPs
 - Bandits are MDPs with one state

Markov Property

“The future is independent of the past given the present”

Definition

A state S_t is *Markov* if and only if

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, \dots, S_t]$$

- The state captures all relevant information from the history
- Once the state is known, the history may be thrown away
- i.e. The state is a sufficient statistic of the future

State Transition Matrix

For a Markov state s and successor state s' , the *state transition probability* is defined by

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$$

State transition matrix \mathcal{P} defines transition probabilities from all states s to all successor states s' ,

$$\mathcal{P} = \text{from} \begin{bmatrix} & & \text{to} \\ \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix}$$

where each row of the matrix sums to 1.

Markov Process

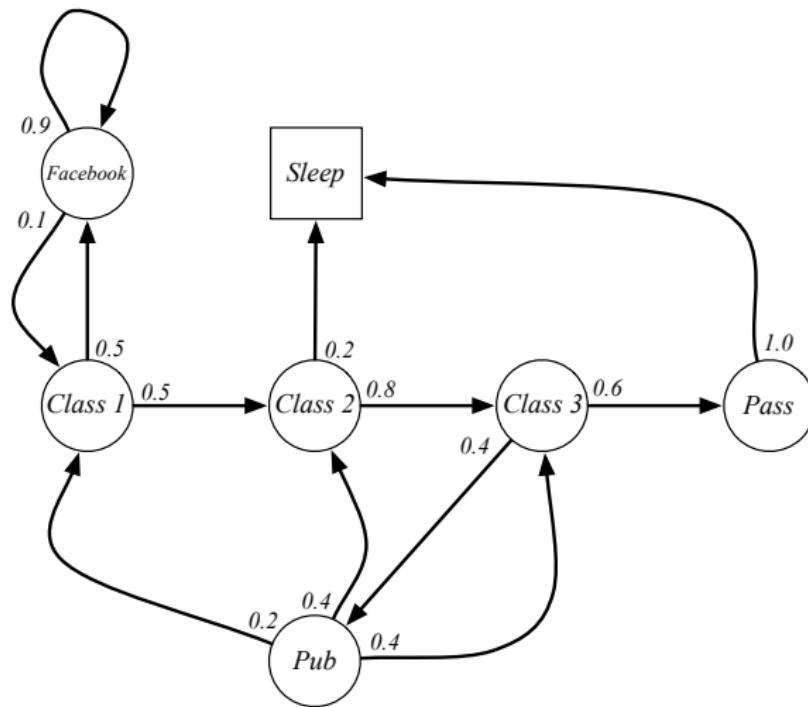
A Markov process is a memoryless random process, i.e. a sequence of random states S_1, S_2, \dots with the Markov property.

Definition

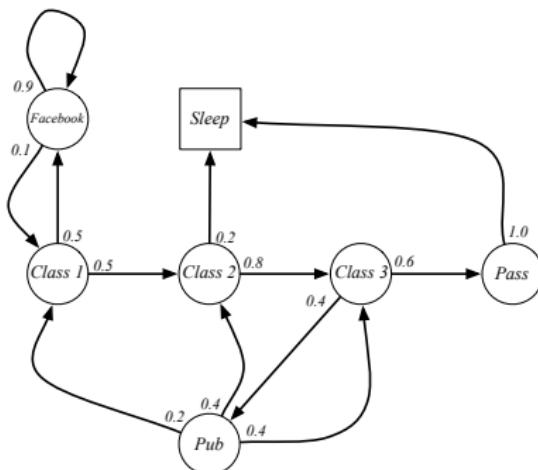
A *Markov Process* (or *Markov Chain*) is a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$

- \mathcal{S} is a (finite) set of states
- \mathcal{P} is a state transition probability matrix,
$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$$

Example: Student Markov Chain



Example: Student Markov Chain Episodes

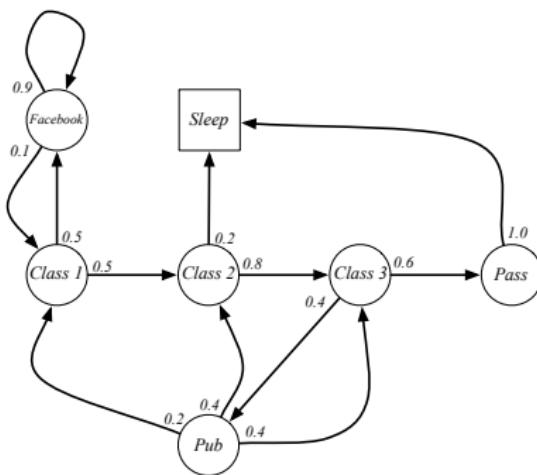


Sample **episodes** for Student Markov Chain starting from $S_1 = C1$

S_1, S_2, \dots, S_T

- C1 C2 C3 Pass Sleep
- C1 FB FB C1 C2 Sleep
- C1 C2 C3 Pub C2 C3 Pass Sleep
- C1 FB FB C1 C2 C3 Pub C1 FB FB FB C1 C2 C3 Pub C2 Sleep

Example: Student Markov Chain Transition Matrix



$$\mathcal{P} = \begin{matrix} & C_1 & C_2 & C_3 & \text{Pass} & \text{Pub} & FB & \text{Sleep} \\ C_1 & 0.5 & & & & & & 0.2 \\ C_2 & & 0.8 & & & & & 0.2 \\ C_3 & & & 0.6 & & 0.4 & & 1.0 \\ \text{Pass} & & & & 0.2 & 0.4 & & 0.9 \\ \text{Pub} & & & & & 0.1 & & 1 \\ FB & & & & & & 0.5 & \\ \text{Sleep} & & & & & & & 1 \end{matrix}$$

Markov Reward Process

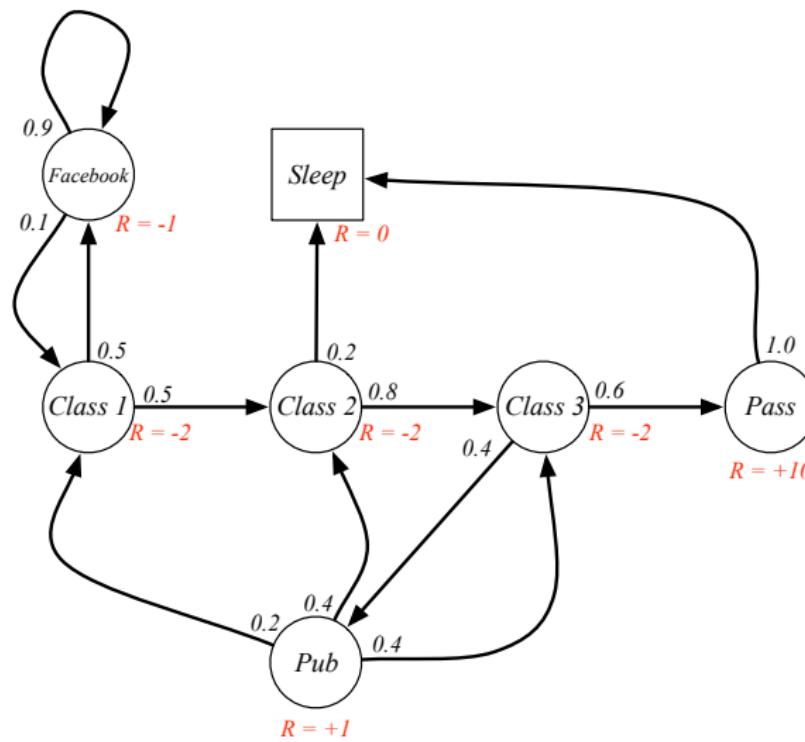
A Markov reward process is a Markov chain with values.

Definition

A *Markov Reward Process* is a tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

- \mathcal{S} is a finite set of states
- \mathcal{P} is a state transition probability matrix,
 $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$
- \mathcal{R} is a reward function, $\mathcal{R}_s = \mathbb{E}[R_{t+1} \mid S_t = s]$
- γ is a discount factor, $\gamma \in [0, 1]$

Example: Student MRP



Return

Definition

The *return* G_t is the total discounted reward from time-step t .

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- The *discount* $\gamma \in [0, 1]$ is the present value of future rewards
- The value of receiving reward R after $k + 1$ time-steps is $\gamma^k R$.
- This values immediate reward above delayed reward.
 - γ close to 0 leads to "myopic" evaluation
 - γ close to 1 leads to "far-sighted" evaluation

Why discount?

Most Markov reward and decision processes are discounted. Why?

- Mathematically convenient to discount rewards
- Avoids infinite returns in cyclic Markov processes
- Uncertainty about the future may not be fully represented
- If the reward is financial, immediate rewards may earn more interest than delayed rewards
- Animal/human behaviour shows preference for immediate reward
- It is sometimes possible to use *undiscounted* Markov reward processes (i.e. $\gamma = 1$), e.g. if all sequences terminate.

Value Function

The value function $v(s)$ gives the long-term value of state s

Definition

The *state value function* $v(s)$ of an MRP is the expected return starting from state s

$$v(s) = \mathbb{E} [G_t \mid S_t = s]$$

Example: Student MRP Returns

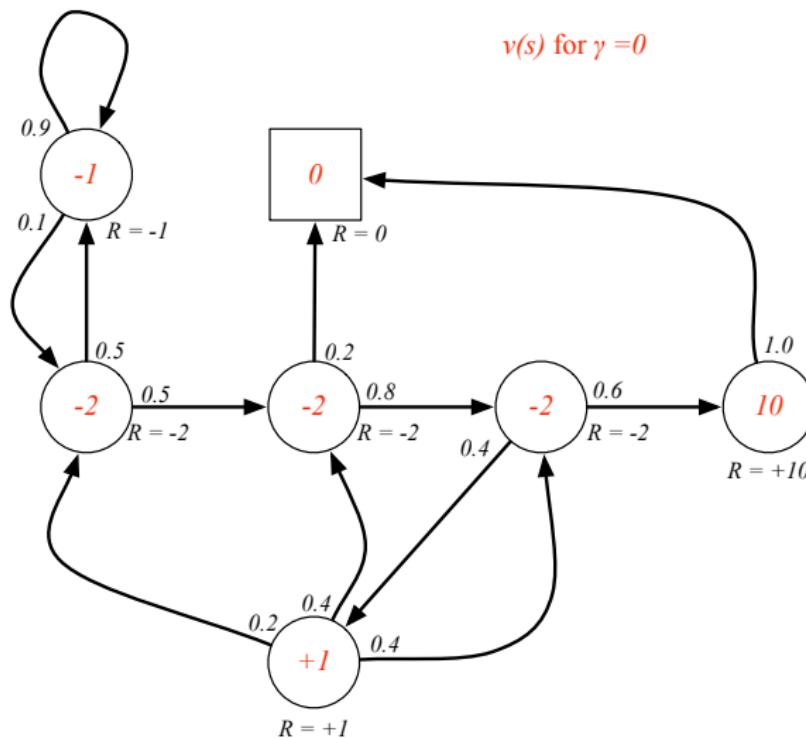
Sample **returns** for Student MRP:

Starting from $S_1 = C1$ with $\gamma = \frac{1}{2}$

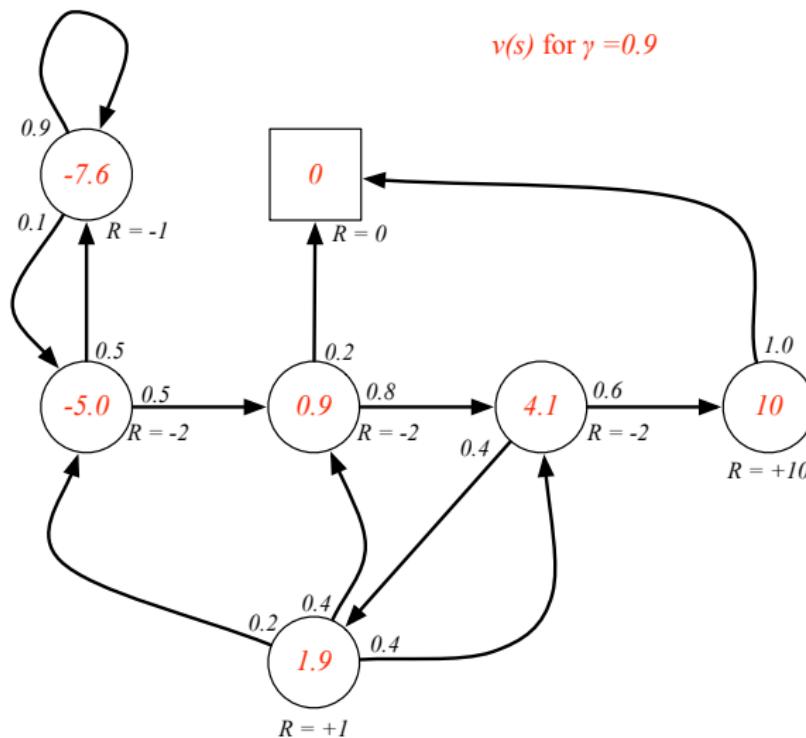
$$G_1 = R_2 + \gamma R_3 + \dots + \gamma^{T-2} R_T$$

| | | | |
|--------------------------------|---|---|--------|
| C1 C2 C3 Pass Sleep | $v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 10 * \frac{1}{8}$ | = | -2.25 |
| C1 FB FB C1 C2 Sleep | $v_1 = -2 - 1 * \frac{1}{2} - 1 * \frac{1}{4} - 2 * \frac{1}{8} - 2 * \frac{1}{16}$ | = | -3.125 |
| C1 C2 C3 Pub C2 C3 Pass Sleep | $v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 1 * \frac{1}{8} - 2 * \frac{1}{16} \dots$ | = | -3.41 |
| C1 FB FB C1 C2 C3 Pub C1 ... | $v_1 = -2 - 1 * \frac{1}{2} - 1 * \frac{1}{4} - 2 * \frac{1}{8} - 2 * \frac{1}{16} \dots$ | = | -3.20 |
| FB FB FB C1 C2 C3 Pub C2 Sleep | | | |

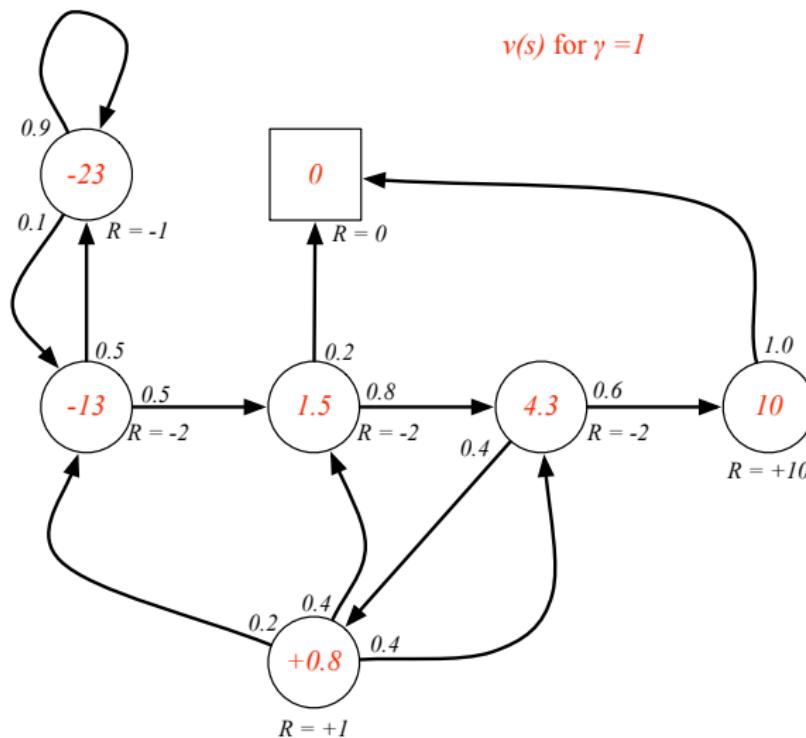
Example: State-Value Function for Student MRP (1)



Example: State-Value Function for Student MRP (2)



Example: State-Value Function for Student MRP (3)



Bellman Equation for MRPs

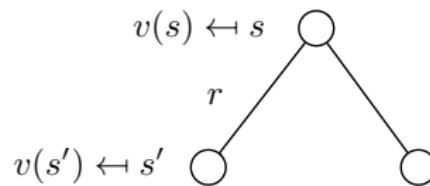
The value function can be decomposed into two parts:

- immediate reward R_{t+1}
- discounted value of successor state $\gamma v(S_{t+1})$

$$\begin{aligned}v(s) &= \mathbb{E}[G_t \mid S_t = s] \\&= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s] \\&= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \mid S_t = s] \\&= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\&= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]\end{aligned}$$

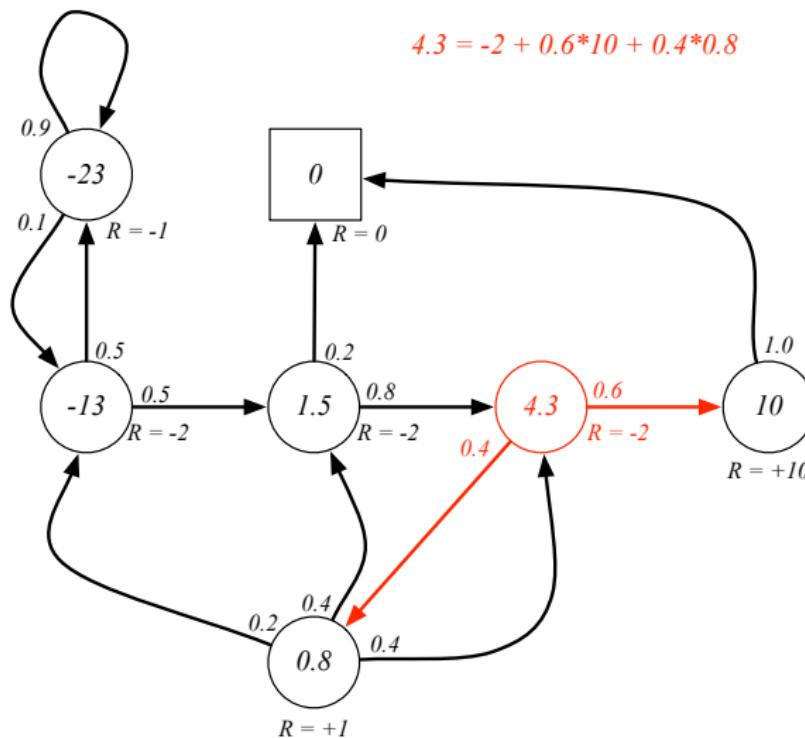
Bellman Equation for MRPs (2)

$$v(s) = \mathbb{E} [R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]$$



$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$

Example: Bellman Equation for Student MRP



Bellman Equation in Matrix Form

The Bellman equation can be expressed concisely using matrices,

$$v = \mathcal{R} + \gamma \mathcal{P} v$$

where v is a column vector with one entry per state

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \vdots \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

Solving the Bellman Equation

- The Bellman equation is a linear equation
- It can be solved directly:

$$\begin{aligned}v &= \mathcal{R} + \gamma \mathcal{P}v \\(I - \gamma \mathcal{P})v &= \mathcal{R} \\v &= (I - \gamma \mathcal{P})^{-1} \mathcal{R}\end{aligned}$$

- Computational complexity is $O(n^3)$ for n states
- Direct solution only possible for small MRPs
- There are many iterative methods for large MRPs, e.g.
 - Dynamic programming
 - Monte-Carlo evaluation
 - Temporal-Difference learning

Markov Decision Process

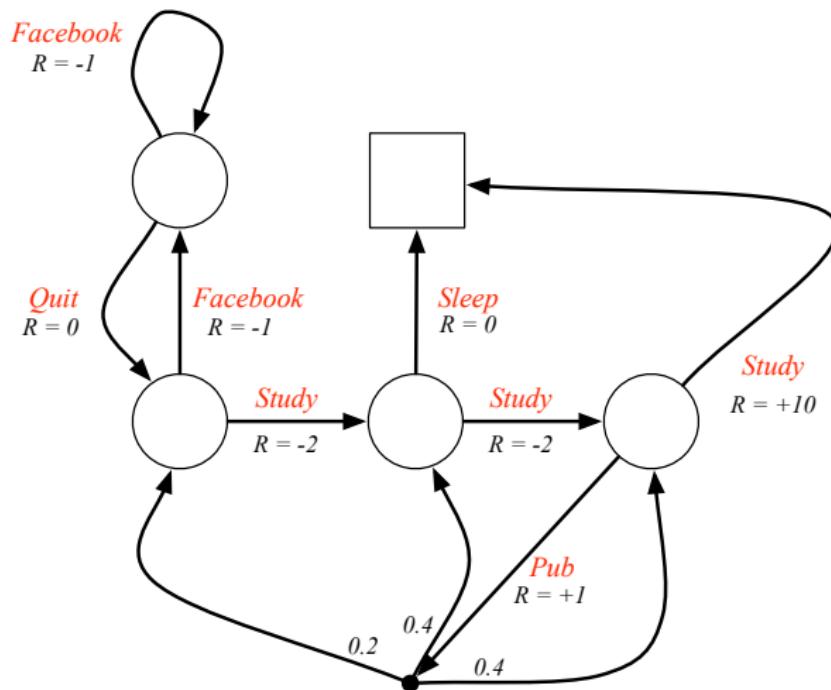
A Markov decision process (MDP) is a Markov reward process with decisions. It is an *environment* in which all states are Markov.

Definition

A *Markov Decision Process* is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

- \mathcal{S} is a finite set of states
- \mathcal{A} is a finite set of actions
- \mathcal{P} is a state transition probability matrix,
 $\mathcal{P}_{ss'}^{\textcolor{red}{a}} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = \textcolor{red}{a}]$
- \mathcal{R} is a reward function, $\mathcal{R}_s^{\textcolor{red}{a}} = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = \textcolor{red}{a}]$
- γ is a discount factor $\gamma \in [0, 1]$.

Example: Student MDP



Policies (1)

Definition

A *policy* π is a distribution over actions given states,

$$\pi(a|s) = \mathbb{P}[A_t = a \mid S_t = s]$$

- A policy fully defines the behaviour of an agent
- MDP policies depend on the current state (not the history)
- i.e. Policies are *stationary* (time-independent),
 $A_t \sim \pi(\cdot|S_t), \forall t > 0$

Policies (2)

- Given an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π
- The state sequence S_1, S_2, \dots is a Markov process $\langle \mathcal{S}, \mathcal{P}^\pi \rangle$
- The state and reward sequence S_1, R_2, S_2, \dots is a Markov reward process $\langle \mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma \rangle$
- where

$$\mathcal{P}_{s,s'}^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{ss'}^a$$

$$\mathcal{R}_s^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_s^a$$

Value Function

Definition

The *state-value function* $v_\pi(s)$ of an MDP is the expected return starting from state s , and then following policy π

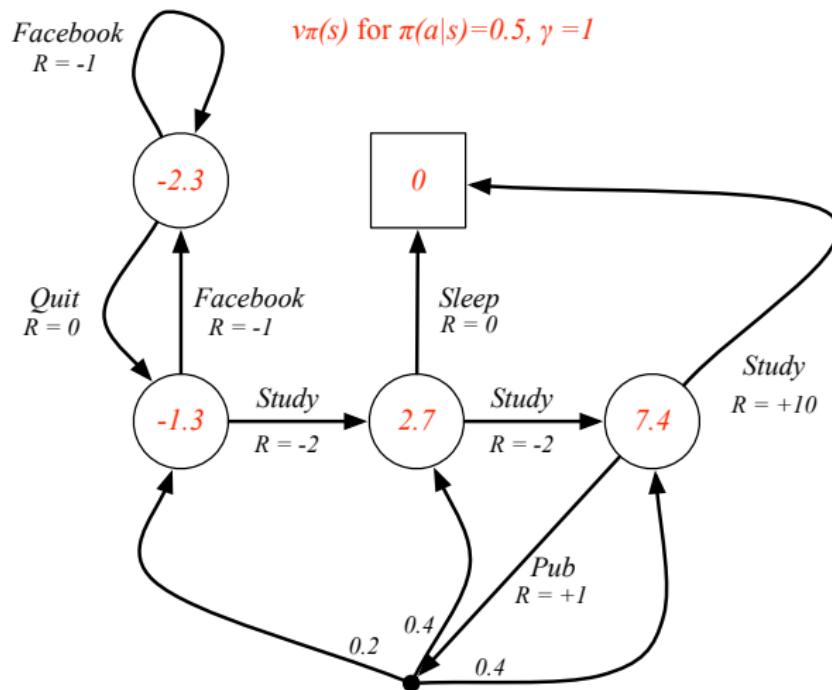
$$v_\pi(s) = \mathbb{E}_\pi [G_t \mid S_t = s]$$

Definition

The *action-value function* $q_\pi(s, a)$ is the expected return starting from state s , taking action a , and then following policy π

$$q_\pi(s, a) = \mathbb{E}_\pi [G_t \mid S_t = s, A_t = a]$$

Example: State-Value Function for Student MDP



Bellman Expectation Equation

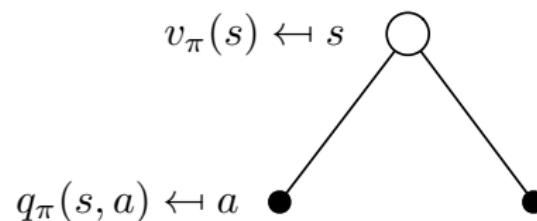
The state-value function can again be decomposed into immediate reward plus discounted value of successor state,

$$v_{\pi}(s) = \mathbb{E}_{\pi} [R_{t+1} + \gamma v_{\pi}(S_{t+1}) \mid S_t = s]$$

The action-value function can similarly be decomposed,

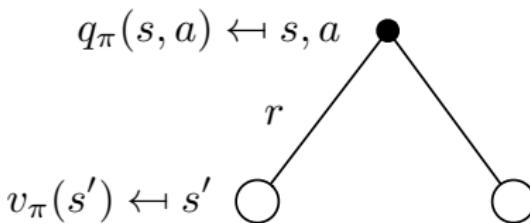
$$q_{\pi}(s, a) = \mathbb{E}_{\pi} [R_{t+1} + \gamma q_{\pi}(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]$$

Bellman Expectation Equation for V^π



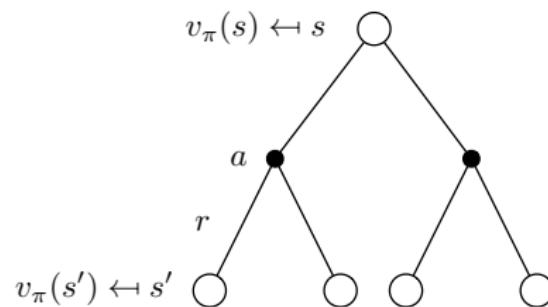
$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a)$$

Bellman Expectation Equation for Q^π



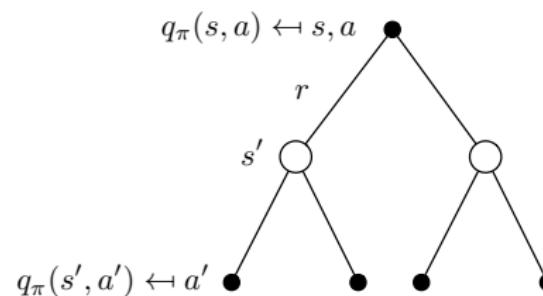
$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$

Bellman Expectation Equation for v_π (2)



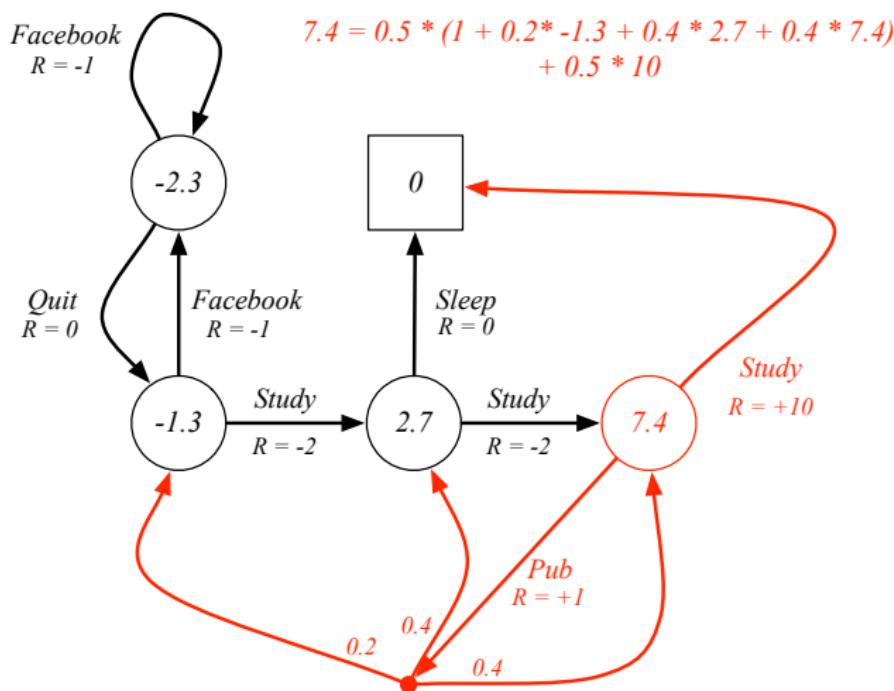
$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

Bellman Expectation Equation for q_π (2)



$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a')$$

Example: Bellman Expectation Equation in Student MDP



Bellman Expectation Equation (Matrix Form)

The Bellman expectation equation can be expressed concisely using the induced MRP,

$$v_\pi = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi$$

with direct solution

$$v_\pi = (I - \gamma \mathcal{P}^\pi)^{-1} \mathcal{R}^\pi$$

Optimal Value Function

Definition

The *optimal state-value function* $v_*(s)$ is the maximum value function over all policies

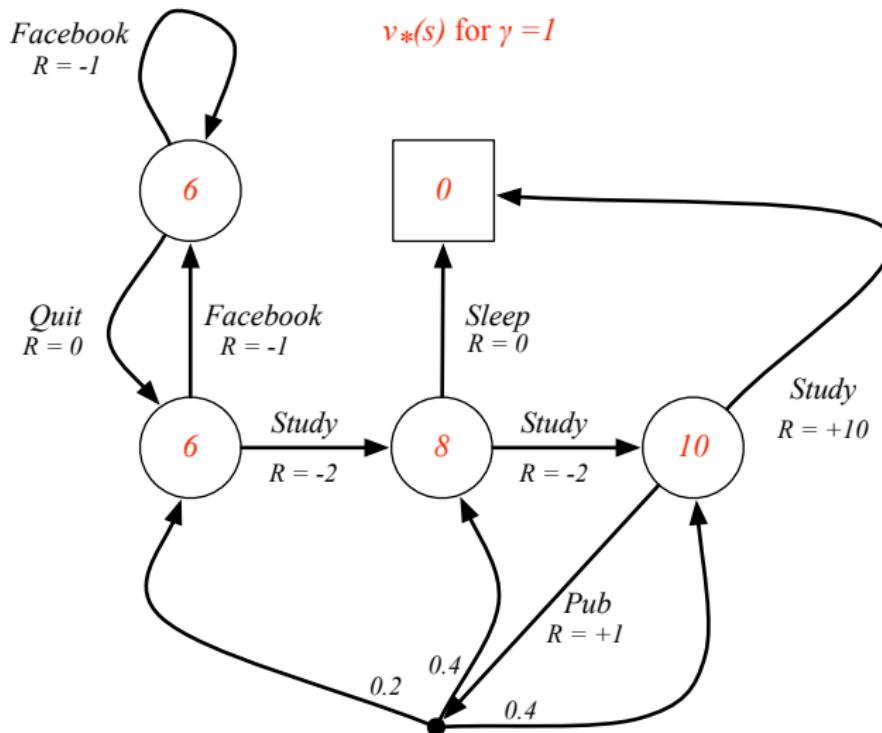
$$v_*(s) = \max_{\pi} v_{\pi}(s)$$

The *optimal action-value function* $q_*(s, a)$ is the maximum action-value function over all policies

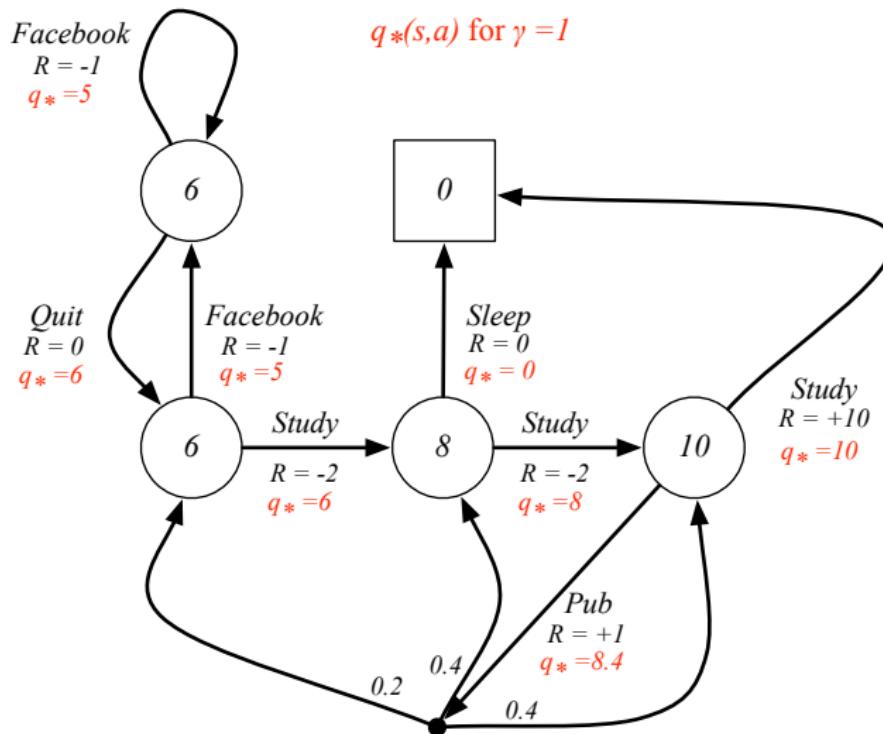
$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a)$$

- The optimal value function specifies the best possible performance in the MDP.
- An MDP is “solved” when we know the optimal value fn.

Example: Optimal Value Function for Student MDP



Example: Optimal Action-Value Function for Student MDP



Optimal Policy

Define a partial ordering over policies

$$\pi \geq \pi' \text{ if } v_\pi(s) \geq v_{\pi'}(s), \forall s$$

Theorem

For any Markov Decision Process

- *There exists an optimal policy π_* that is better than or equal to all other policies, $\pi_* \geq \pi, \forall \pi$*
- *All optimal policies achieve the optimal value function, $v_{\pi_*}(s) = v_*(s)$*
- *All optimal policies achieve the optimal action-value function, $q_{\pi_*}(s, a) = q_*(s, a)$*

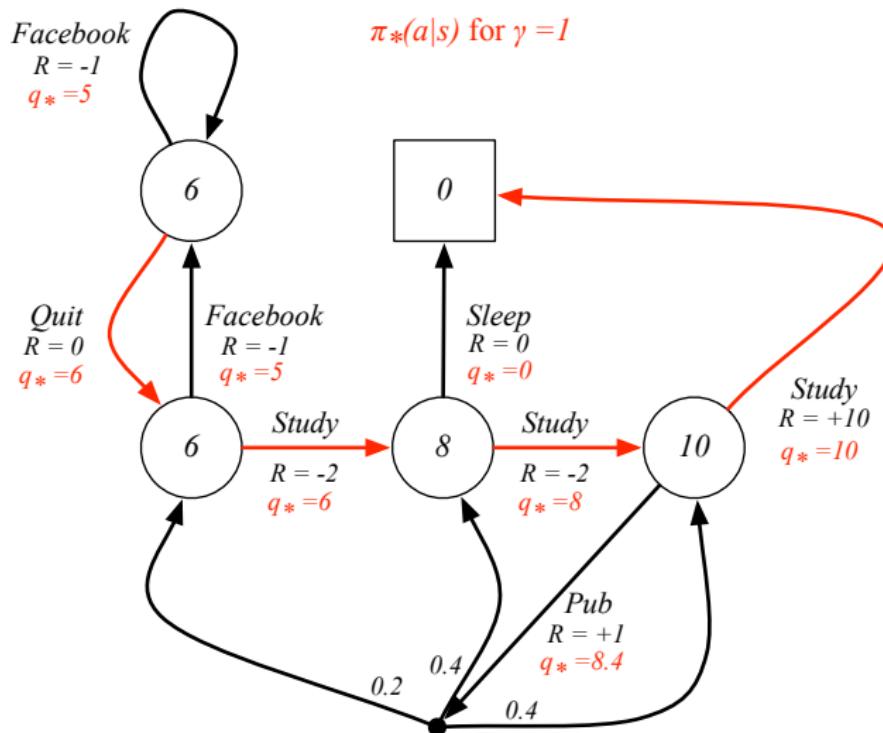
Finding an Optimal Policy

An optimal policy can be found by maximising over $q_*(s, a)$,

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}} q_*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

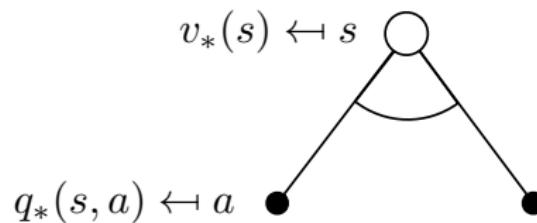
- There is always a deterministic optimal policy for any MDP
- If we know $q_*(s, a)$, we immediately have the optimal policy

Example: Optimal Policy for Student MDP



Bellman Optimality Equation for v_*

The optimal value functions are recursively related by the Bellman optimality equations:



$$v_*(s) = \max_a q_*(s, a)$$

Bellman Optimality Equation for Q^*

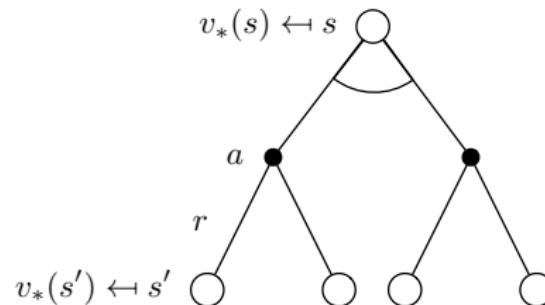
$$q_*(s, a) \leftarrow s, a$$

r

$$v_*(s') \leftarrow s'$$

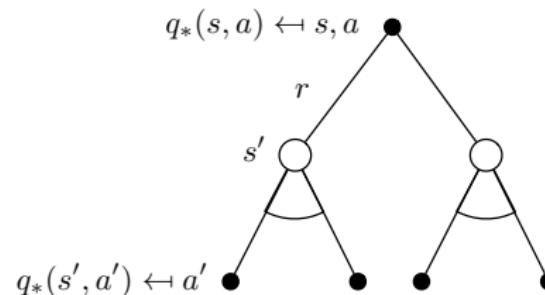
$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

Bellman Optimality Equation for V^* (2)



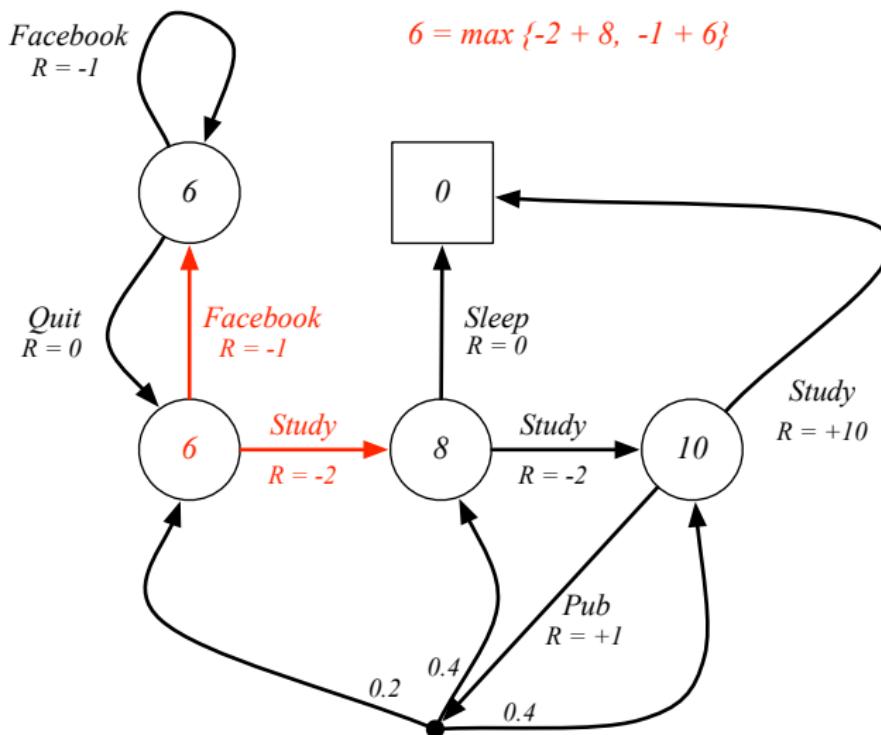
$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

Bellman Optimality Equation for Q^* (2)



$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a')$$

Example: Bellman Optimality Equation in Student MDP



Solving the Bellman Optimality Equation

- Bellman Optimality Equation is non-linear
- No closed form solution (in general)
- Many iterative solution methods
 - Value Iteration
 - Policy Iteration
 - Q-learning
 - Sarsa

Extensions to MDPs

(no exam)

- Infinite and continuous MDPs
- Partially observable MDPs
- Undiscounted, average reward MDPs

Infinite MDPs

(no exam)

The following extensions are all possible:

- Countably infinite state and/or action spaces
 - Straightforward
- Continuous state and/or action spaces
 - Closed form for linear quadratic model (LQR)
- Continuous time
 - Requires partial differential equations
 - Hamilton-Jacobi-Bellman (HJB) equation
 - Limiting case of Bellman equation as time-step $\rightarrow 0$

POMDPs

(no exam)

A Partially Observable Markov Decision Process is an MDP with hidden states. It is a hidden Markov model with actions.

Definition

A POMDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma \rangle$

- \mathcal{S} is a finite set of states
- \mathcal{A} is a finite set of actions
- \mathcal{O} is a finite set of observations
- \mathcal{P} is a state transition probability matrix,
$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$$
- \mathcal{R} is a reward function, $\mathcal{R}_s^a = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$
- \mathcal{Z} is an observation function,
$$\mathcal{Z}_{s'o}^a = \mathbb{P}[O_{t+1} = o \mid S_{t+1} = s', A_t = a]$$
- γ is a discount factor $\gamma \in [0, 1]$.

Belief States

(no exam)

Definition

A *history* H_t is a sequence of actions, observations and rewards,

$$H_t = A_0, O_1, R_1, \dots, A_{t-1}, O_t, R_t$$

Definition

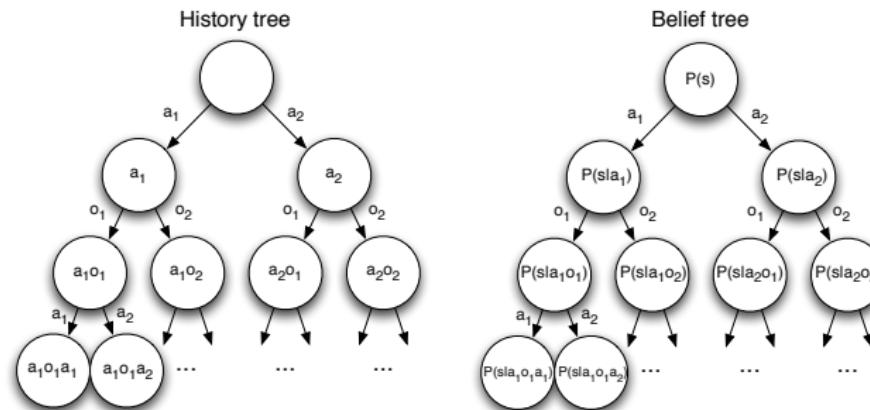
A *belief state* $b(h)$ is a probability distribution over states, conditioned on the history h

$$b(h) = (\mathbb{P}[S_t = s^1 \mid H_t = h], \dots, \mathbb{P}[S_t = s^n \mid H_t = h])$$

Reductions of POMDPs

(no exam)

- The history H_t satisfies the Markov property
- The belief state $b(H_t)$ satisfies the Markov property



- A POMDP can be reduced to an (infinite) history tree
- A POMDP can be reduced to an (infinite) belief state tree

Ergodic Markov Process

(no exam)

An ergodic Markov process is

- *Recurrent*: each state is visited an infinite number of times
- *Aperiodic*: each state is visited without any systematic period

Theorem

An ergodic Markov process has a limiting stationary distribution $d^\pi(s)$ with the property

$$d^\pi(s) = \sum_{s' \in \mathcal{S}} d^\pi(s') \mathcal{P}_{s's}$$

Ergodic MDP

(no exam)

Definition

An MDP is ergodic if the Markov chain induced by any policy is ergodic.

For any policy π , an ergodic MDP has an *average reward per time-step* ρ^π that is independent of start state.

$$\rho^\pi = \lim_{T \rightarrow \infty} \frac{1}{T} \mathbb{E} \left[\sum_{t=1}^T R_t \right]$$

Average Reward Value Function

(no exam)

- The value function of an undiscounted, ergodic MDP can be expressed in terms of average reward.
- $\tilde{v}_\pi(s)$ is the extra reward due to starting from state s ,

$$\tilde{v}_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=1}^{\infty} (R_{t+k} - \rho^\pi) \mid S_t = s \right]$$

There is a corresponding average reward Bellman equation,

$$\begin{aligned}\tilde{v}_\pi(s) &= \mathbb{E}_\pi \left[(R_{t+1} - \rho^\pi) + \sum_{k=1}^{\infty} (R_{t+k+1} - \rho^\pi) \mid S_t = s \right] \\ &= \mathbb{E}_\pi [(R_{t+1} - \rho^\pi) + \tilde{v}_\pi(S_{t+1}) \mid S_t = s]\end{aligned}$$

Questions?

The only stupid question is the one you were afraid to ask but never did.

-Rich Sutton

Lecture 3: Planning by Dynamic Programming

David Silver

Outline

- 1 Introduction
- 2 Policy Evaluation
- 3 Policy Iteration
- 4 Value Iteration
- 5 Extensions to Dynamic Programming
- 6 Contraction Mapping

What is Dynamic Programming?

Dynamic sequential or temporal component to the problem

Programming optimising a “program”, i.e. a policy

- c.f. linear programming
- A method for solving complex problems
- By breaking them down into subproblems
 - Solve the subproblems
 - Combine solutions to subproblems

Requirements for Dynamic Programming

Dynamic Programming is a very general solution method for problems which have two properties:

- Optimal substructure
 - *Principle of optimality* applies
 - Optimal solution can be decomposed into subproblems
- Overlapping subproblems
 - Subproblems recur many times
 - Solutions can be cached and reused
- Markov decision processes satisfy both properties
 - Bellman equation gives recursive decomposition
 - Value function stores and reuses solutions

Planning by Dynamic Programming

- Dynamic programming assumes full knowledge of the MDP
- It is used for *planning* in an MDP
- For prediction:
 - Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and policy π
 - or: MRP $\langle \mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma \rangle$
 - Output: value function v_π
- Or for control:
 - Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
 - Output: optimal value function v_*
 - and: optimal policy π_*

Other Applications of Dynamic Programming

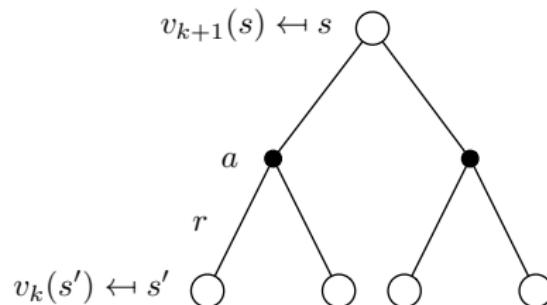
Dynamic programming is used to solve many other problems, e.g.

- Scheduling algorithms
- String algorithms (e.g. sequence alignment)
- Graph algorithms (e.g. shortest path algorithms)
- Graphical models (e.g. Viterbi algorithm)
- Bioinformatics (e.g. lattice models)

Iterative Policy Evaluation

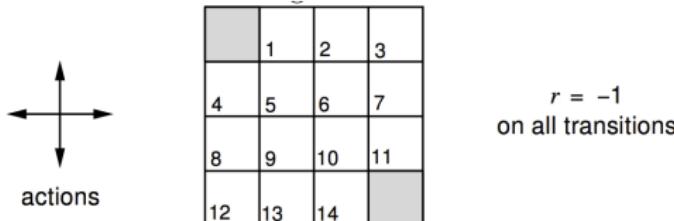
- Problem: evaluate a given policy π
- Solution: iterative application of Bellman expectation backup
- $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_\pi$
- Using *synchronous* backups,
 - At each iteration $k + 1$
 - For all states $s \in \mathcal{S}$
 - Update $v_{k+1}(s)$ from $v_k(s')$
 - where s' is a successor state of s
- We will discuss *asynchronous* backups later
- Convergence to v_π will be proven at the end of the lecture

Iterative Policy Evaluation (2)



$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$
$$\mathbf{v}^{k+1} = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi \mathbf{v}^k$$

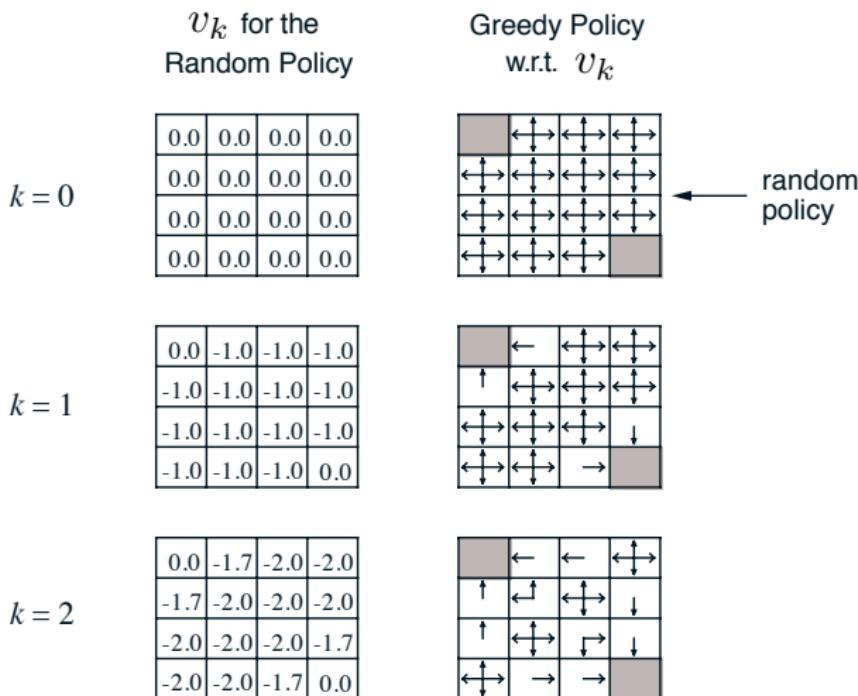
Evaluating a Random Policy in the Small Gridworld



- Undiscounted episodic MDP ($\gamma = 1$)
- Nonterminal states 1, ..., 14
- One terminal state (shown twice as shaded squares)
- Actions leading out of the grid leave state unchanged
- Reward is -1 until the terminal state is reached
- Agent follows uniform random policy

$$\pi(n|\cdot) = \pi(e|\cdot) = \pi(s|\cdot) = \pi(w|\cdot) = 0.25$$

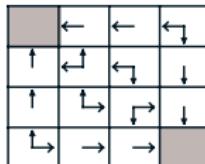
Iterative Policy Evaluation in Small Gridworld



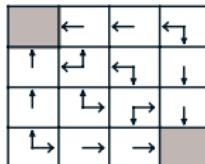
Iterative Policy Evaluation in Small Gridworld (2)

 $k = 3$

| | | | |
|------|------|------|------|
| 0.0 | -2.4 | -2.9 | -3.0 |
| -2.4 | -2.9 | -3.0 | -2.9 |
| -2.9 | -3.0 | -2.9 | -2.4 |
| -3.0 | -2.9 | -2.4 | 0.0 |

 $k = 10$

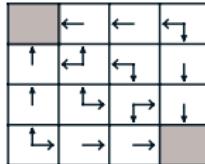
| | | | |
|------|------|------|------|
| 0.0 | -6.1 | -8.4 | -9.0 |
| -6.1 | -7.7 | -8.4 | -8.4 |
| -8.4 | -8.4 | -7.7 | -6.1 |
| -9.0 | -8.4 | -6.1 | 0.0 |



optimal policy

 $k = \infty$

| | | | |
|------|------|------|------|
| 0.0 | -14. | -20. | -22. |
| -14. | -18. | -20. | -20. |
| -20. | -20. | -18. | -14. |
| -22. | -20. | -14. | 0.0 |



How to Improve a Policy

- Given a policy π
 - Evaluate the policy π

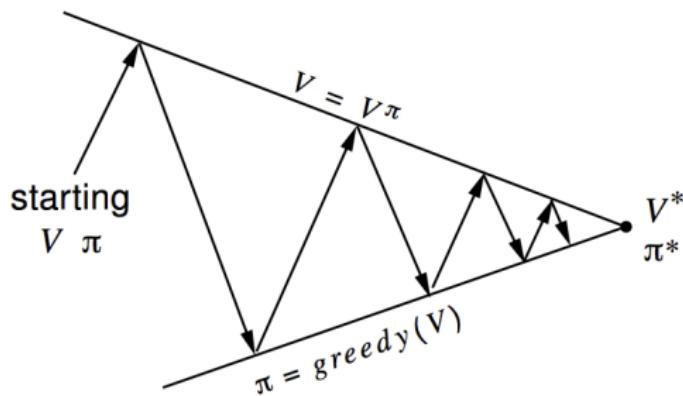
$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots | S_t = s]$$

- Improve the policy by acting greedily with respect to v_π

$$\pi' = \text{greedy}(v_\pi)$$

- In Small Gridworld improved policy was optimal, $\pi' = \pi^*$
- In general, need more iterations of improvement / evaluation
- But this process of policy iteration always converges to π^*

Policy Iteration

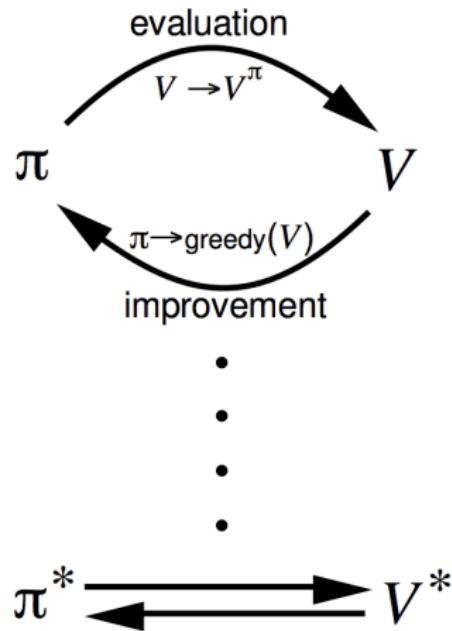


Policy evaluation Estimate v_π

Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$

Greedy policy improvement

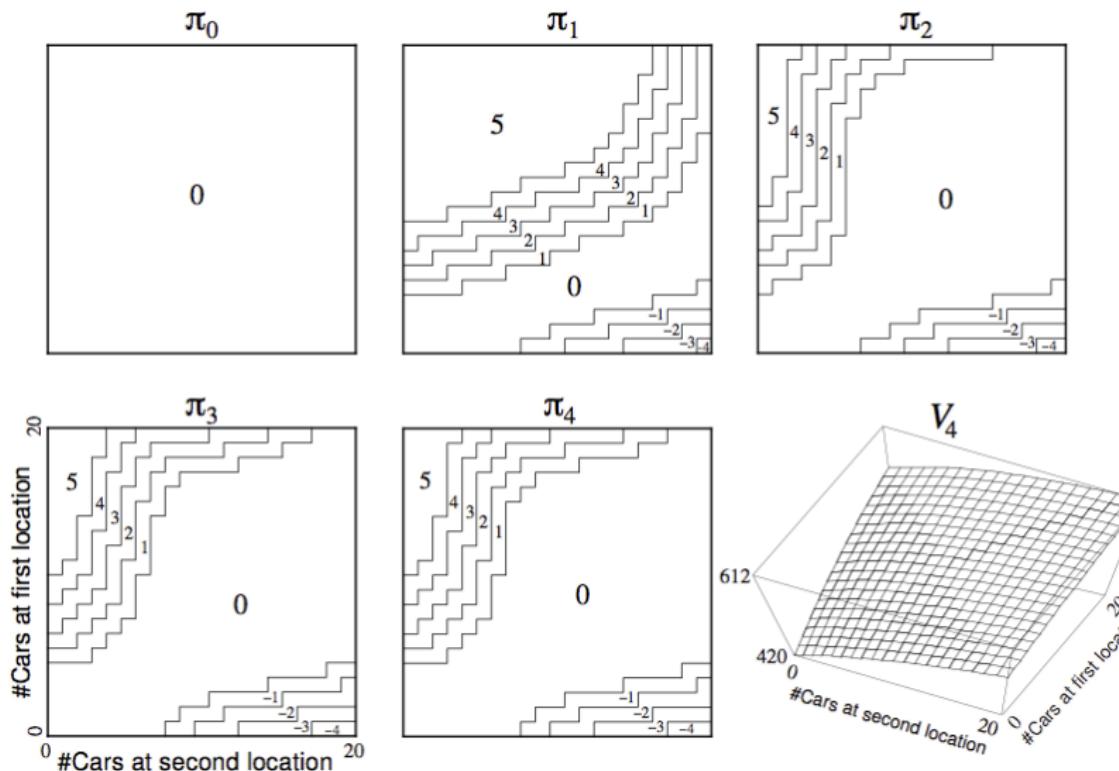


Jack's Car Rental



- States: Two locations, maximum of 20 cars at each
- Actions: Move up to 5 cars between locations overnight
- Reward: \$10 for each car rented (must be available)
- Transitions: Cars returned and requested randomly
 - Poisson distribution, n returns/requests with prob $\frac{\lambda^n}{n!} e^{-\lambda}$
 - 1st location: average requests = 3, average returns = 3
 - 2nd location: average requests = 4, average returns = 2

Policy Iteration in Jack's Car Rental



Policy Improvement

- Consider a deterministic policy, $a = \pi(s)$
- We can *improve* the policy by acting greedily

$$\pi'(s) = \operatorname{argmax}_{a \in \mathcal{A}} q_\pi(s, a)$$

- This improves the value from any state s over one step,

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$$

- It therefore improves the value function, $v_{\pi'}(s) \geq v_\pi(s)$

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) = \mathbb{E}_{\pi'} [R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'} [R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'} [R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'} [R_{t+1} + \gamma R_{t+2} + \dots \mid S_t = s] = v_{\pi'}(s) \end{aligned}$$

Policy Improvement (2)

- If improvements stop,

$$q_{\pi}(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_{\pi}(s, a) = q_{\pi}(s, \pi(s)) = v_{\pi}(s)$$

- Then the Bellman optimality equation has been satisfied

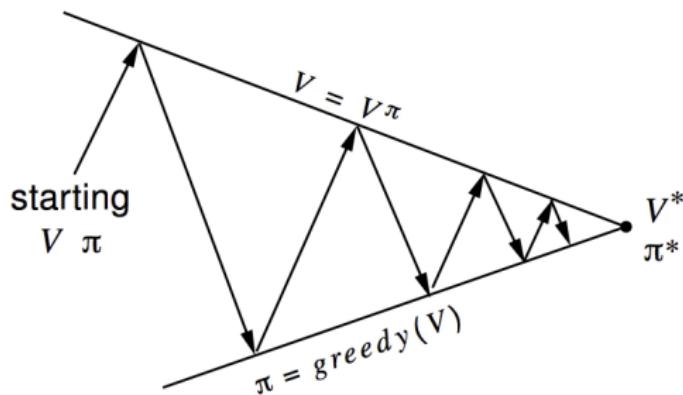
$$v_{\pi}(s) = \max_{a \in \mathcal{A}} q_{\pi}(s, a)$$

- Therefore $v_{\pi}(s) = v_*(s)$ for all $s \in \mathcal{S}$
- so π is an optimal policy

Modified Policy Iteration

- Does policy evaluation need to converge to v_π ?
- Or should we introduce a stopping condition
 - e.g. ϵ -convergence of value function
- Or simply stop after k iterations of iterative policy evaluation?
- For example, in the small gridworld $k = 3$ was sufficient to achieve optimal policy
- Why not update policy every iteration? i.e. stop after $k = 1$
 - This is equivalent to *value iteration* (next section)

Generalised Policy Iteration

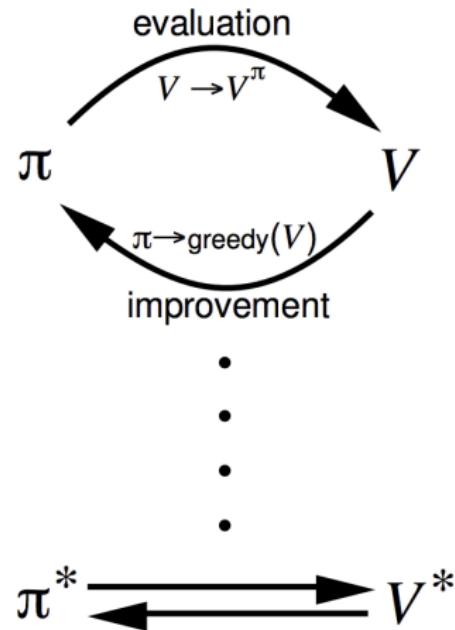


Policy evaluation Estimate v_π

Any policy evaluation algorithm

Policy improvement Generate $\pi' \geq \pi$

Any policy improvement algorithm



Principle of Optimality

Any optimal policy can be subdivided into two components:

- An optimal first action A_*
- Followed by an optimal policy from successor state S'

Theorem (Principle of Optimality)

A policy $\pi(a|s)$ achieves the optimal value from state s ,

$v_\pi(s) = v_*(s)$, if and only if

- *For any state s' reachable from s*
- π *achieves the optimal value from state s' , $v_\pi(s') = v_*(s')$*

Deterministic Value Iteration

- If we know the solution to subproblems $v_*(s')$
- Then solution $v_*(s)$ can be found by one-step lookahead

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

- The idea of value iteration is to apply these updates iteratively
- Intuition: start with final rewards and work backwards
- Still works with loopy, stochastic MDPs

Lecture 3: Planning by Dynamic Programming

└ Value Iteration

└ Value Iteration in MDPs

Example: Shortest Path

| | | | | |
|---|--|--|--|--|
| g | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Problem

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

V_1

| | | | |
|----|----|----|----|
| 0 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 |

V_2

| | | | |
|----|----|----|----|
| 0 | -1 | -2 | -2 |
| -1 | -2 | -2 | -2 |
| -2 | -2 | -2 | -2 |
| -2 | -2 | -2 | -2 |

V_3

| | | | |
|----|----|----|----|
| 0 | -1 | -2 | -3 |
| -1 | -2 | -3 | -3 |
| -2 | -3 | -3 | -3 |
| -3 | -3 | -3 | -3 |

V_4

| | | | |
|----|----|----|----|
| 0 | -1 | -2 | -3 |
| -1 | -2 | -3 | -4 |
| -2 | -3 | -4 | -4 |
| -3 | -4 | -4 | -4 |

V_5

| | | | |
|----|----|----|----|
| 0 | -1 | -2 | -3 |
| -1 | -2 | -3 | -4 |
| -2 | -3 | -4 | -5 |
| -3 | -4 | -5 | -5 |

V_6

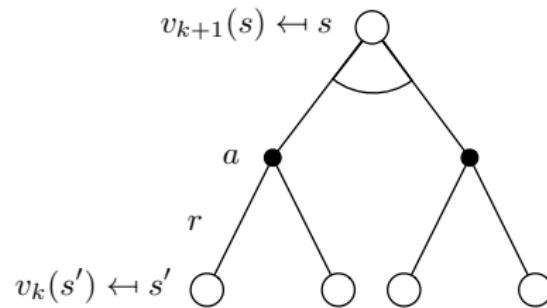
| | | | |
|----|----|----|----|
| 0 | -1 | -2 | -3 |
| -1 | -2 | -3 | -4 |
| -2 | -3 | -4 | -5 |
| -3 | -4 | -5 | -6 |

V_7

Value Iteration

- Problem: find optimal policy π
- Solution: iterative application of Bellman optimality backup
- $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_*$
- Using synchronous backups
 - At each iteration $k + 1$
 - For all states $s \in S$
 - Update $v_{k+1}(s)$ from $v_k(s')$
- Convergence to v_* will be proven later
- Unlike policy iteration, there is no explicit policy
- Intermediate value functions may not correspond to any policy

Value Iteration (2)



$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$$\mathbf{v}_{k+1} = \max_{a \in \mathcal{A}} \mathcal{R}^a + \gamma \mathcal{P}^a \mathbf{v}_k$$

Example of Value Iteration in Practice

<http://www.cs.ubc.ca/~poole/demos/mdp/vi.html>

Synchronous Dynamic Programming Algorithms

| Problem | Bellman Equation | Algorithm |
|------------|---|-----------------------------|
| Prediction | Bellman Expectation Equation | Iterative Policy Evaluation |
| Control | Bellman Expectation Equation + Greedy Policy Improvement | Policy Iteration |
| Control | Bellman Optimality Equation | Value Iteration |

- Algorithms are based on state-value function $v_\pi(s)$ or $v_*(s)$
- Complexity $O(mn^2)$ per iteration, for m actions and n states
- Could also apply to action-value function $q_\pi(s, a)$ or $q_*(s, a)$
- Complexity $O(m^2n^2)$ per iteration

Asynchronous Dynamic Programming

- DP methods described so far used *synchronous* backups
- i.e. all states are backed up in parallel
- *Asynchronous DP* backs up states individually, in any order
- For each selected state, apply the appropriate backup
- Can significantly reduce computation
- Guaranteed to converge if all states continue to be selected

Asynchronous Dynamic Programming

Three simple ideas for asynchronous dynamic programming:

- *In-place* dynamic programming
- *Prioritised sweeping*
- *Real-time* dynamic programming

In-Place Dynamic Programming

- Synchronous value iteration stores two copies of value function
for all s in \mathcal{S}

$$v_{\text{new}}(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_{\text{old}}(s') \right)$$

$$v_{\text{old}} \leftarrow v_{\text{new}}$$

- In-place value iteration only stores one copy of value function
for all s in \mathcal{S}

$$v(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right)$$

Prioritised Sweeping

- Use magnitude of Bellman error to guide state selection, e.g.

$$\left| \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right) - v(s) \right|$$

- Backup the state with the largest remaining Bellman error
- Update Bellman error of affected states after each backup
- Requires knowledge of reverse dynamics (predecessor states)
- Can be implemented efficiently by maintaining a priority queue

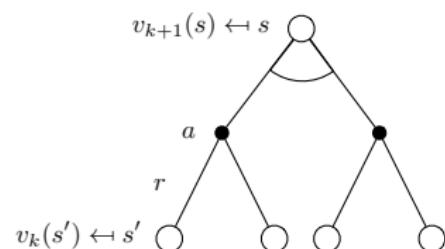
Real-Time Dynamic Programming

- Idea: only states that are relevant to agent
- Use agent's experience to guide the selection of states
- After each time-step S_t, A_t, R_{t+1}
- Backup the state S_t

$$v(S_t) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_{S_t}^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{S_t s'}^a v(s') \right)$$

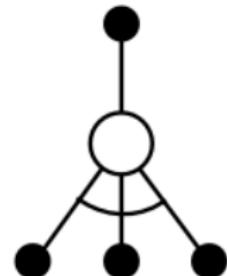
Full-Width Backups

- DP uses *full-width* backups
- For each backup (sync or async)
 - Every successor state and action is considered
 - Using knowledge of the MDP transitions and reward function
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers Bellman's *curse of dimensionality*
 - Number of states $n = |\mathcal{S}|$ grows exponentially with number of state variables
- Even one backup can be too expensive



Sample Backups

- In subsequent lectures we will consider *sample backups*
- Using sample rewards and sample transitions
 $\langle S, A, R, S' \rangle$
- Instead of reward function \mathcal{R} and transition dynamics \mathcal{P}
- Advantages:
 - Model-free: no advance knowledge of MDP required
 - Breaks the curse of dimensionality through sampling
 - Cost of backup is constant, independent of $n = |\mathcal{S}|$



Approximate Dynamic Programming

- Approximate the value function
- Using a *function approximator* $\hat{v}(s, \mathbf{w})$
- Apply dynamic programming to $\hat{v}(\cdot, \mathbf{w})$
- e.g. Fitted Value Iteration repeats at each iteration k ,
 - Sample states $\tilde{\mathcal{S}} \subseteq \mathcal{S}$
 - For each state $s \in \tilde{\mathcal{S}}$, estimate target value using Bellman optimality equation,

$$\tilde{v}_k(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \hat{v}(s', \mathbf{w}_k) \right)$$

- Train next value function $\hat{v}(\cdot, \mathbf{w}_{k+1})$ using targets $\{\langle s, \tilde{v}_k(s) \rangle\}$

Some Technical Questions

- How do we know that value iteration converges to v_* ?
- Or that iterative policy evaluation converges to v_π ?
- And therefore that policy iteration converges to v_* ?
- Is the solution unique?
- How fast do these algorithms converge?
- These questions are resolved by *contraction mapping theorem*

Value Function Space

- Consider the vector space \mathcal{V} over value functions
- There are $|\mathcal{S}|$ dimensions
- Each point in this space fully specifies a value function $v(s)$
- What does a Bellman backup do to points in this space?
- We will show that it brings value functions *closer*
- And therefore the backups must converge on a unique solution

Value Function ∞ -Norm

- We will measure distance between state-value functions u and v by the ∞ -norm
- i.e. the largest difference between state values,

$$\|u - v\|_\infty = \max_{s \in \mathcal{S}} |u(s) - v(s)|$$

Bellman Expectation Backup is a Contraction

- Define the *Bellman expectation backup operator* T^π ,

$$T^\pi(v) = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v$$

- This operator is a γ -contraction, i.e. it makes value functions closer by at least γ ,

$$\begin{aligned} \|T^\pi(u) - T^\pi(v)\|_\infty &= \|(\mathcal{R}^\pi + \gamma \mathcal{P}^\pi u) - (\mathcal{R}^\pi + \gamma \mathcal{P}^\pi v)\|_\infty \\ &= \|\gamma \mathcal{P}^\pi(u - v)\|_\infty \\ &\leq \|\gamma \mathcal{P}^\pi\| \|u - v\|_\infty \\ &\leq \gamma \|u - v\|_\infty \end{aligned}$$

Contraction Mapping Theorem

Theorem (Contraction Mapping Theorem)

For any metric space \mathcal{V} that is complete (i.e. closed) under an operator $T(v)$, where T is a γ -contraction,

- *T converges to a unique fixed point*
- *At a linear convergence rate of γ*

Convergence of Iter. Policy Evaluation and Policy Iteration

- The Bellman expectation operator T^π has a unique fixed point
- v_π is a fixed point of T^π (by Bellman expectation equation)
- By contraction mapping theorem
- Iterative policy evaluation converges on v_π
- Policy iteration converges on v_*

Bellman Optimality Backup is a Contraction

- Define the *Bellman optimality backup operator* T^* ,

$$T^*(v) = \max_{a \in \mathcal{A}} \mathcal{R}^a + \gamma \mathcal{P}^a v$$

- This operator is a γ -contraction, i.e. it makes value functions closer by at least γ (similar to previous proof)

$$\|T^*(u) - T^*(v)\|_\infty \leq \gamma \|u - v\|_\infty$$

Convergence of Value Iteration

- The Bellman optimality operator T^* has a unique fixed point
- v_* is a fixed point of T^* (by Bellman optimality equation)
- By contraction mapping theorem
- Value iteration converges on v_*

Lecture 4: Model-Free Prediction

David Silver

Outline

- 1 Introduction
- 2 Monte-Carlo Learning
- 3 Temporal-Difference Learning
- 4 TD(λ)

Model-Free Reinforcement Learning

- Last lecture:
 - Planning by dynamic programming
 - Solve a *known* MDP
- This lecture:
 - Model-free prediction
 - Estimate the value function of an *unknown* MDP
- Next lecture:
 - Model-free control
 - Optimise the value function of an *unknown* MDP

Monte-Carlo Reinforcement Learning

- MC methods learn directly from episodes of experience
- MC is *model-free*: no knowledge of MDP transitions / rewards
- MC learns from *complete* episodes: no bootstrapping
- MC uses the simplest possible idea: value = mean return
- Caveat: can only apply MC to *episodic* MDPs
 - All episodes must terminate

Monte-Carlo Policy Evaluation

- Goal: learn v_π from episodes of experience under policy π

$$S_1, A_1, R_2, \dots, S_k \sim \pi$$

- Recall that the *return* is the total discounted reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$$

- Recall that the value function is the expected return:

$$v_\pi(s) = \mathbb{E}_\pi [G_t \mid S_t = s]$$

- Monte-Carlo policy evaluation uses *empirical mean* return instead of *expected* return

First-Visit Monte-Carlo Policy Evaluation

- To evaluate state s
- The **first** time-step t that state s is visited in an episode,
- Increment counter $N(s) \leftarrow N(s) + 1$
- Increment total return $S(s) \leftarrow S(s) + G_t$
- Value is estimated by mean return $V(s) = S(s)/N(s)$
- By law of large numbers, $V(s) \rightarrow v_\pi(s)$ as $N(s) \rightarrow \infty$

Every-Visit Monte-Carlo Policy Evaluation

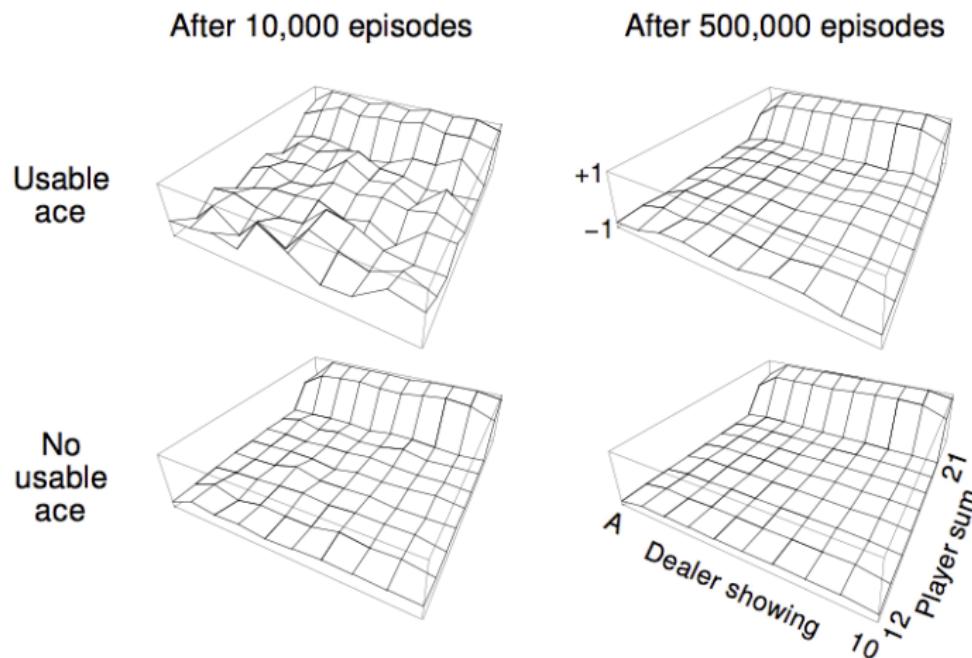
- To evaluate state s
- **Every** time-step t that state s is visited in an episode,
- Increment counter $N(s) \leftarrow N(s) + 1$
- Increment total return $S(s) \leftarrow S(s) + G_t$
- Value is estimated by mean return $V(s) = S(s)/N(s)$
- Again, $V(s) \rightarrow v_\pi(s)$ as $N(s) \rightarrow \infty$

Blackjack Example

- States (200 of them):
 - Current sum (12-21)
 - Dealer's showing card (ace-10)
 - Do I have a "useable" ace? (yes-no)
- Action **stick**: Stop receiving cards (and terminate)
- Action **twist**: Take another card (no replacement)
- Reward for **stick**:
 - +1 if sum of cards > sum of dealer cards
 - 0 if sum of cards = sum of dealer cards
 - -1 if sum of cards < sum of dealer cards
- Reward for **twist**:
 - -1 if sum of cards > 21 (and terminate)
 - 0 otherwise
- Transitions: automatically **twist** if sum of cards < 12



Blackjack Value Function after Monte-Carlo Learning



Policy: **stick** if sum of cards ≥ 20 , otherwise **twist**

Incremental Mean

The mean μ_1, μ_2, \dots of a sequence x_1, x_2, \dots can be computed incrementally,

$$\begin{aligned}\mu_k &= \frac{1}{k} \sum_{j=1}^k x_j \\ &= \frac{1}{k} \left(x_k + \sum_{j=1}^{k-1} x_j \right) \\ &= \frac{1}{k} (x_k + (k-1)\mu_{k-1}) \\ &= \mu_{k-1} + \frac{1}{k} (x_k - \mu_{k-1})\end{aligned}$$

Incremental Monte-Carlo Updates

- Update $V(s)$ incrementally after episode $S_1, A_1, R_2, \dots, S_T$
- For each state S_t with return G_t

$$N(S_t) \leftarrow N(S_t) + 1$$

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)} (G_t - V(S_t))$$

- In non-stationary problems, it can be useful to track a running mean, i.e. forget old episodes.

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

Temporal-Difference Learning

- TD methods learn directly from episodes of experience
- TD is *model-free*: no knowledge of MDP transitions / rewards
- TD learns from *incomplete* episodes, by *bootstrapping*
- TD updates a guess towards a guess

MC and TD

- Goal: learn v_π online from experience under policy π
- Incremental every-visit Monte-Carlo
 - Update value $V(S_t)$ toward *actual* return G_t

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

- Simplest temporal-difference learning algorithm: TD(0)
 - Update value $V(S_t)$ toward *estimated* return $R_{t+1} + \gamma V(S_{t+1})$

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

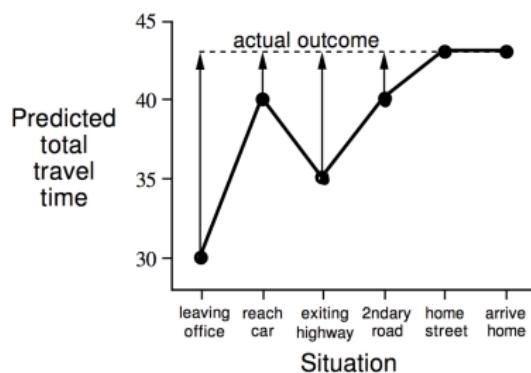
- $R_{t+1} + \gamma V(S_{t+1})$ is called the *TD target*
- $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the *TD error*

Driving Home Example

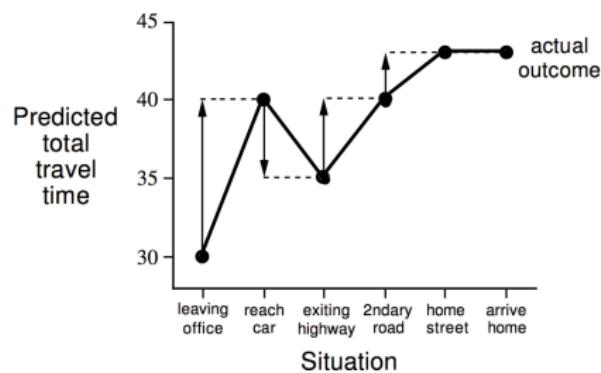
| State | Elapsed Time (minutes) | Predicted Time to Go | Predicted Total Time |
|--------------------|-----------------------------------|---------------------------------|---------------------------------|
| leaving office | 0 | 30 | 30 |
| reach car, raining | 5 | 35 | 40 |
| exit highway | 20 | 15 | 35 |
| behind truck | 30 | 10 | 40 |
| home street | 40 | 3 | 43 |
| arrive home | 43 | 0 | 43 |

Driving Home Example: MC vs. TD

Changes recommended by Monte Carlo methods ($\alpha=1$)



Changes recommended by TD methods ($\alpha=1$)



Advantages and Disadvantages of MC vs. TD

- TD can learn *before* knowing the final outcome
 - TD can learn online after every step
 - MC must wait until end of episode before return is known
- TD can learn *without* the final outcome
 - TD can learn from incomplete sequences
 - MC can only learn from complete sequences
 - TD works in continuing (non-terminating) environments
 - MC only works for episodic (terminating) environments

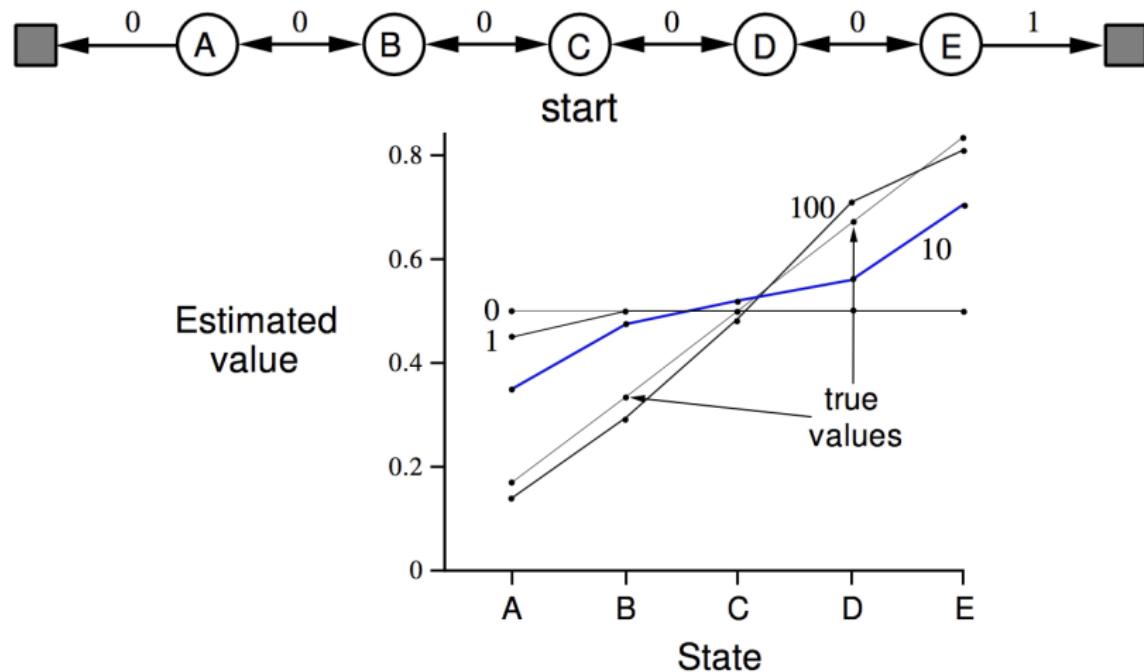
Bias/Variance Trade-Off

- Return $G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$ is *unbiased* estimate of $v_\pi(S_t)$
- True TD target $R_{t+1} + \gamma v_\pi(S_{t+1})$ is *unbiased* estimate of $v_\pi(S_t)$
- TD target $R_{t+1} + \gamma V(S_{t+1})$ is *biased* estimate of $v_\pi(S_t)$
- TD target is much lower variance than the return:
 - Return depends on *many* random actions, transitions, rewards
 - TD target depends on *one* random action, transition, reward

Advantages and Disadvantages of MC vs. TD (2)

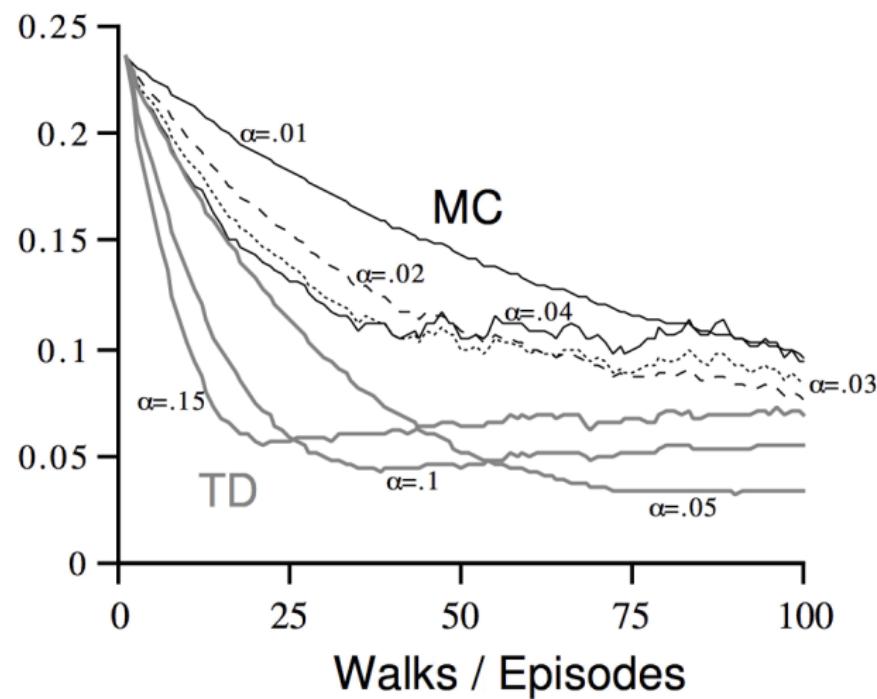
- MC has high variance, zero bias
 - Good convergence properties
 - (even with function approximation)
 - Not very sensitive to initial value
 - Very simple to understand and use
- TD has low variance, some bias
 - Usually more efficient than MC
 - TD(0) converges to $v_\pi(s)$
 - (but not always with function approximation)
 - More sensitive to initial value

Random Walk Example



Random Walk: MC vs. TD

RMS error,
averaged
over states



Batch MC and TD

- MC and TD converge: $V(s) \rightarrow v_\pi(s)$ as experience $\rightarrow \infty$
- But what about batch solution for finite experience?

$$s_1^1, a_1^1, r_2^1, \dots, s_{T_1}^1$$

$$\vdots$$

$$s_1^K, a_1^K, r_2^K, \dots, s_{T_K}^K$$

- e.g. Repeatedly sample episode $k \in [1, K]$
- Apply MC or TD(0) to episode k

AB Example

Two states A, B ; no discounting; 8 episodes of experience

A, 0, B, 0

B, 1

B, 1

B, 1

B, 1

B, 1

B, 1

B, 0

What is $V(A), V(B)$?

AB Example

Two states A, B ; no discounting; 8 episodes of experience

A, 0, B, 0

B, 1

B, 1

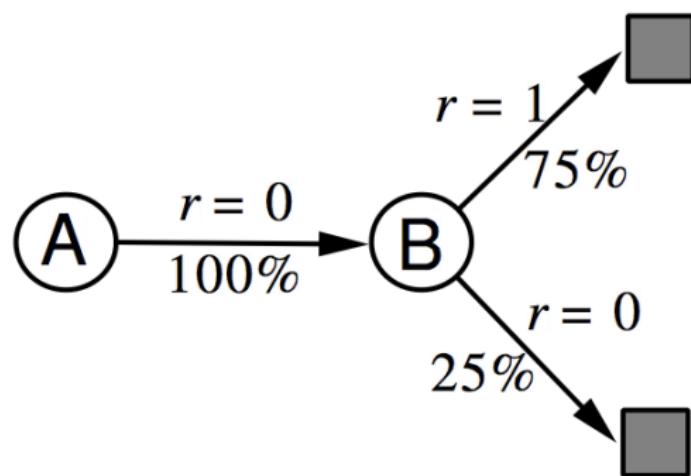
B, 1

B, 1

B, 1

B, 1

B, 0



What is $V(A), V(B)$?

Certainty Equivalence

- MC converges to solution with minimum mean-squared error
 - Best fit to the observed returns

$$\sum_{k=1}^K \sum_{t=1}^{T_k} (G_t^k - V(s_t^k))^2$$

- In the AB example, $V(A) = 0$
- TD(0) converges to solution of max likelihood Markov model
 - Solution to the MDP $\langle \mathcal{S}, \mathcal{A}, \hat{\mathcal{P}}, \hat{\mathcal{R}}, \gamma \rangle$ that best fits the data

$$\hat{\mathcal{P}}_{s,s'}^a = \frac{1}{N(s,a)} \sum_{k=1}^K \sum_{t=1}^{T_k} \mathbf{1}(s_t^k, a_t^k, s_{t+1}^k = s, a, s')$$

$$\hat{\mathcal{R}}_s^a = \frac{1}{N(s,a)} \sum_{k=1}^K \sum_{t=1}^{T_k} \mathbf{1}(s_t^k, a_t^k = s, a) r_t^k$$

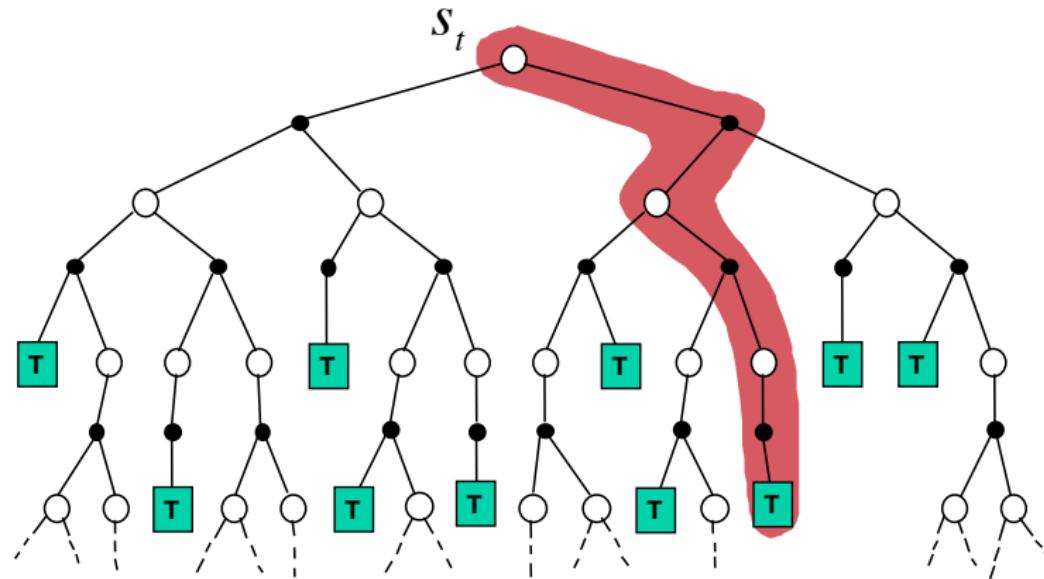
- In the AB example, $V(A) = 0.75$

Advantages and Disadvantages of MC vs. TD (3)

- TD exploits Markov property
 - Usually more efficient in Markov environments
- MC does not exploit Markov property
 - Usually more effective in non-Markov environments

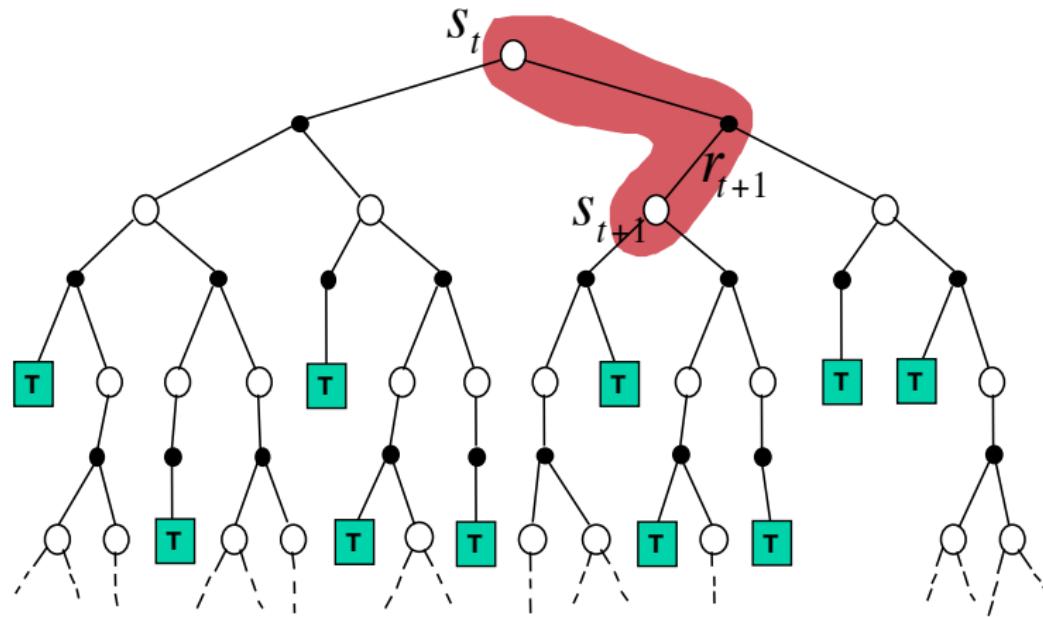
Monte-Carlo Backup

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



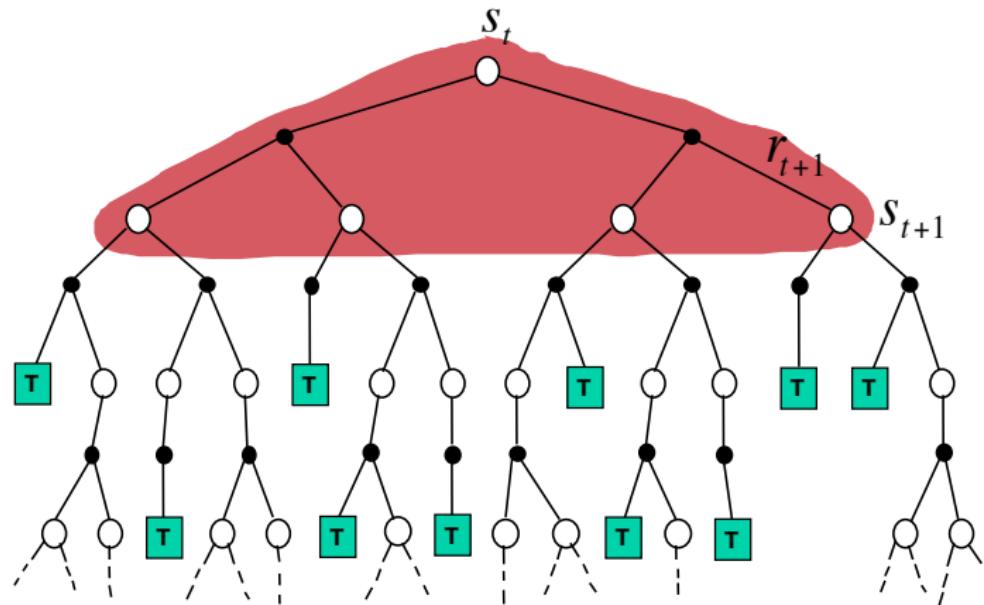
Temporal-Difference Backup

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



Dynamic Programming Backup

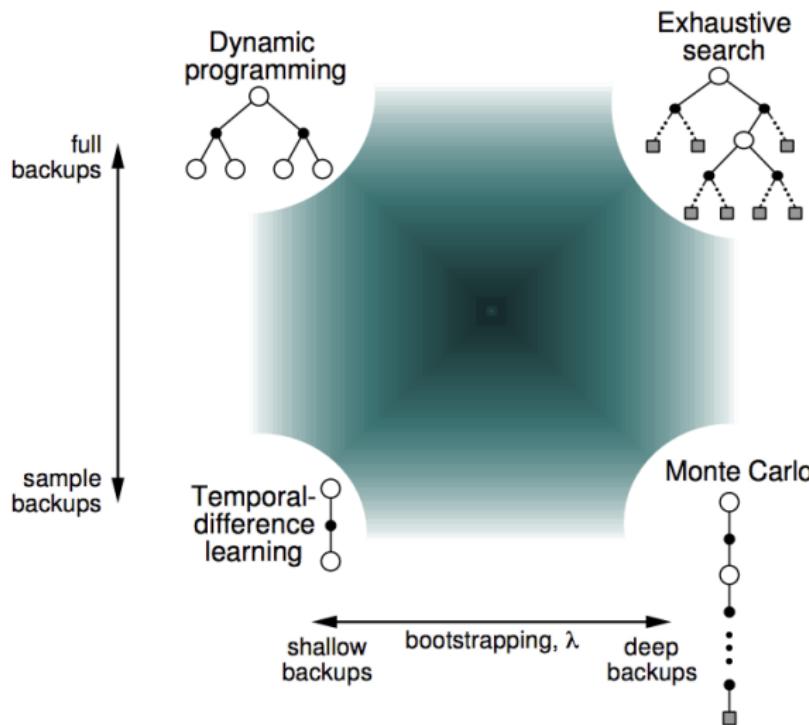
$$V(S_t) \leftarrow \mathbb{E}_\pi [R_{t+1} + \gamma V(S_{t+1})]$$



Bootstrapping and Sampling

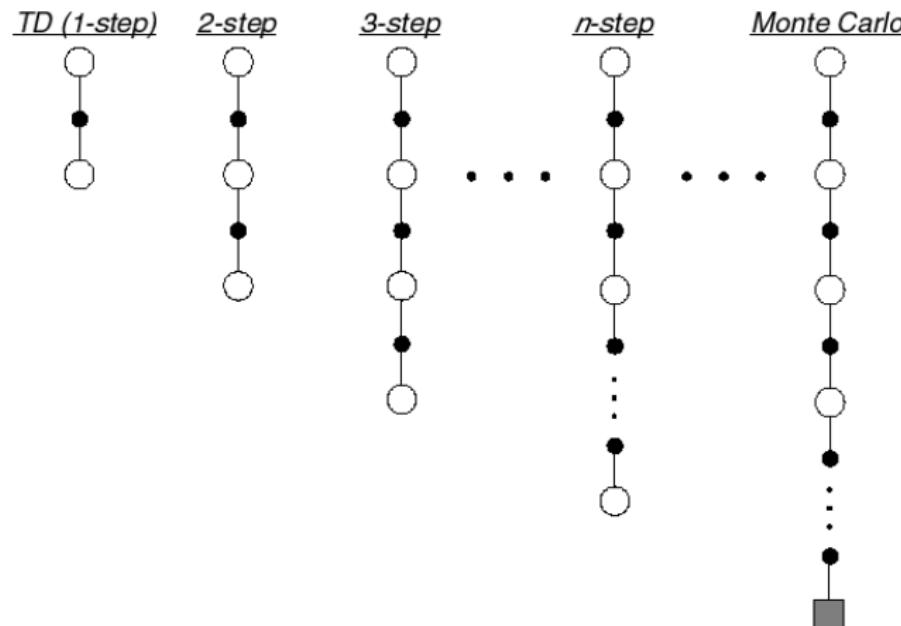
- **Bootstrapping**: update involves an estimate
 - MC does not bootstrap
 - DP bootstraps
 - TD bootstraps
- **Sampling**: update samples an expectation
 - MC samples
 - DP does not sample
 - TD samples

Unified View of Reinforcement Learning



n -Step Prediction

- Let TD target look n steps into the future



n -Step Return

- Consider the following n -step returns for $n = 1, 2, \infty$:

$$\begin{aligned} n = 1 & \quad (\text{TD}) \quad G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1}) \\ n = 2 & \quad G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \\ & \vdots \qquad \vdots \\ n = \infty & \quad (\text{MC}) \quad G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T \end{aligned}$$

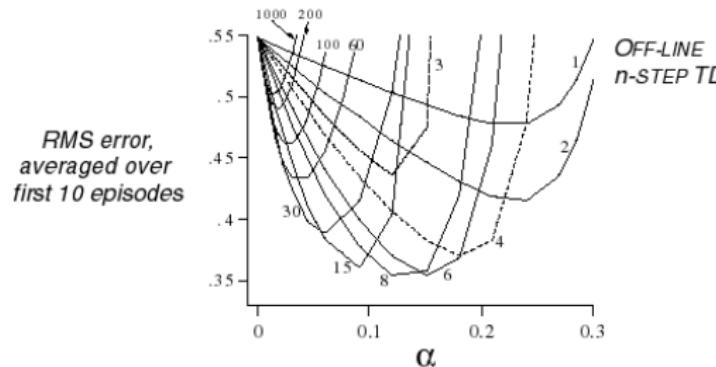
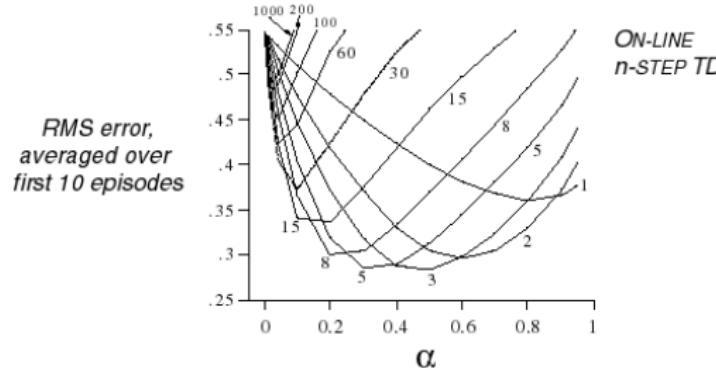
- Define the n -step return

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

- n -step temporal-difference learning

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t^{(n)} - V(S_t) \right)$$

Large Random Walk Example

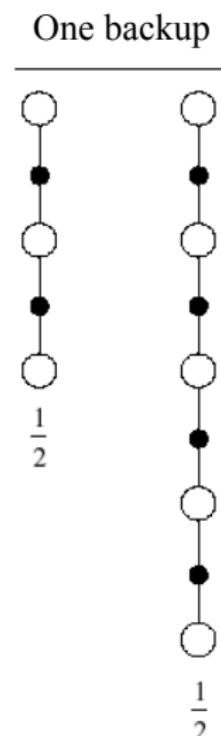


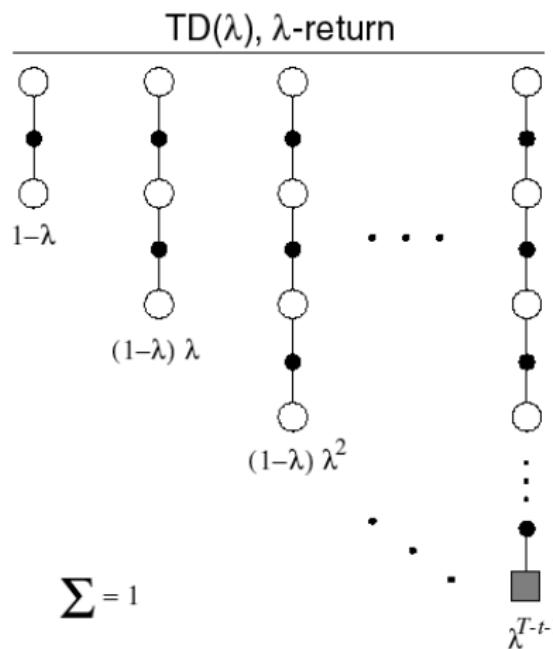
Averaging n -Step Returns

- We can average n -step returns over different n
- e.g. average the 2-step and 4-step returns

$$\frac{1}{2}G^{(2)} + \frac{1}{2}G^{(4)}$$

- Combines information from two different time-steps
- Can we efficiently combine information from all time-steps?



λ -return

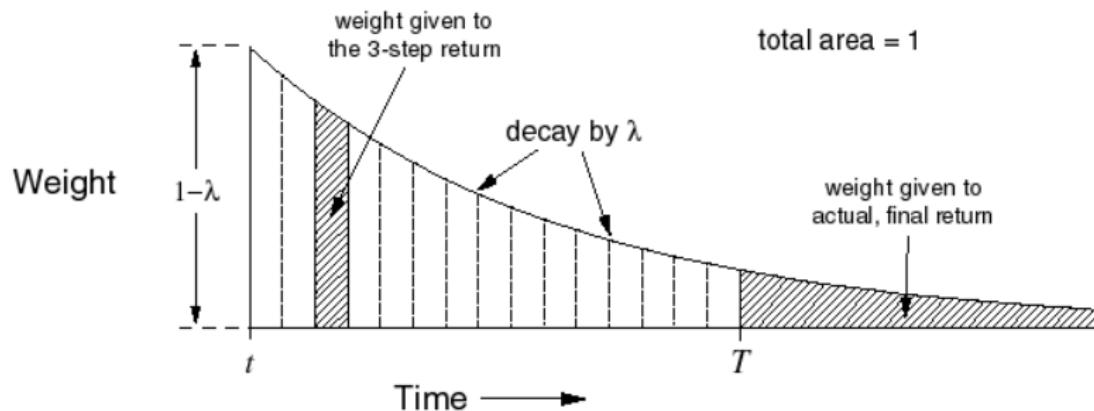
- The λ -return G_t^λ combines all n -step returns $G_t^{(n)}$
- Using weight $(1 - \lambda)\lambda^{n-1}$

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

- Forward-view TD(λ)

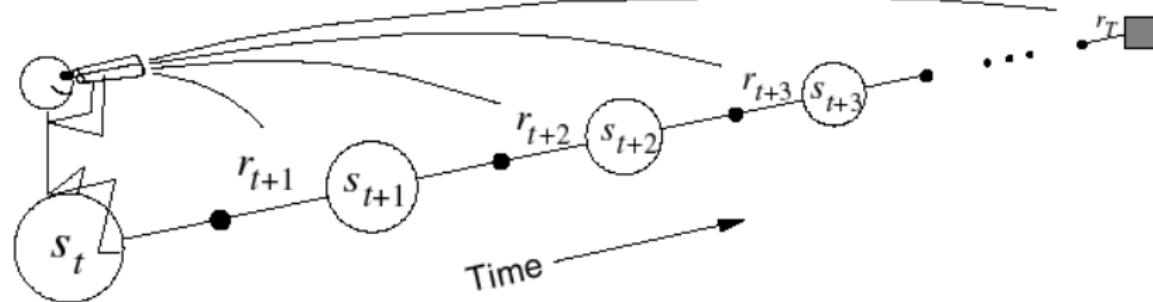
$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t^\lambda - V(S_t) \right)$$

TD(λ) Weighting Function

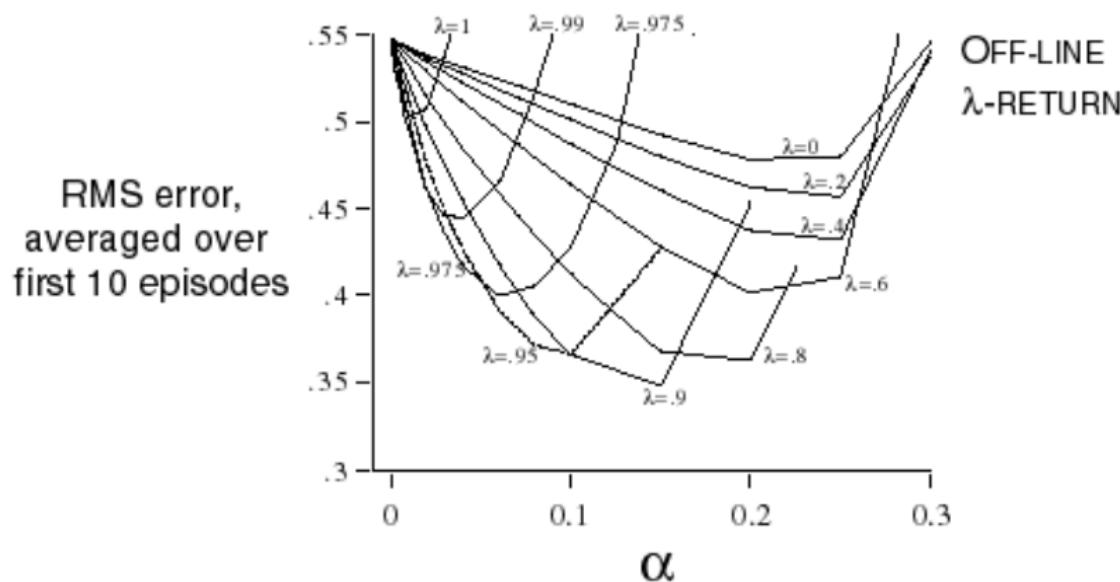


$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

Forward-view TD(λ)



- Update value function towards the λ -return
- Forward-view looks into the future to compute G_t^λ
- Like MC, can only be computed from complete episodes

Forward-View TD(λ) on Large Random Walk

Backward View TD(λ)

- Forward view provides theory
- Backward view provides mechanism
- Update online, every step, from incomplete sequences

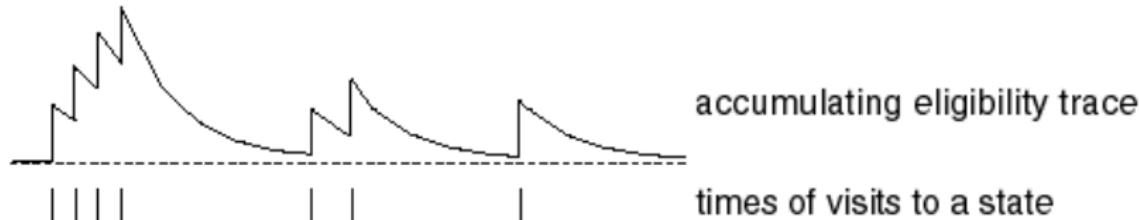
Eligibility Traces



- Credit assignment problem: did bell or light cause shock?
- **Frequency heuristic**: assign credit to most frequent states
- **Recency heuristic**: assign credit to most recent states
- *Eligibility traces* combine both heuristics

$$E_0(s) = 0$$

$$E_t(s) = \gamma \lambda E_{t-1}(s) + \mathbf{1}(S_t = s)$$

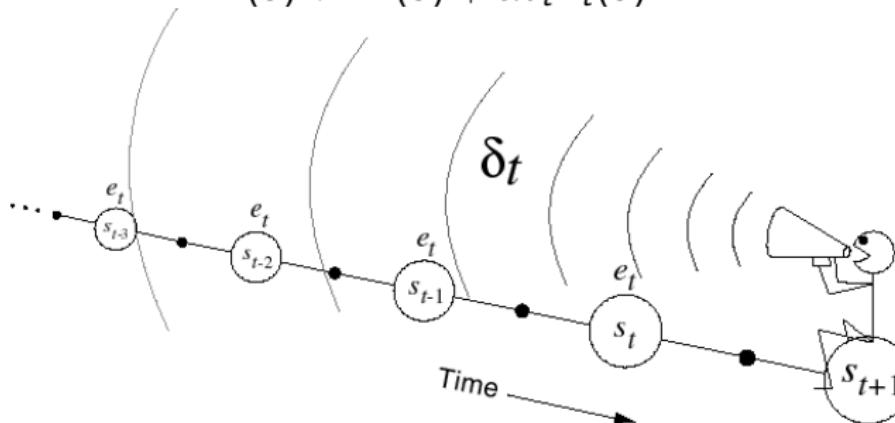


Backward View TD(λ)

- Keep an eligibility trace for every state s
- Update value $V(s)$ for every state s
- In proportion to TD-error δ_t and eligibility trace $E_t(s)$

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

$$V(s) \leftarrow V(s) + \alpha \delta_t E_t(s)$$



TD(λ) and TD(0)

- When $\lambda = 0$, only current state is updated

$$E_t(s) = \mathbf{1}(S_t = s)$$

$$V(s) \leftarrow V(s) + \alpha \delta_t E_t(s)$$

- This is exactly equivalent to TD(0) update

$$V(S_t) \leftarrow V(S_t) + \alpha \delta_t$$

TD(λ) and MC

- When $\lambda = 1$, credit is deferred until end of episode
- Consider episodic environments with offline updates
- Over the course of an episode, total update for TD(1) is the same as total update for MC

Theorem

The sum of offline updates is identical for forward-view and backward-view TD(λ)

$$\sum_{t=1}^T \alpha \delta_t E_t(s) = \sum_{t=1}^T \alpha \left(G_t^\lambda - V(S_t) \right) \mathbf{1}(S_t = s)$$

MC and TD(1)

- Consider an episode where s is visited once at time-step k ,
- TD(1) eligibility trace discounts time since visit,

$$\begin{aligned} E_t(s) &= \gamma E_{t-1}(s) + \mathbf{1}(S_t = s) \\ &= \begin{cases} 0 & \text{if } t < k \\ \gamma^{t-k} & \text{if } t \geq k \end{cases} \end{aligned}$$

- TD(1) updates accumulate error *online*

$$\sum_{t=1}^{T-1} \alpha \delta_t E_t(s) = \alpha \sum_{t=k}^{T-1} \gamma^{t-k} \delta_t = \alpha (G_k - V(S_k))$$

- By end of episode it accumulates total error

$$\delta_k + \gamma \delta_{k+1} + \gamma^2 \delta_{k+2} + \dots + \gamma^{T-1-k} \delta_{T-1}$$

Telescoping in TD(1)

When $\lambda = 1$, sum of TD errors telescopes into MC error,

$$\begin{aligned} & \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \dots + \gamma^{T-1-t}\delta_{T-1} \\ &= R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \\ &\quad + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) - \gamma V(S_{t+1}) \\ &\quad + \gamma^2 R_{t+3} + \gamma^3 V(S_{t+3}) - \gamma^2 V(S_{t+2}) \\ &\quad \vdots \\ &\quad + \gamma^{T-1-t} R_T + \gamma^{T-t} V(S_T) - \gamma^{T-1-t} V(S_{T-1}) \\ &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots + \gamma^{T-1-t} R_T - V(S_t) \\ &= G_t - V(S_t) \end{aligned}$$

TD(λ) and TD(1)

- TD(1) is roughly equivalent to every-visit Monte-Carlo
- Error is accumulated online, step-by-step
- If value function is only updated offline at end of episode
- Then total update is exactly the same as MC

Telescoping in TD(λ)

For general λ , TD errors also telescope to λ -error, $G_t^\lambda - V(S_t)$

$$\begin{aligned}
 G_t^\lambda - V(S_t) &= -V(S_t) + (1-\lambda)\lambda^0(R_{t+1} + \gamma V(S_{t+1})) \\
 &\quad + (1-\lambda)\lambda^1(R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})) \\
 &\quad + (1-\lambda)\lambda^2(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V(S_{t+3})) \\
 &\quad + \dots \\
 &= -V(S_t) + (\gamma\lambda)^0(R_{t+1} + \gamma V(S_{t+1}) - \gamma\lambda V(S_{t+1})) \\
 &\quad + (\gamma\lambda)^1(R_{t+2} + \gamma V(S_{t+2}) - \gamma\lambda V(S_{t+2})) \\
 &\quad + (\gamma\lambda)^2(R_{t+3} + \gamma V(S_{t+3}) - \gamma\lambda V(S_{t+3})) \\
 &\quad + \dots \\
 &= (\gamma\lambda)^0(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \\
 &\quad + (\gamma\lambda)^1(R_{t+2} + \gamma V(S_{t+2}) - V(S_{t+1})) \\
 &\quad + (\gamma\lambda)^2(R_{t+3} + \gamma V(S_{t+3}) - V(S_{t+2})) \\
 &\quad + \dots \\
 &= \delta_t + \gamma\lambda\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots
 \end{aligned}$$

Forwards and Backwards TD(λ)

- Consider an episode where s is visited once at time-step k ,
- TD(λ) eligibility trace discounts time since visit,

$$\begin{aligned} E_t(s) &= \gamma\lambda E_{t-1}(s) + \mathbf{1}(S_t = s) \\ &= \begin{cases} 0 & \text{if } t < k \\ (\gamma\lambda)^{t-k} & \text{if } t \geq k \end{cases} \end{aligned}$$

- Backward TD(λ) updates accumulate error *online*

$$\sum_{t=1}^T \alpha \delta_t E_t(s) = \alpha \sum_{t=k}^T (\gamma\lambda)^{t-k} \delta_t = \alpha \left(G_k^\lambda - V(S_k) \right)$$

- By end of episode it accumulates total error for λ -return
- For multiple visits to s , $E_t(s)$ accumulates many errors

Offline Equivalence of Forward and Backward TD

Offline updates

- Updates are accumulated within episode
- but applied in batch at the end of episode

Online Equivalence of Forward and Backward TD

Online updates

- TD(λ) updates are applied online at each step within episode
- Forward and backward-view TD(λ) are slightly different
- **NEW:** Exact online TD(λ) achieves perfect equivalence
- By using a slightly different form of eligibility trace
- Sutton and von Seijen, ICML 2014

Summary of Forward and Backward TD(λ)

| | | | |
|-----------------|---------------|------------------------------|--------------------|
| Offline updates | $\lambda = 0$ | $\lambda \in (0, 1)$ | $\lambda = 1$ |
| Backward view | TD(0) | TD(λ) | TD(1) |
| Forward view | TD(0) | Forward TD(λ) | MC |
| Online updates | $\lambda = 0$ | $\lambda \in (0, 1)$ | $\lambda = 1$ |
| Backward view | TD(0) | TD(λ) ※ | TD(1) ※ |
| Forward view | TD(0) | Forward TD(λ) | MC |
| Exact Online | TD(0) | Exact Online TD(λ) | Exact Online TD(1) |

= here indicates equivalence in total update at end of episode.

Lecture 5: Model-Free Control

David Silver

Outline

- 1 Introduction
- 2 On-Policy Monte-Carlo Control
- 3 On-Policy Temporal-Difference Learning
- 4 Off-Policy Learning
- 5 Summary

Model-Free Reinforcement Learning

- Last lecture:
 - Model-free prediction
 - Estimate the value function of an *unknown* MDP
- This lecture:
 - Model-free control
 - Optimise the value function of an *unknown* MDP

Uses of Model-Free Control

Some example problems that can be modelled as MDPs

- Elevator
- Parallel Parking
- Ship Steering
- Bioreactor
- Helicopter
- Aeroplane Logistics
- Robocup Soccer
- Quake
- Portfolio management
- Protein Folding
- Robot walking
- Game of Go

For most of these problems, either:

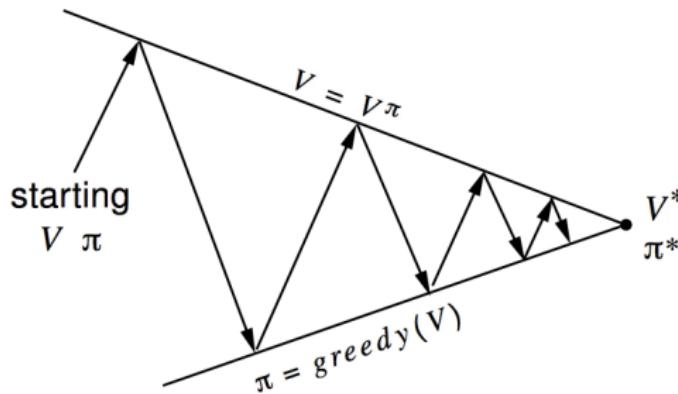
- MDP model is unknown, but experience can be sampled
- MDP model is known, but is too big to use, except by samples

Model-free control can solve these problems

On and Off-Policy Learning

- **On-policy learning**
 - “Learn on the job”
 - Learn about policy π from experience sampled from π
- **Off-policy learning**
 - “Look over someone’s shoulder”
 - Learn about policy π from experience sampled from μ

Generalised Policy Iteration (Refresher)

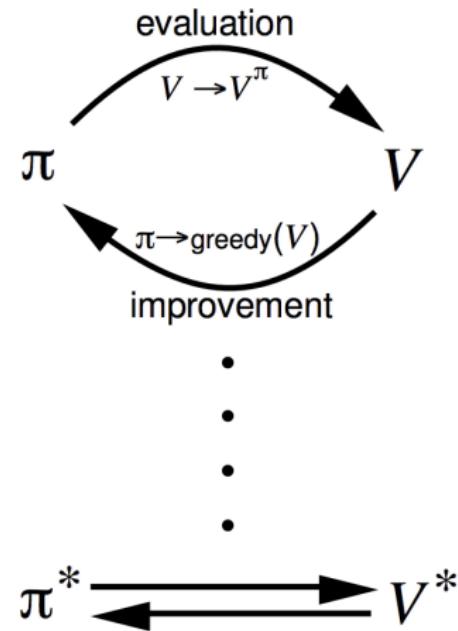


Policy evaluation Estimate v_π

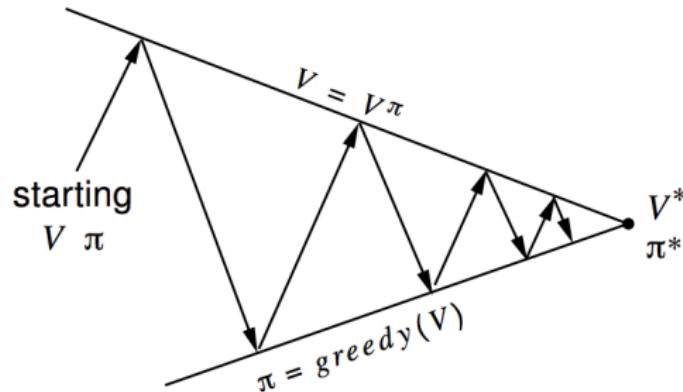
e.g. Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$

e.g. Greedy policy improvement



Generalised Policy Iteration With Monte-Carlo Evaluation



Policy evaluation Monte-Carlo policy evaluation, $V = v_\pi$?

Policy improvement Greedy policy improvement?

Model-Free Policy Iteration Using Action-Value Function

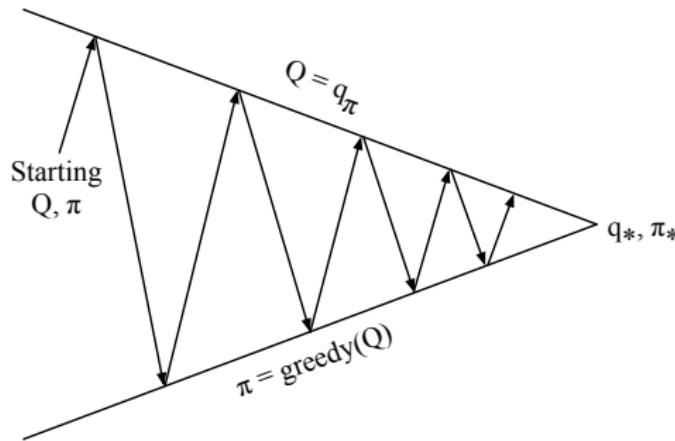
- Greedy policy improvement over $V(s)$ requires model of MDP

$$\pi'(s) = \operatorname{argmax}_{a \in \mathcal{A}} \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$$

- Greedy policy improvement over $Q(s, a)$ is model-free

$$\pi'(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q(s, a)$$

Generalised Policy Iteration with Action-Value Function



Policy evaluation Monte-Carlo policy evaluation, $Q = q_\pi$

Policy improvement Greedy policy improvement?

Example of Greedy Action Selection



"Behind one door is tenure - behind the other is flipping burgers at McDonald's."

- There are two doors in front of you.
- You open the left door and get reward 0
 $V(\text{left}) = 0$
- You open the right door and get reward +1
 $V(\text{right}) = +1$
- You open the right door and get reward +3
 $V(\text{right}) = +2$
- You open the right door and get reward +2
 $V(\text{right}) = +2$
- ⋮
- Are you sure you've chosen the best door?

ϵ -Greedy Exploration

- Simplest idea for ensuring continual exploration
- All m actions are tried with non-zero probability
- With probability $1 - \epsilon$ choose the greedy action
- With probability ϵ choose an action at random

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q(s, a) \\ \epsilon/m & \text{otherwise} \end{cases}$$

ϵ -Greedy Policy Improvement

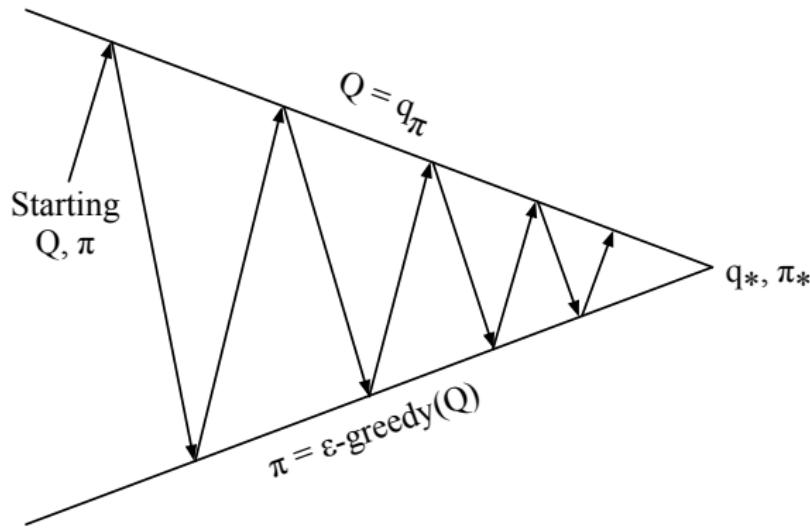
Theorem

For any ϵ -greedy policy π , the ϵ -greedy policy π' with respect to q_π is an improvement, $v_{\pi'}(s) \geq v_\pi(s)$

$$\begin{aligned}
 q_\pi(s, \pi'(s)) &= \sum_{a \in \mathcal{A}} \pi'(a|s) q_\pi(s, a) \\
 &= \epsilon/m \sum_{a \in \mathcal{A}} q_\pi(s, a) + (1 - \epsilon) \max_{a \in \mathcal{A}} q_\pi(s, a) \\
 &\geq \epsilon/m \sum_{a \in \mathcal{A}} q_\pi(s, a) + (1 - \epsilon) \sum_{a \in \mathcal{A}} \frac{\pi(a|s) - \epsilon/m}{1 - \epsilon} q_\pi(s, a) \\
 &= \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a) = v_\pi(s)
 \end{aligned}$$

Therefore from policy improvement theorem, $v_{\pi'}(s) \geq v_\pi(s)$

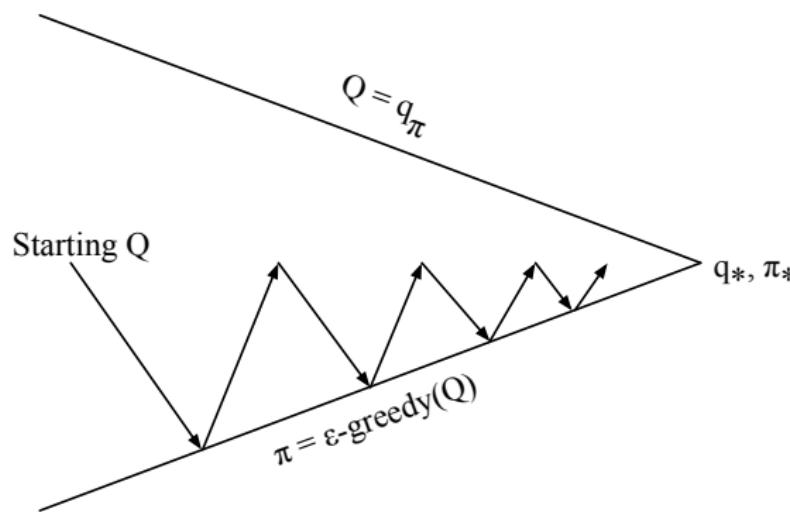
Monte-Carlo Policy Iteration



Policy evaluation Monte-Carlo policy evaluation, $Q = q_\pi$

Policy improvement ϵ -greedy policy improvement

Monte-Carlo Control



Every episode:

Policy evaluation Monte-Carlo policy evaluation, $Q \approx q_\pi$

Policy improvement ϵ -greedy policy improvement

GLIE

Definition

Greedy in the Limit with Infinite Exploration (GLIE)

- All state-action pairs are explored infinitely many times,

$$\lim_{k \rightarrow \infty} N_k(s, a) = \infty$$

- The policy converges on a greedy policy,

$$\lim_{k \rightarrow \infty} \pi_k(a|s) = \mathbf{1}(a = \operatorname{argmax}_{a' \in \mathcal{A}} Q_k(s, a'))$$

- For example, ϵ -greedy is GLIE if ϵ reduces to zero at $\epsilon_k = \frac{1}{k}$

GLIE Monte-Carlo Control

- Sample k th episode using π : $\{S_1, A_1, R_2, \dots, S_T\} \sim \pi$
- For each state S_t and action A_t in the episode,

$$N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)} (G_t - Q(S_t, A_t))$$

- Improve policy based on new action-value function

$$\epsilon \leftarrow 1/k$$

$$\pi \leftarrow \epsilon\text{-greedy}(Q)$$

Theorem

GLIE Monte-Carlo control converges to the optimal action-value function, $Q(s, a) \rightarrow q_(s, a)$*

Lecture 5: Model-Free Control

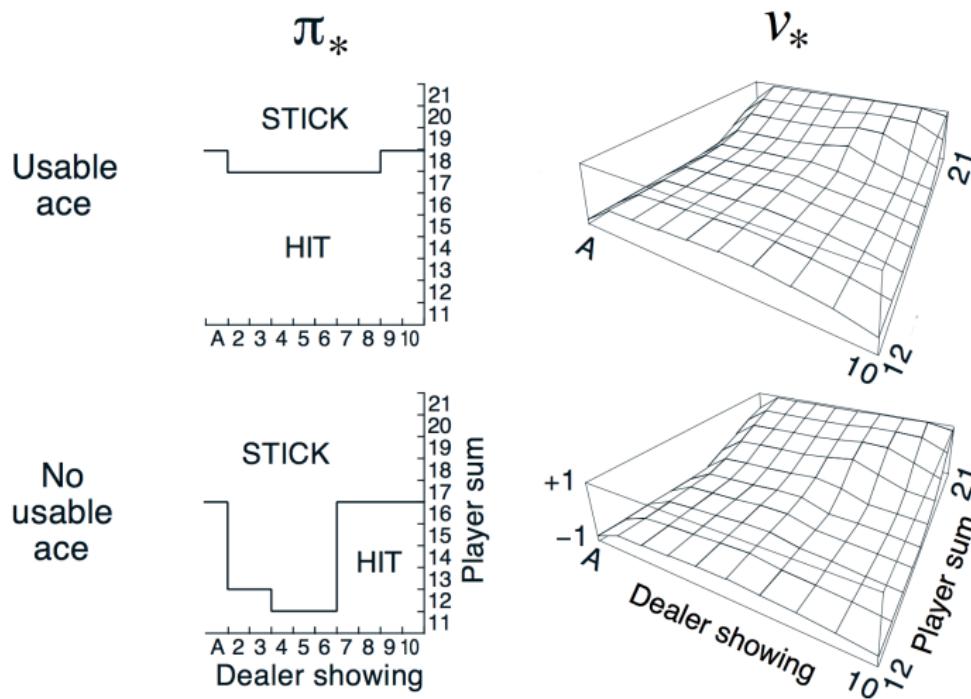
└ On-Policy Monte-Carlo Control

└ Blackjack Example

Back to the Blackjack Example



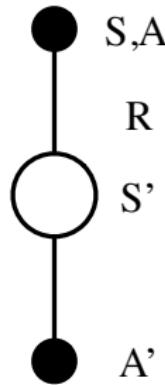
Monte-Carlo Control in Blackjack



MC vs. TD Control

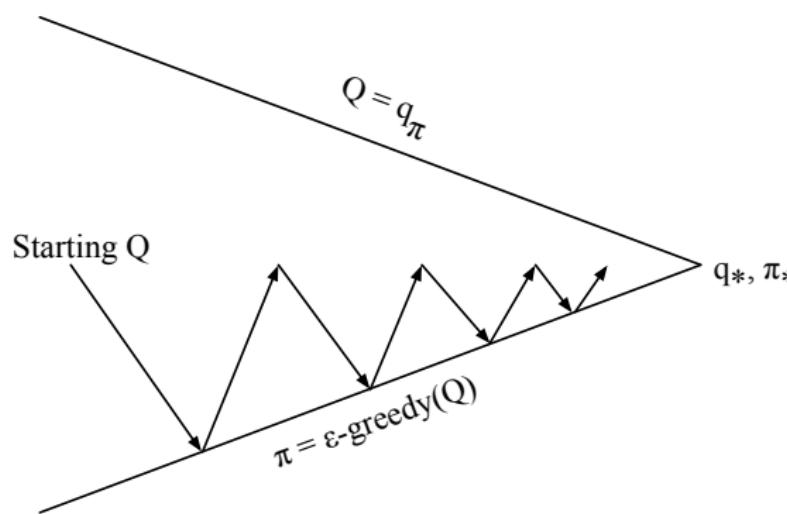
- Temporal-difference (TD) learning has several advantages over Monte-Carlo (MC)
 - Lower variance
 - Online
 - Incomplete sequences
- Natural idea: use TD instead of MC in our control loop
 - Apply TD to $Q(S, A)$
 - Use ϵ -greedy policy improvement
 - Update every time-step

Updating Action-Value Functions with Sarsa



$$Q(S, A) \leftarrow Q(S, A) + \alpha (R + \gamma Q(S', A') - Q(S, A))$$

On-Policy Control With Sarsa



Every **time-step**:

Policy evaluation **Sarsa**, $Q \approx q_\pi$

Policy improvement ϵ -greedy policy improvement

Sarsa Algorithm for On-Policy Control

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$
Repeat (for each episode):

 Initialize S

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Repeat (for each step of episode):

 Take action A , observe R, S'

 Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

$S \leftarrow S'; A \leftarrow A'$;

 until S is terminal

Convergence of Sarsa

Theorem

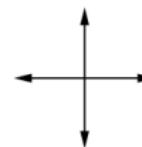
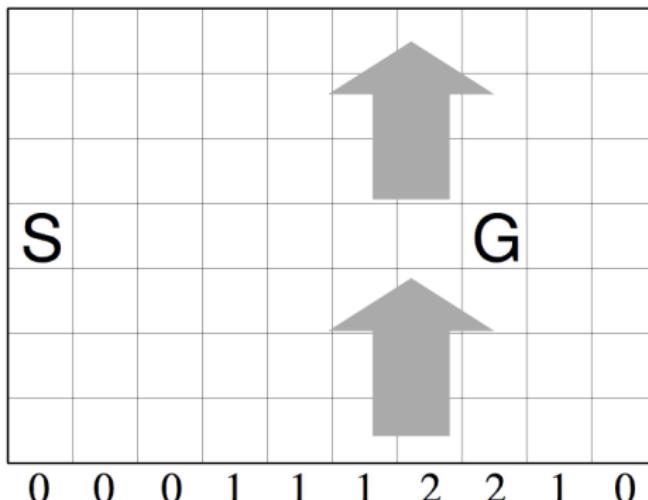
*Sarsa converges to the optimal action-value function,
 $Q(s, a) \rightarrow q_*(s, a)$, under the following conditions:*

- GLIE sequence of policies $\pi_t(a|s)$
- Robbins-Monro sequence of step-sizes α_t

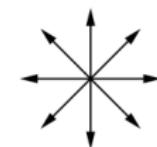
$$\sum_{t=1}^{\infty} \alpha_t = \infty$$

$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

Windy Gridworld Example



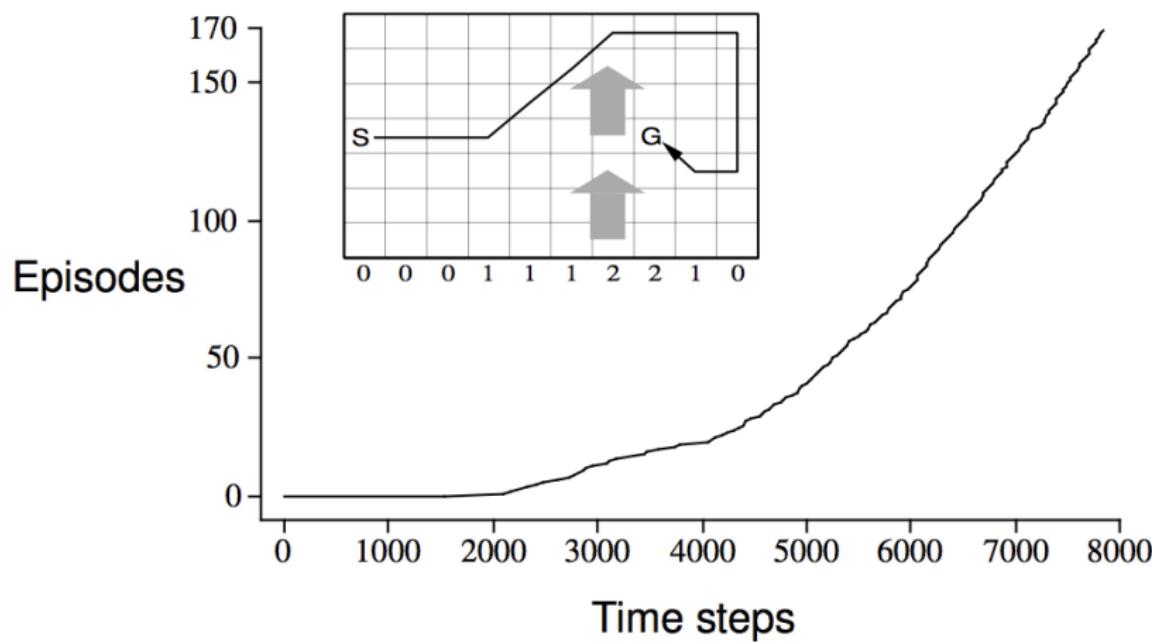
standard moves



king's moves

- Reward = -1 per time-step until reaching goal
- Undiscounted

Sarsa on the Windy Gridworld



n -Step Sarsa

- Consider the following n -step returns for $n = 1, 2, \infty$:

$$n = 1 \quad (\text{Sarsa}) \quad q_t^{(1)} = R_{t+1} + \gamma Q(S_{t+1})$$

$$n = 2 \quad q_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2})$$

$$\vdots$$

$$\vdots$$

$$n = \infty \quad (\text{MC}) \quad q_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$$

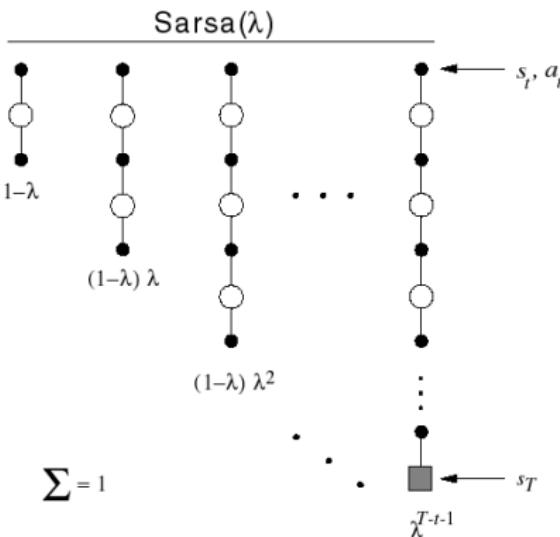
- Define the n -step Q-return

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n})$$

- n -step Sarsa updates $Q(s, a)$ towards the n -step Q-return

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(q_t^{(n)} - Q(S_t, A_t) \right)$$

Forward View Sarsa(λ)



- The q^λ return combines all n -step Q>Returns $q_t^{(n)}$
- Using weight $(1 - \lambda)\lambda^{n-1}$

$$q_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}$$

- Forward-view Sarsa(λ)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(q_t^\lambda - Q(S_t, A_t) \right)$$

Backward View Sarsa(λ)

- Just like TD(λ), we use **eligibility traces** in an online algorithm
- But Sarsa(λ) has one eligibility trace for each state-action pair

$$E_0(s, a) = 0$$

$$E_t(s, a) = \gamma \lambda E_{t-1}(s, a) + \mathbf{1}(S_t = s, A_t = a)$$

- $Q(s, a)$ is updated for every state s and action a
- In proportion to TD-error δ_t and eligibility trace $E_t(s, a)$

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta_t E_t(s, a)$$

Sarsa(λ) Algorithm

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

Repeat (for each episode):

$$E(s, a) = 0, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s)$$

Initialize S, A

Repeat (for each step of episode):

Take action A , observe R, S'

Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$$\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$$

$$E(S, A) \leftarrow E(S, A) + 1$$

For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta E(s, a)$$

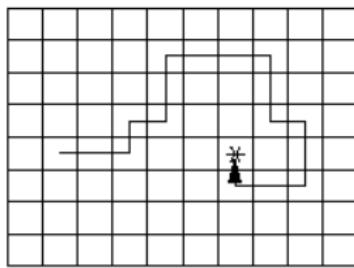
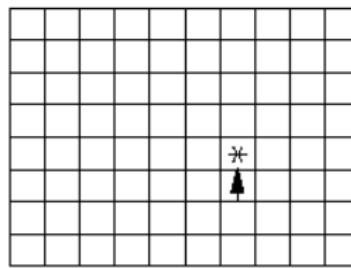
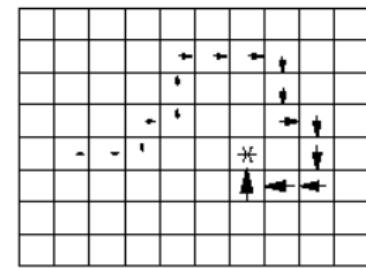
$$E(s, a) \leftarrow \gamma \lambda E(s, a)$$

$$S \leftarrow S'; A \leftarrow A'$$

until S is terminal

Sarsa(λ) Gridworld Example

Path taken

Action values increased
by one-step SarsaAction values increased
by Sarsa(λ) with $\lambda=0.9$ 

Off-Policy Learning

- Evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s, a)$
- While following behaviour policy $\mu(a|s)$

$$\{S_1, A_1, R_2, \dots, S_T\} \sim \mu$$

- Why is this important?
- Learn from observing humans or other agents
- Re-use experience generated from old policies $\pi_1, \pi_2, \dots, \pi_{t-1}$
- Learn about *optimal* policy while following *exploratory* policy
- Learn about *multiple* policies while following *one* policy

Importance Sampling

- Estimate the expectation of a different distribution

$$\begin{aligned}\mathbb{E}_{X \sim P}[f(X)] &= \sum P(X)f(X) \\ &= \sum Q(X)\frac{P(X)}{Q(X)}f(X) \\ &= \mathbb{E}_{X \sim Q}\left[\frac{P(X)}{Q(X)}f(X)\right]\end{aligned}$$

Importance Sampling for Off-Policy Monte-Carlo

- Use returns generated from μ to evaluate π
- Weight return G_t according to similarity between policies
- Multiply importance sampling corrections along whole episode

$$G_t^{\pi/\mu} = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \cdots \frac{\pi(A_T|S_T)}{\mu(A_T|S_T)} G_t$$

- Update value towards *corrected* return

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t^{\pi/\mu} - V(S_t) \right)$$

- Cannot use if μ is zero when π is non-zero
- Importance sampling can dramatically increase variance

Importance Sampling for Off-Policy TD

- Use TD targets generated from μ to evaluate π
- Weight TD target $R + \gamma V(S')$ by importance sampling
- Only need a single importance sampling correction

$$V(S_t) \leftarrow V(S_t) + \alpha \left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma V(S_{t+1})) - V(S_t) \right)$$

- Much lower variance than Monte-Carlo importance sampling
- Policies only need to be similar over a single step

Q-Learning

- We now consider off-policy learning of action-values $Q(s, a)$
- **No** importance sampling is required
- Next action is chosen using behaviour policy $A_{t+1} \sim \mu(\cdot | S_t)$
- But we consider alternative successor action $A' \sim \pi(\cdot | S_t)$
- And update $Q(S_t, A_t)$ towards value of alternative action

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t))$$

Off-Policy Control with Q-Learning

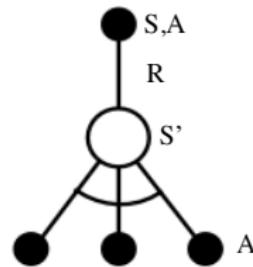
- We now allow both behaviour and target policies to **improve**
- The target policy π is **greedy** w.r.t. $Q(s, a)$

$$\pi(S_{t+1}) = \operatorname{argmax}_{a'} Q(S_{t+1}, a')$$

- The behaviour policy μ is e.g. **ϵ -greedy** w.r.t. $Q(s, a)$
- The Q-learning target then simplifies:

$$\begin{aligned} & R_{t+1} + \gamma Q(S_{t+1}, A') \\ &= R_{t+1} + \gamma Q(S_{t+1}, \operatorname{argmax}_{a'} Q(S_{t+1}, a')) \\ &= R_{t+1} + \max_{a'} \gamma Q(S_{t+1}, a') \end{aligned}$$

Q-Learning Control Algorithm



$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \max_{a'} Q(S', a') - Q(S, A) \right)$$

Theorem

Q-learning control converges to the optimal action-value function,
 $Q(s, a) \rightarrow q_*(s, a)$

Q-Learning Algorithm for Off-Policy Control

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

 Initialize S

 Repeat (for each step of episode):

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Take action A , observe R, S'

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

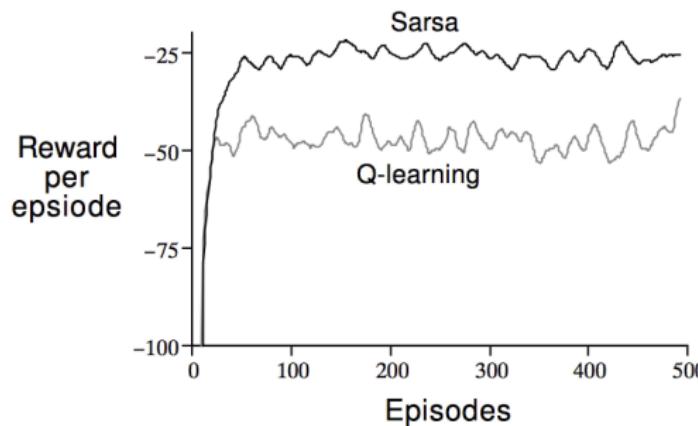
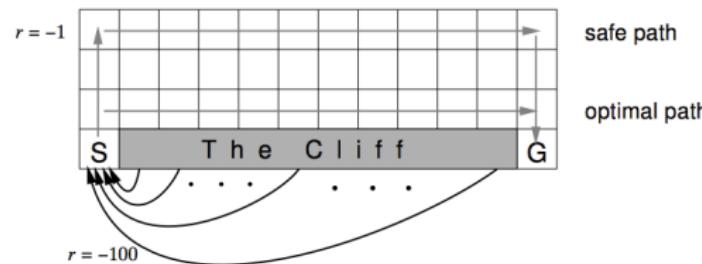
$S \leftarrow S'$;

 until S is terminal

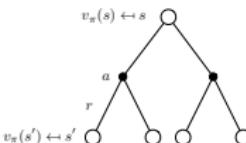
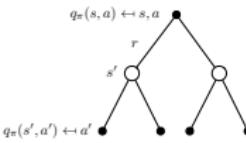
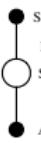
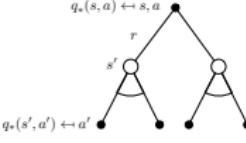
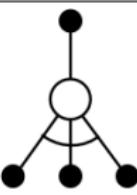
Q-Learning Demo

Q-Learning Demo

Cliff Walking Example



Relationship Between DP and TD

| | <i>Full Backup (DP)</i> | <i>Sample Backup (TD)</i> |
|--|---|--|
| Bellman Expectation Equation for $v_\pi(s)$ | $v_\pi(s) \leftarrow s$  <p>Iterative Policy Evaluation</p> |  <p>TD Learning</p> |
| Bellman Expectation Equation for $q_\pi(s, a)$ | $q_\pi(s, a) \leftarrow s, a$  <p>Q-Policy Iteration</p> |  <p>Sarsa</p> |
| Bellman Optimality Equation for $q_*(s, a)$ | $q_*(s, a) \leftarrow s, a$  <p>Q-Value Iteration</p> |  <p>Q-Learning</p> |

Relationship Between DP and TD (2)

| <i>Full Backup (DP)</i> | <i>Sample Backup (TD)</i> |
|--|--|
| Iterative Policy Evaluation $V(s) \leftarrow \mathbb{E}[R + \gamma V(S') \mid s]$ | TD Learning $V(S) \xleftarrow{\alpha} R + \gamma V(S')$ |
| Q-Policy Iteration $Q(s, a) \leftarrow \mathbb{E}[R + \gamma Q(S', A') \mid s, a]$ | Sarsa $Q(S, A) \xleftarrow{\alpha} R + \gamma Q(S', A')$ |
| Q-Value Iteration $Q(s, a) \leftarrow \mathbb{E} \left[R + \gamma \max_{a' \in \mathcal{A}} Q(S', a') \mid s, a \right]$ | Q-Learning $Q(S, A) \xleftarrow{\alpha} R + \gamma \max_{a' \in \mathcal{A}} Q(S', a')$ |

where $x \xleftarrow{\alpha} y \equiv x \leftarrow x + \alpha(y - x)$

Questions?

Lecture 6: Value Function Approximation

David Silver

Outline

1 Introduction

2 Incremental Methods

3 Batch Methods

Outline

1 Introduction

2 Incremental Methods

3 Batch Methods

Large-Scale Reinforcement Learning

Reinforcement learning can be used to solve *large* problems, e.g.

- Backgammon: 10^{20} states
- Computer Go: 10^{170} states
- Helicopter: continuous state space

Large-Scale Reinforcement Learning

Reinforcement learning can be used to solve *large* problems, e.g.

- Backgammon: 10^{20} states
- Computer Go: 10^{170} states
- Helicopter: continuous state space

How can we scale up the model-free methods for *prediction* and *control* from the last two lectures?

Value Function Approximation

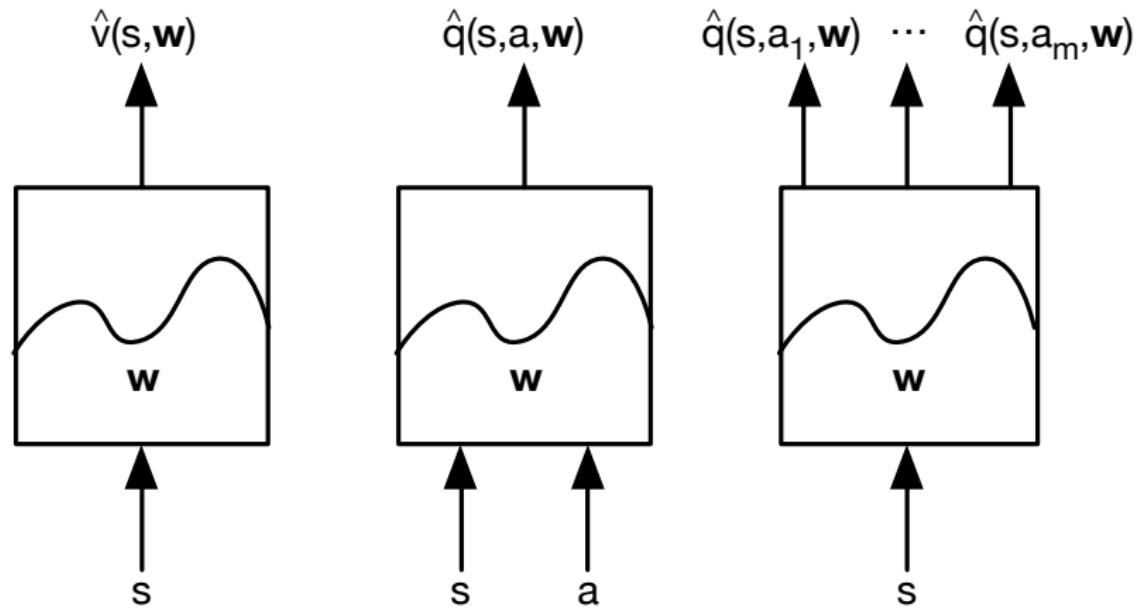
- So far we have represented value function by a *lookup table*
 - Every state s has an entry $V(s)$
 - Or every state-action pair s, a has an entry $Q(s, a)$
- Problem with large MDPs:
 - There are too many states and/or actions to store in memory
 - It is too slow to learn the value of each state individually
- Solution for large MDPs:
 - Estimate value function with *function approximation*

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

$$\text{or } \hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

- Generalise from seen states to unseen states
- Update parameter \mathbf{w} using MC or TD learning

Types of Value Function Approximation



Which Function Approximator?

There are many function approximators, e.g.

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

Which Function Approximator?

We consider **differentiable** function approximators, e.g.

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

Furthermore, we require a training method that is suitable for
non-stationary, non-iid data

Outline

1 Introduction

2 Incremental Methods

3 Batch Methods

Gradient Descent

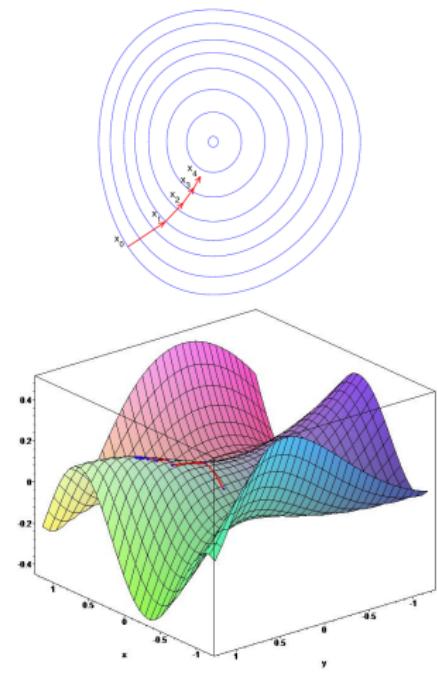
- Let $J(\mathbf{w})$ be a differentiable function of parameter vector \mathbf{w}
 - Define the *gradient* of $J(\mathbf{w})$ to be

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$

- To find a local minimum of $J(\mathbf{w})$
 - Adjust \mathbf{w} in direction of -ve gradient

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

where α is a step-size parameter



Value Function Approx. By Stochastic Gradient Descent

- Goal: find parameter vector \mathbf{w} minimising mean-squared error between approximate value fn $\hat{v}(s, \mathbf{w})$ and true value fn $v_\pi(s)$

$$J(\mathbf{w}) = \mathbb{E}_\pi [(v_\pi(S) - \hat{v}(S, \mathbf{w}))^2]$$

- Gradient descent finds a local minimum

$$\begin{aligned}\Delta \mathbf{w} &= -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi [(v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})]\end{aligned}$$

- Stochastic gradient descent *samples* the gradient

$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

- Expected update is equal to full gradient update

Feature Vectors

- Represent state by a *feature vector*

$$\mathbf{x}(S) = \begin{pmatrix} \mathbf{x}_1(S) \\ \vdots \\ \mathbf{x}_n(S) \end{pmatrix}$$

- For example:
 - Distance of robot from landmarks
 - Trends in the stock market
 - Piece and pawn configurations in chess

Linear Value Function Approximation

- Represent value function by a linear combination of features

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^\top \mathbf{w} = \sum_{j=1}^n \mathbf{x}_j(S) w_j$$

- Objective function is quadratic in parameters \mathbf{w}

$$J(\mathbf{w}) = \mathbb{E}_{\pi} \left[(v_{\pi}(S) - \mathbf{x}(S)^\top \mathbf{w})^2 \right]$$

- Stochastic gradient descent converges on *global* optimum
- Update rule is particularly simple

$$\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) = \mathbf{x}(S)$$

$$\Delta \mathbf{w} = \alpha (v_{\pi}(S) - \hat{v}(S, \mathbf{w})) \mathbf{x}(S)$$

Update = step-size × prediction error × feature value

Table Lookup Features

- Table lookup is a special case of linear value function approximation
- Using *table lookup features*

$$\mathbf{x}^{table}(S) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix}$$

- Parameter vector \mathbf{w} gives value of each individual state

$$\hat{v}(S, \mathbf{w}) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_n \end{pmatrix}$$

Incremental Prediction Algorithms

- Have assumed true value function $v_\pi(s)$ given by supervisor
- But in RL there is no supervisor, only rewards
- In practice, we substitute a *target* for $v_\pi(s)$
 - For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$$

- For TD(0), the target is the TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$$

- For TD(λ), the target is the λ -return G_t^λ

$$\Delta \mathbf{w} = \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$$

Monte-Carlo with Value Function Approximation

- Return G_t is an unbiased, noisy sample of true value $v_\pi(S_t)$
- Can therefore apply supervised learning to “training data”:

$$\langle S_1, G_1 \rangle, \langle S_2, G_2 \rangle, \dots, \langle S_T, G_T \rangle$$

- For example, using *linear Monte-Carlo policy evaluation*

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t - \hat{v}(S_t, \mathbf{w})) \mathbf{x}(S_t)\end{aligned}$$

- Monte-Carlo evaluation converges to a local optimum
- Even when using non-linear value function approximation

TD Learning with Value Function Approximation

- The TD-target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ is a *biased* sample of true value $v_\pi(S_t)$
- Can still apply supervised learning to “training data”:

$$\langle S_1, R_2 + \gamma \hat{v}(S_2, \mathbf{w}) \rangle, \langle S_2, R_3 + \gamma \hat{v}(S_3, \mathbf{w}) \rangle, \dots, \langle S_{T-1}, R_T \rangle$$

- For example, using *linear TD(0)*

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \\ &= \alpha \delta \mathbf{x}(S)\end{aligned}$$

- Linear TD(0) converges (close) to global optimum

TD(λ) with Value Function Approximation

- The λ -return G_t^λ is also a biased sample of true value $v_\pi(s)$
- Can again apply supervised learning to “training data”:

$$\langle S_1, G_1^\lambda \rangle, \langle S_2, G_2^\lambda \rangle, \dots, \langle S_{T-1}, G_{T-1}^\lambda \rangle$$

- Forward view linear TD(λ)

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w})) \mathbf{x}(S_t)\end{aligned}$$

- Backward view linear TD(λ)

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$$

$$E_t = \gamma \lambda E_{t-1} + \mathbf{x}(S_t)$$

$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

TD(λ) with Value Function Approximation

- The λ -return G_t^λ is also a biased sample of true value $v_\pi(s)$
- Can again apply supervised learning to “training data”:

$$\langle S_1, G_1^\lambda \rangle, \langle S_2, G_2^\lambda \rangle, \dots, \langle S_{T-1}, G_{T-1}^\lambda \rangle$$

- Forward view linear TD(λ)

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w})) \mathbf{x}(S_t)\end{aligned}$$

- Backward view linear TD(λ)

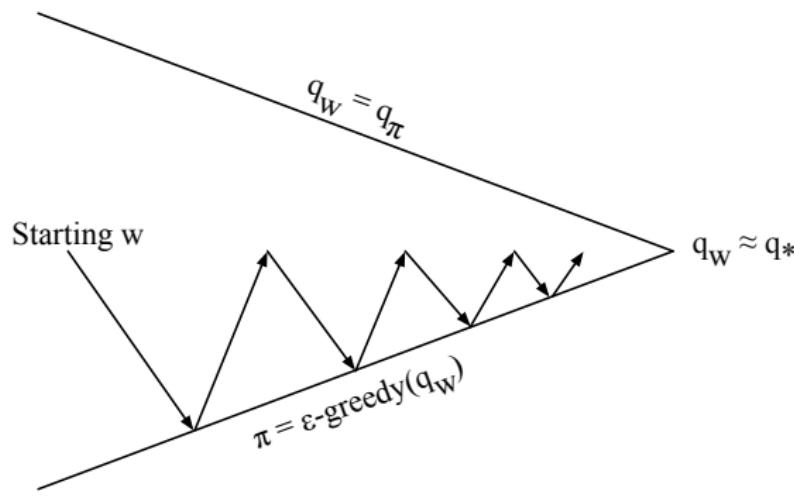
$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$$

$$E_t = \gamma \lambda E_{t-1} + \mathbf{x}(S_t)$$

$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

Forward view and backward view linear TD(λ) are equivalent

Control with Value Function Approximation



Policy evaluation **Approximate** policy evaluation, $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$

Policy improvement ϵ -greedy policy improvement

Action-Value Function Approximation

- Approximate the action-value function

$$\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$$

- Minimise mean-squared error between approximate action-value fn $\hat{q}(S, A, \mathbf{w})$ and true action-value fn $q_\pi(S, A)$

$$J(\mathbf{w}) = \mathbb{E}_\pi [(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2]$$

- Use stochastic gradient descent to find a local minimum

$$-\frac{1}{2} \nabla_{\mathbf{w}} J(\mathbf{w}) = (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

$$\Delta \mathbf{w} = \alpha (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

Linear Action-Value Function Approximation

- Represent state *and* action by a *feature vector*

$$\mathbf{x}(S, A) = \begin{pmatrix} \mathbf{x}_1(S, A) \\ \vdots \\ \mathbf{x}_n(S, A) \end{pmatrix}$$

- Represent action-value fn by linear combination of features

$$\hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)^\top \mathbf{w} = \sum_{j=1}^n \mathbf{x}_j(S, A) w_j$$

- Stochastic gradient descent update

$$\nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)$$

$$\Delta \mathbf{w} = \alpha(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \mathbf{x}(S, A)$$

Incremental Control Algorithms

- Like prediction, we must substitute a *target* for $q_{\pi}(S, A)$
 - For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For TD(0), the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For forward-view TD(λ), target is the action-value λ -return

$$\Delta \mathbf{w} = \alpha(q_t^{\lambda} - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For backward-view TD(λ), equivalent update is

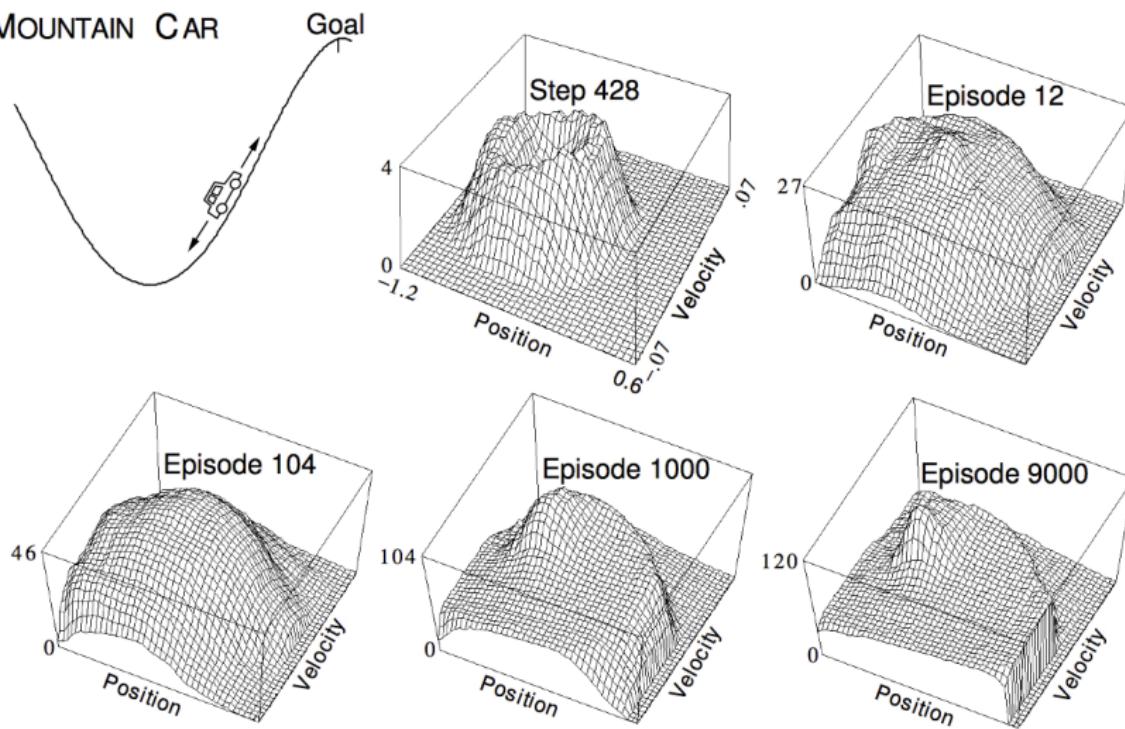
$$\delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})$$

$$E_t = \gamma \lambda E_{t-1} + \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

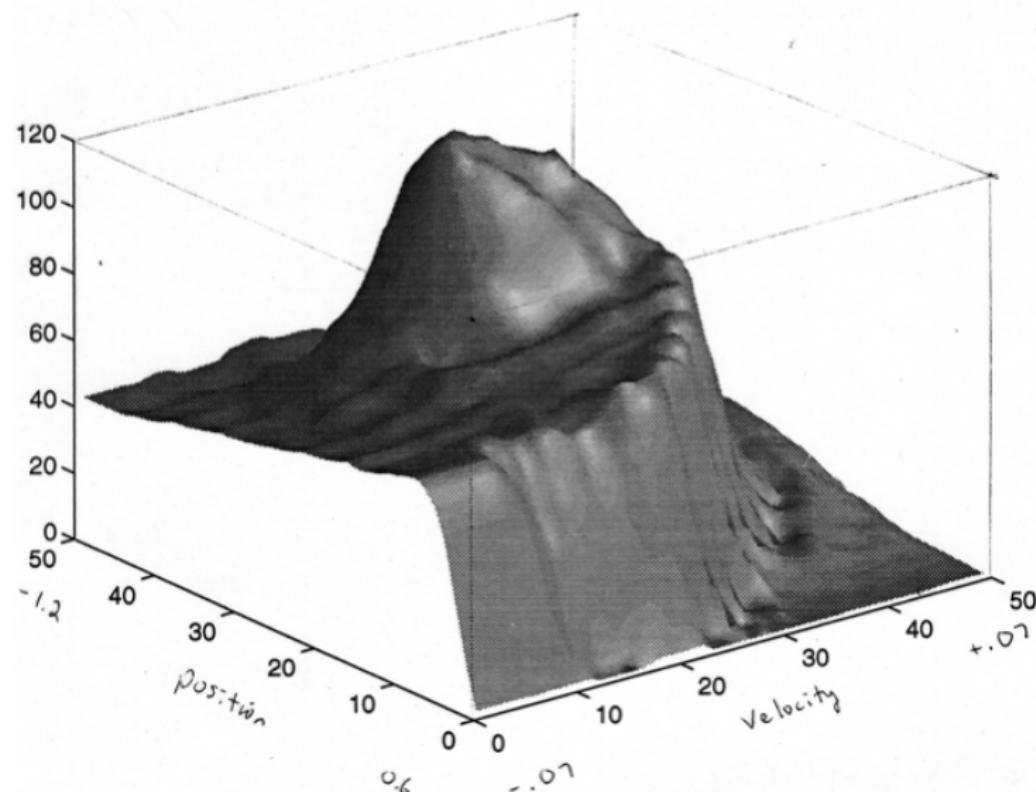
$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

Linear Sarsa with Coarse Coding in Mountain Car

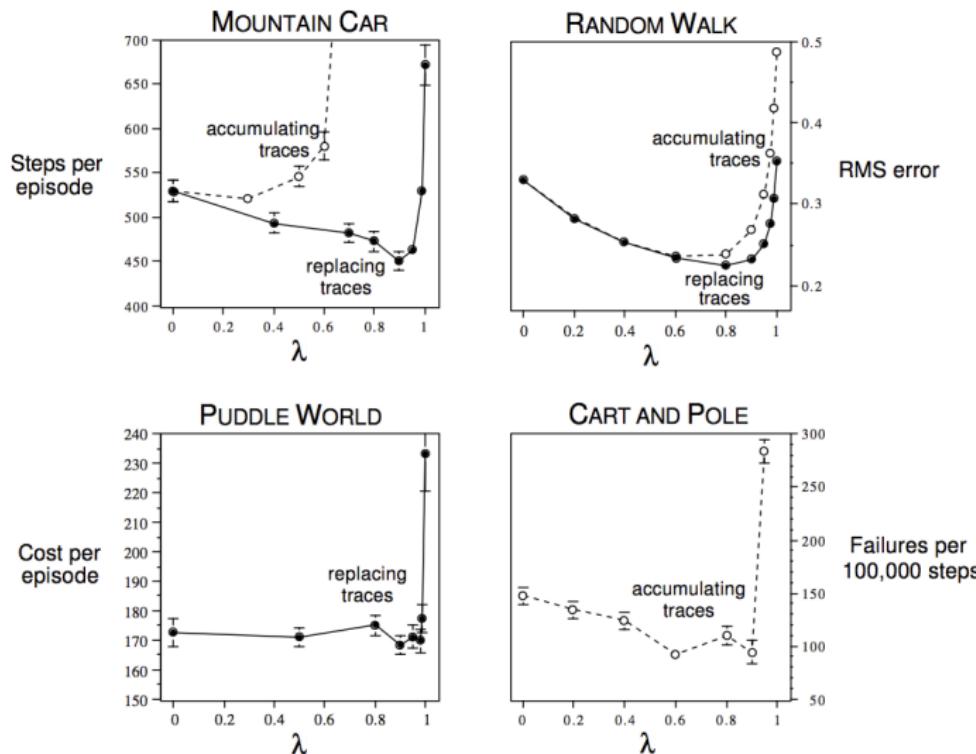
MOUNTAIN CAR



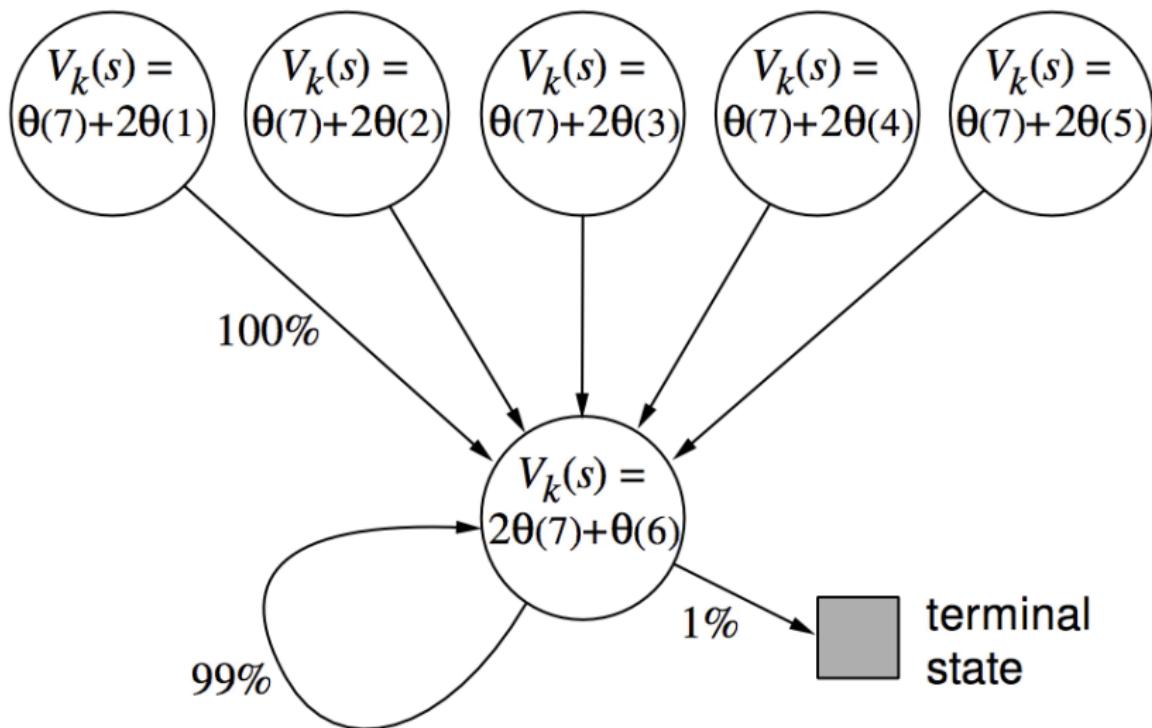
Linear Sarsa with Radial Basis Functions in Mountain Car



Study of λ : Should We Bootstrap?

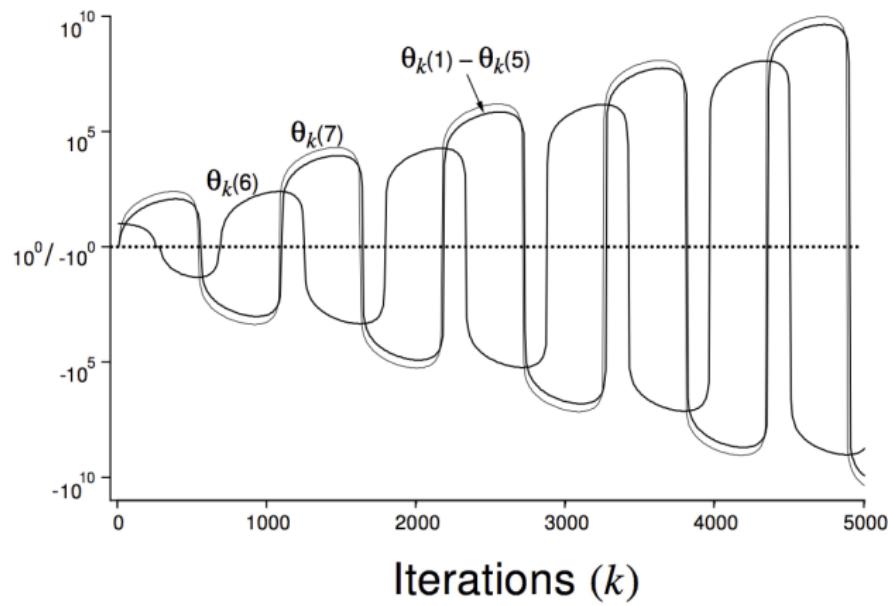


Baird's Counterexample



Parameter Divergence in Baird's Counterexample

Parameter values, $\theta_k(i)$
(log scale,
broken at ± 1)



Convergence of Prediction Algorithms

| On/Off-Policy | Algorithm | Table Lookup | Linear | Non-Linear |
|---------------|-----------------|--------------|--------|------------|
| On-Policy | MC | ✓ | ✓ | ✓ |
| | TD(0) | ✓ | ✓ | ✗ |
| | TD(λ) | ✓ | ✓ | ✗ |
| Off-Policy | MC | ✓ | ✓ | ✓ |
| | TD(0) | ✓ | ✗ | ✗ |
| | TD(λ) | ✓ | ✗ | ✗ |

Gradient Temporal-Difference Learning

- TD does not follow the gradient of *any* objective function
- This is why TD can diverge when off-policy or using non-linear function approximation
- Gradient TD follows true gradient of projected Bellman error

| On/Off-Policy | Algorithm | Table Lookup | Linear | Non-Linear |
|---------------|-------------|--------------|--------|------------|
| On-Policy | MC | ✓ | ✓ | ✓ |
| | TD | ✓ | ✓ | ✗ |
| | Gradient TD | ✓ | ✓ | ✓ |
| Off-Policy | MC | ✓ | ✓ | ✓ |
| | TD | ✓ | ✗ | ✗ |
| | Gradient TD | ✓ | ✓ | ✓ |

Convergence of Control Algorithms

| Algorithm | Table Lookup | Linear | Non-Linear |
|---------------------|--------------|--------|------------|
| Monte-Carlo Control | ✓ | (✓) | ✗ |
| Sarsa | ✓ | (✓) | ✗ |
| Q-learning | ✓ | ✗ | ✗ |
| Gradient Q-learning | ✓ | ✓ | ✗ |

(✓) = chatters around near-optimal value function

Outline

1 Introduction

2 Incremental Methods

3 Batch Methods

Batch Reinforcement Learning

- Gradient descent is simple and appealing
- But it is *not* sample efficient
- Batch methods seek to find the best fitting value function
- Given the agent's experience ("training data")

Least Squares Prediction

- Given value function approximation $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$
- And *experience* \mathcal{D} consisting of $\langle \text{state}, \text{value} \rangle$ pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \dots, \langle s_T, v_T^\pi \rangle\}$$

- Which parameters \mathbf{w} give the *best fitting* value fn $\hat{v}(s, \mathbf{w})$?
- **Least squares** algorithms find parameter vector \mathbf{w} minimising sum-squared error between $\hat{v}(s_t, \mathbf{w})$ and target values v_t^π ,

$$\begin{aligned} LS(\mathbf{w}) &= \sum_{t=1}^T (v_t^\pi - \hat{v}(s_t, \mathbf{w}))^2 \\ &= \mathbb{E}_{\mathcal{D}} [(v^\pi - \hat{v}(s, \mathbf{w}))^2] \end{aligned}$$

Stochastic Gradient Descent with Experience Replay

Given experience consisting of $\langle \text{state}, \text{value} \rangle$ pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \dots, \langle s_T, v_T^\pi \rangle\}$$

Repeat:

- 1 Sample state, value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

- 2 Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(v^\pi - \hat{v}(s, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$$

Stochastic Gradient Descent with Experience Replay

Given experience consisting of $\langle \text{state}, \text{value} \rangle$ pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \dots, \langle s_T, v_T^\pi \rangle\}$$

Repeat:

- 1 Sample state, value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

- 2 Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(v^\pi - \hat{v}(s, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$$

Converges to least squares solution

$$\mathbf{w}^\pi = \underset{\mathbf{w}}{\operatorname{argmin}} LS(\mathbf{w})$$

Experience Replay in Deep Q-Networks (DQN)

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

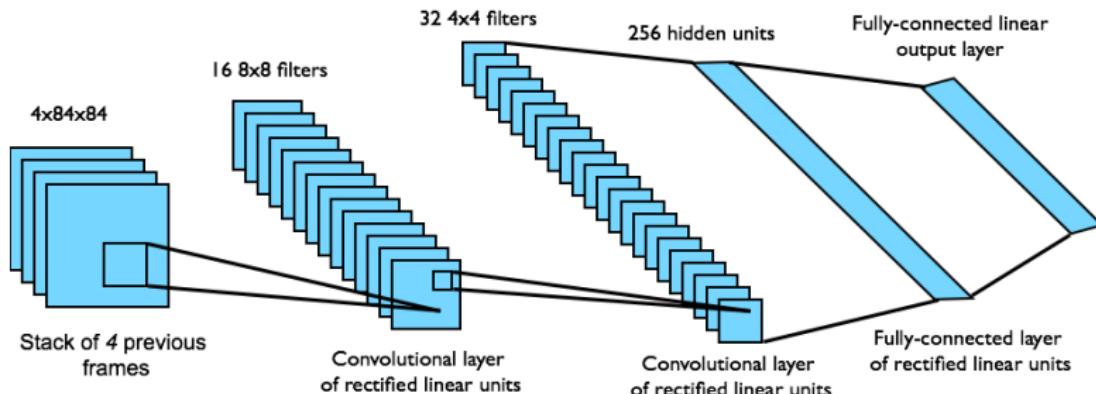
- Take action a_t according to ϵ -greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory \mathcal{D}
- Sample random mini-batch of transitions (s, a, r, s') from \mathcal{D}
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning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

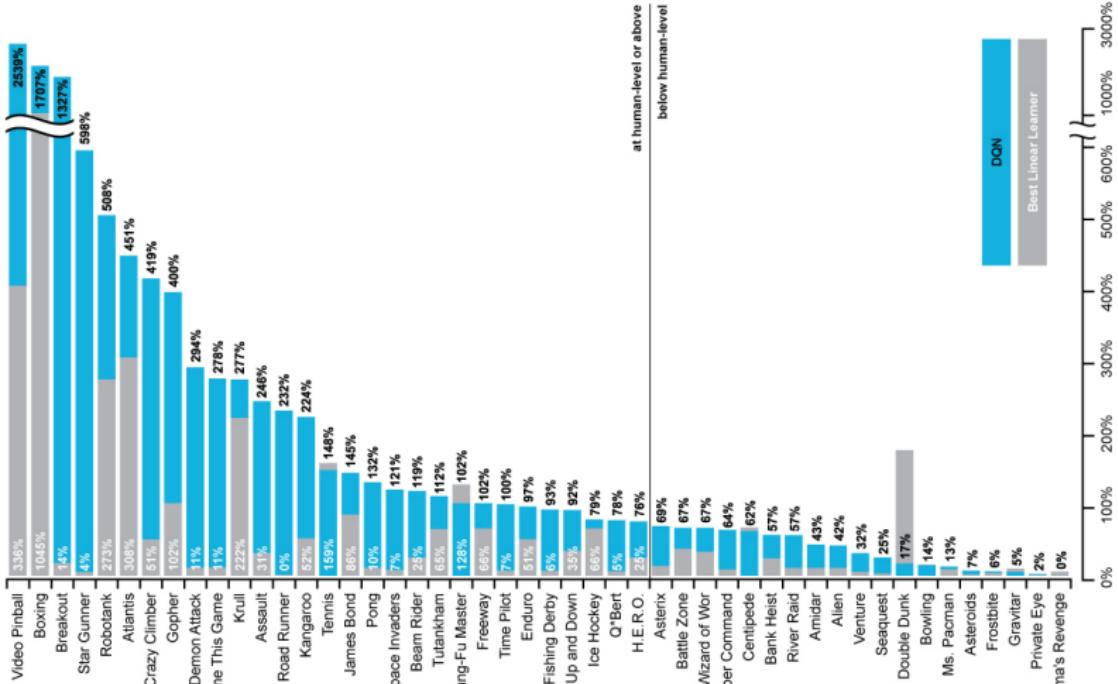
DQN in Atari

- 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

DQN Results in Atari



How much does DQN help?

| | 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 |

Linear Least Squares Prediction

- Experience replay finds least squares solution
- But it may take many iterations
- Using *linear* value function approximation $\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^\top \mathbf{w}$
- We can solve the least squares solution directly

Linear Least Squares Prediction (2)

- At minimum of $LS(\mathbf{w})$, the expected update must be zero

$$\mathbb{E}_{\mathcal{D}} [\Delta \mathbf{w}] = 0$$

$$\alpha \sum_{t=1}^T \mathbf{x}(s_t)(v_t^\pi - \mathbf{x}(s_t)^\top \mathbf{w}) = 0$$

$$\sum_{t=1}^T \mathbf{x}(s_t)v_t^\pi = \sum_{t=1}^T \mathbf{x}(s_t)\mathbf{x}(s_t)^\top \mathbf{w}$$

$$\mathbf{w} = \left(\sum_{t=1}^T \mathbf{x}(s_t)\mathbf{x}(s_t)^\top \right)^{-1} \sum_{t=1}^T \mathbf{x}(s_t)v_t^\pi$$

- For N features, direct solution time is $O(N^3)$
- Incremental solution time is $O(N^2)$ using Sherman-Morrison

Linear Least Squares Prediction Algorithms

- We do not know true values v_t^π
- In practice, our “training data” must use noisy or biased samples of v_t^π

LSMC Least Squares Monte-Carlo uses return

$$v_t^\pi \approx G_t$$

LSTD Least Squares Temporal-Difference uses TD target

$$v_t^\pi \approx R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$$

LSTD(λ) Least Squares TD(λ) uses λ -return

$$v_t^\pi \approx G_t^\lambda$$

- In each case solve directly for fixed point of MC / TD / TD(λ)

Linear Least Squares Prediction Algorithms (2)

LSMC

$$0 = \sum_{t=1}^T \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^T \mathbf{x}(S_t) \mathbf{x}(S_t)^\top \right)^{-1} \sum_{t=1}^T \mathbf{x}(S_t) G_t$$

LSTD

$$0 = \sum_{t=1}^T \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^T \mathbf{x}(S_t) (\mathbf{x}(S_t) - \gamma \mathbf{x}(S_{t+1}))^\top \right)^{-1} \sum_{t=1}^T \mathbf{x}(S_t) R_{t+1}$$

LSTD(λ)

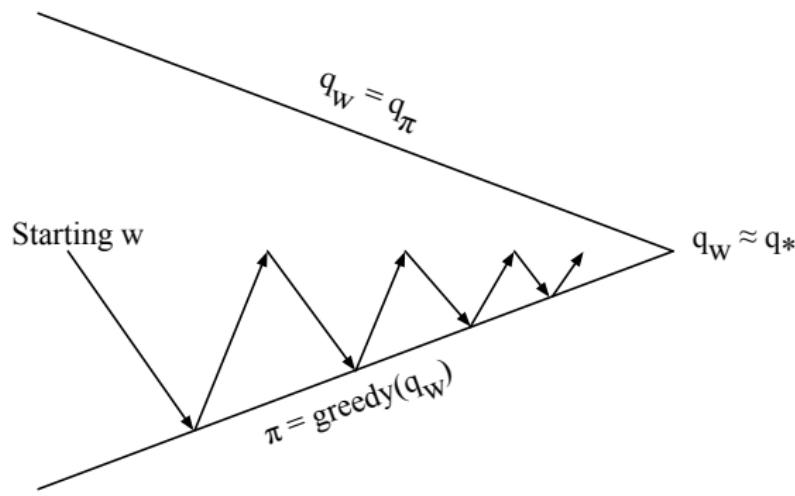
$$0 = \sum_{t=1}^T \alpha \delta_t E_t$$

$$\mathbf{w} = \left(\sum_{t=1}^T E_t (\mathbf{x}(S_t) - \gamma \mathbf{x}(S_{t+1}))^\top \right)^{-1} \sum_{t=1}^T E_t R_{t+1}$$

Convergence of Linear Least Squares Prediction Algorithms

| On/Off-Policy | Algorithm | Table Lookup | Linear | Non-Linear |
|---------------|-----------|--------------|--------|------------|
| On-Policy | MC | ✓ | ✓ | ✓ |
| | LSMC | ✓ | ✓ | - |
| | TD | ✓ | ✓ | ✗ |
| | LSTD | ✓ | ✓ | - |
| Off-Policy | MC | ✓ | ✓ | ✓ |
| | LSMC | ✓ | ✓ | - |
| | TD | ✓ | ✗ | ✗ |
| | LSTD | ✓ | ✓ | - |

Least Squares Policy Iteration



Policy evaluation Policy evaluation by least squares Q-learning

Policy improvement Greedy policy improvement

Least Squares Action-Value Function Approximation

- Approximate action-value function $q_\pi(s, a)$
- using linear combination of features $\mathbf{x}(s, a)$

$$\hat{q}(s, a, \mathbf{w}) = \mathbf{x}(s, a)^\top \mathbf{w} \approx q_\pi(s, a)$$

- Minimise least squares error between $\hat{q}(s, a, \mathbf{w})$ and $q_\pi(s, a)$
- from experience generated using policy π
- consisting of $\langle (state, action), value \rangle$ pairs

$$\mathcal{D} = \{\langle (s_1, a_1), v_1^\pi \rangle, \langle (s_2, a_2), v_2^\pi \rangle, \dots, \langle (s_T, a_T), v_T^\pi \rangle\}$$

Least Squares Control

- For policy evaluation, we want to efficiently use all experience
- For control, we also want to improve the policy
- This experience is generated from many policies
- So to evaluate $q_{\pi}(S, A)$ we must learn **off-policy**
- We use the same idea as Q-learning:
 - Use experience generated by old policy
 $S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{old}$
 - Consider alternative successor action $A' = \pi_{new}(S_{t+1})$
 - Update $\hat{q}(S_t, A_t, \mathbf{w})$ towards value of alternative action
 $R_{t+1} + \gamma \hat{q}(S_{t+1}, A', \mathbf{w})$

Least Squares Q-Learning

- Consider the following linear Q-learning update

$$\delta = R_{t+1} + \gamma \hat{q}(S_{t+1}, \pi(S_{t+1}), \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})$$

$$\Delta \mathbf{w} = \alpha \delta \mathbf{x}(S_t, A_t)$$

- LSTDQ algorithm: solve for total update = zero

$$0 = \sum_{t=1}^T \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, \pi(S_{t+1}), \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})) \mathbf{x}(S_t, A_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^T \mathbf{x}(S_t, A_t) (\mathbf{x}(S_t, A_t) - \gamma \mathbf{x}(S_{t+1}, \pi(S_{t+1})))^\top \right)^{-1} \sum_{t=1}^T \mathbf{x}(S_t, A_t) R_{t+1}$$

Least Squares Policy Iteration Algorithm

- The following pseudocode uses LSTDQ for policy evaluation
- It repeatedly re-evaluates experience \mathcal{D} with different policies

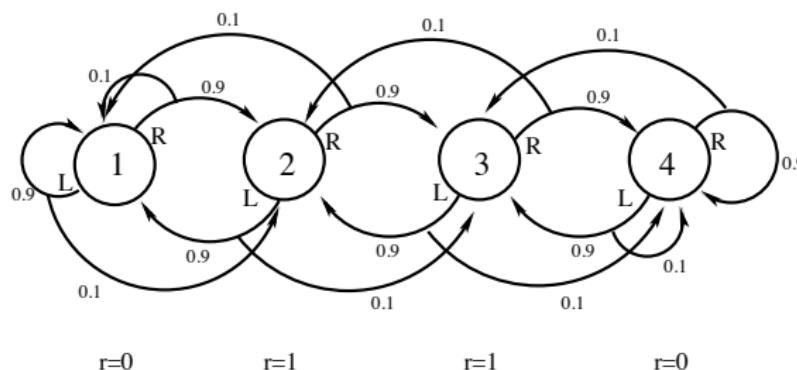
```
function LSPI-TD( $\mathcal{D}, \pi_0$ )
     $\pi' \leftarrow \pi_0$ 
    repeat
         $\pi \leftarrow \pi'$ 
         $Q \leftarrow \text{LSTDQ}(\pi, \mathcal{D})$ 
        for all  $s \in \mathcal{S}$  do
             $\pi'(s) \leftarrow \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q(s, a)$ 
        end for
    until ( $\pi \approx \pi'$ )
    return  $\pi$ 
end function
```

Convergence of Control Algorithms

| Algorithm | Table Lookup | Linear | Non-Linear |
|---------------------|--------------|--------|------------|
| Monte-Carlo Control | ✓ | (✓) | ✗ |
| Sarsa | ✓ | (✓) | ✗ |
| Q-learning | ✓ | ✗ | ✗ |
| LSPI | ✓ | (✓) | - |

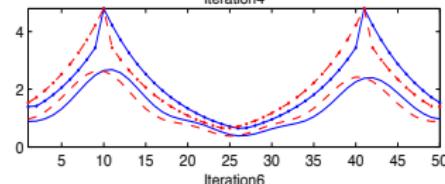
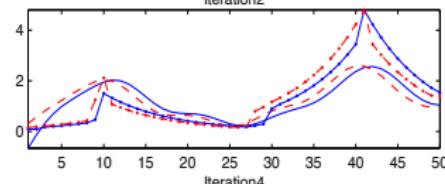
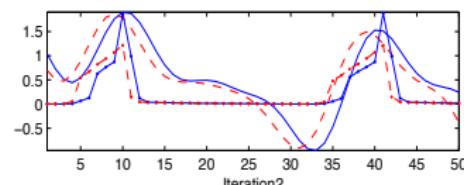
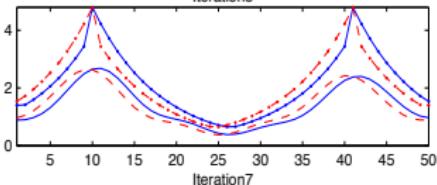
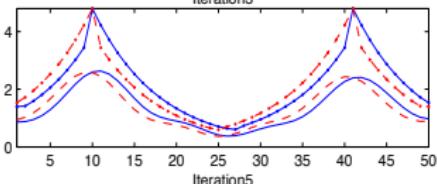
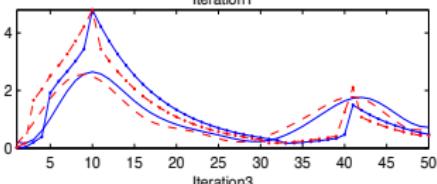
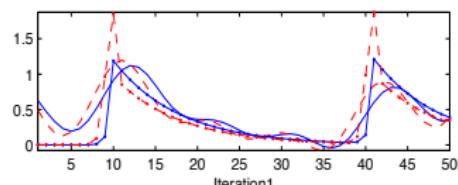
(✓) = chatters around near-optimal value function

Chain Walk Example

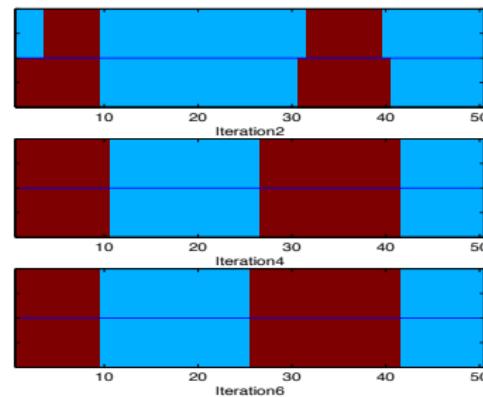
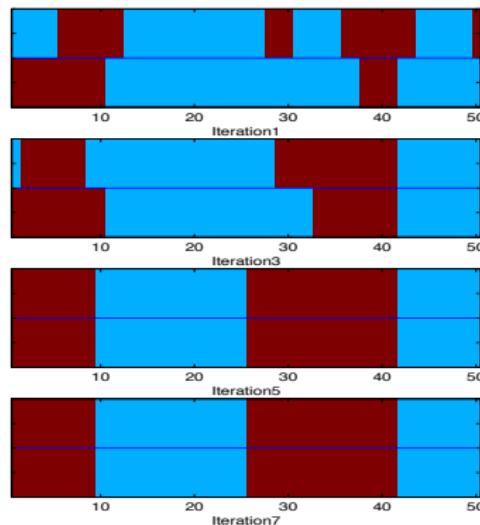


- Consider the 50 state version of this problem
- Reward $+1$ in states 10 and 41, 0 elsewhere
- Optimal policy: R (1-9), L (10-25), R (26-41), L (42, 50)
- Features: 10 evenly spaced Gaussians ($\sigma = 4$) for each action
- Experience: 10,000 steps from random walk policy

LSPI in Chain Walk: Action-Value Function



LSPI in Chain Walk: Policy



Questions?

Lecture 7: Policy Gradient

David Silver

Outline

- 1** Introduction
- 2** Finite Difference Policy Gradient
- 3** Monte-Carlo Policy Gradient
- 4** Actor-Critic Policy Gradient

Policy-Based Reinforcement Learning

- In the last lecture we approximated the value or action-value function using parameters θ ,

$$V_\theta(s) \approx V^\pi(s)$$

$$Q_\theta(s, a) \approx Q^\pi(s, a)$$

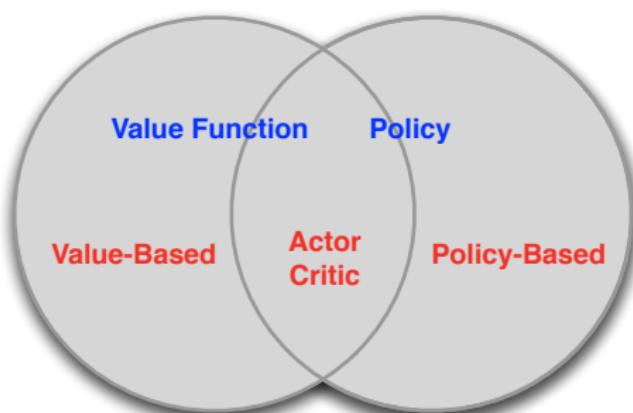
- A policy was generated directly from the value function
 - e.g. using ϵ -greedy
- In this lecture we will directly parametrise the **policy**

$$\pi_\theta(s, a) = \mathbb{P}[a | s, \theta]$$

- We will focus again on **model-free** reinforcement learning

Value-Based and Policy-Based RL

- Value Based
 - Learnt Value Function
 - Implicit policy
(e.g. ϵ -greedy)
- Policy Based
 - No Value Function
 - Learnt Policy
- Actor-Critic
 - Learnt Value Function
 - Learnt Policy



Advantages of Policy-Based RL

Advantages:

- Better convergence properties
- Effective in high-dimensional or continuous action spaces
- Can learn stochastic policies

Disadvantages:

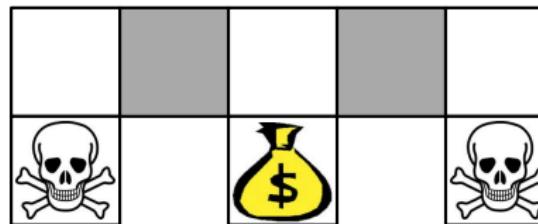
- Typically converge to a local rather than global optimum
- Evaluating a policy is typically inefficient and high variance

Example: Rock-Paper-Scissors



- 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 (i.e. Nash equilibrium)

Example: Aliased Gridworld (1)



- The agent cannot differentiate the grey states
- Consider features of the following form (for all N, E, S, W)

$$\phi(s, a) = \mathbf{1}(\text{wall to N}, a = \text{move E})$$

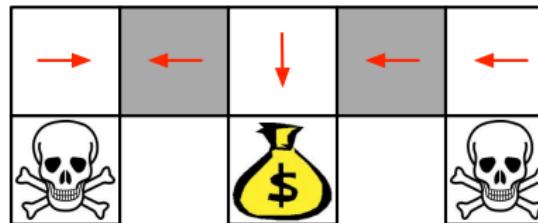
- Compare value-based RL, using an approximate value function

$$Q_\theta(s, a) = f(\phi(s, a), \theta)$$

- To policy-based RL, using a parametrised policy

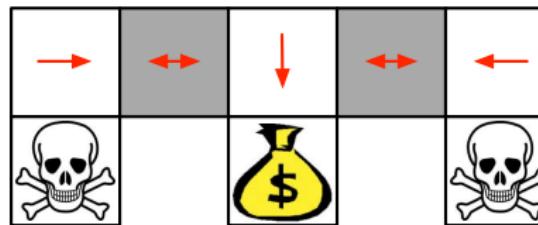
$$\pi_\theta(s, a) = g(\phi(s, a), \theta)$$

Example: Aliased Gridworld (2)



- Under aliasing, an optimal **deterministic** policy will either
 - move W in both grey states (shown by red arrows)
 - move E in both grey states
- Either way, it can get stuck and *never* reach the money
- Value-based RL learns a near-deterministic policy
 - e.g. greedy or ϵ -greedy
- So it will traverse the corridor for a long time

Example: Aliased Gridworld (3)



- An optimal **stochastic** policy will randomly move E or W in grey states

$$\pi_{\theta}(\text{wall to N and S, move E}) = 0.5$$

$$\pi_{\theta}(\text{wall to N and S, move W}) = 0.5$$

- It will reach the goal state in a few steps with high probability
- Policy-based RL can learn the optimal stochastic policy

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 Optimisation

- Policy based reinforcement learning is an **optimisation** problem
- Find θ that maximises $J(\theta)$
- Some approaches do not use gradient
 - Hill climbing
 - Simplex / amoeba / Nelder Mead
 - Genetic algorithms
- Greater efficiency often possible using gradient
 - Gradient descent
 - Conjugate gradient
 - Quasi-newton
- We focus on gradient descent, many extensions possible
- And on methods that exploit sequential structure

Policy Gradient

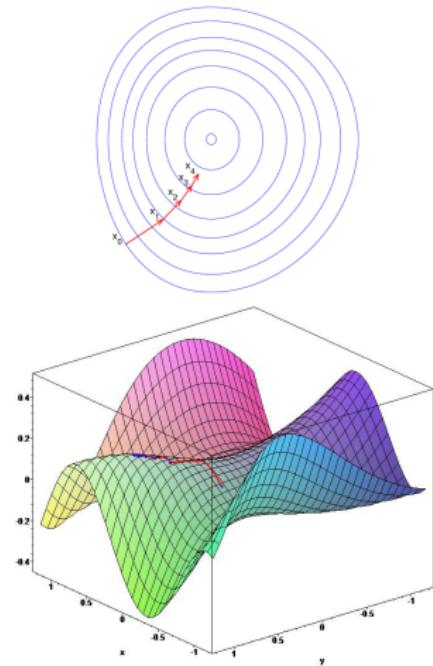
- Let $J(\theta)$ be any policy objective function
- Policy gradient algorithms search for a *local* maximum in $J(\theta)$ by ascending the gradient of the policy, w.r.t. parameters θ

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

- 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

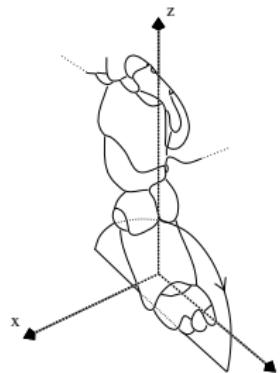
- To evaluate policy gradient of $\pi_\theta(s, a)$
- For each dimension $k \in [1, n]$
 - Estimate k th partial derivative of objective function w.r.t. θ
 - By perturbing θ by small amount ϵ in k th dimension

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

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

Training AIBO to 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

AIBO Walk Policies

- Before training
- During training
- After training

Score Function

- 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

- We will use a softmax policy as a running example
- 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

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

Gaussian Policy

- In continuous action spaces, a Gaussian policy is natural
- Mean is a linear combination of state features $\mu(s) = \phi(s)^\top \theta$
- Variance may be fixed σ^2 , or can also parametrised
- Policy is Gaussian, $a \sim \mathcal{N}(\mu(s), \sigma^2)$
- The score function is

$$\nabla_\theta \log \pi_\theta(s, a) = \frac{(a - \mu(s))\phi(s)}{\sigma^2}$$

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

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

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

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 policy gradient theorem
- 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 θ 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**

$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$

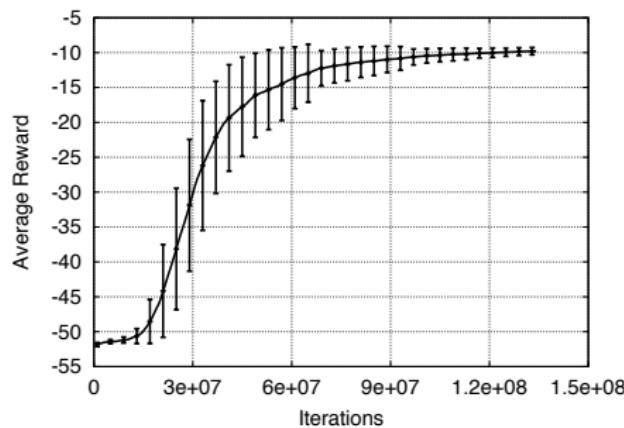
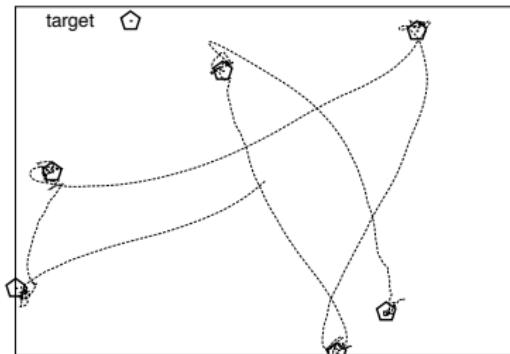
end for

end for

return θ

end function

Puck World Example



- Continuous actions exert small force on puck
- Puck is rewarded for getting close to target
- Target location is reset every 30 seconds
- Policy is trained using variant of Monte-Carlo policy gradient

Reducing Variance Using a Critic

- Monte-Carlo policy gradient still has high variance
- We use a **critic** to estimate the action-value function,

$$Q_w(s, a) \approx Q^{\pi_\theta}(s, a)$$

- Actor-critic algorithms maintain *two* sets of parameters
 - Critic** Updates action-value function parameters w
 - Actor** Updates policy parameters θ , in direction suggested by critic
- Actor-critic algorithms follow an *approximate* policy gradient

$$\nabla_{\theta} J(\theta) \approx \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) \ Q_w(s, a)]$$
$$\Delta \theta = \alpha \nabla_{\theta} \log \pi_{\theta}(s, a) \ Q_w(s, a)$$

Estimating the Action-Value Function

- The critic is solving a familiar problem: policy evaluation
- How good is policy π_θ for current parameters θ ?
- This problem was explored in previous two lectures, e.g.
 - Monte-Carlo policy evaluation
 - Temporal-Difference learning
 - $TD(\lambda)$
- Could also use e.g. least-squares policy evaluation

Action-Value Actor-Critic

- Simple actor-critic algorithm based on action-value critic
- Using linear value fn approx. $Q_w(s, a) = \phi(s, a)^\top w$
 - Critic** Updates w by linear TD(0)
 - Actor** Updates θ by policy gradient

function QAC

 Initialise s, θ

 Sample $a \sim \pi_\theta$

for each step **do**

 Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,a}$.

 Sample action $a' \sim \pi_\theta(s', a')$

$\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$

$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$

$w \leftarrow w + \beta \delta \phi(s, a)$

$a \leftarrow a', s \leftarrow s'$

end for

end function

Bias in Actor-Critic Algorithms

- Approximating the policy gradient introduces bias
- A biased policy gradient may not find the right solution
 - e.g. if $Q_w(s, a)$ uses aliased features, can we solve gridworld example?
- Luckily, if we choose value function approximation carefully
- Then we can avoid introducing any bias
- i.e. We can still follow the *exact* policy gradient

Compatible Function Approximation

Theorem (Compatible Function Approximation Theorem)

If the following two conditions are satisfied:

- 1 Value function approximator is **compatible** to the policy

$$\nabla_w Q_w(s, a) = \nabla_\theta \log \pi_\theta(s, a)$$

- 2 Value function parameters w minimise the mean-squared error

$$\varepsilon = \mathbb{E}_{\pi_\theta} [(Q^{\pi_\theta}(s, a) - Q_w(s, a))^2]$$

Then the policy gradient is exact,

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

Proof of Compatible Function Approximation Theorem

If w is chosen to minimise mean-squared error, gradient of ε w.r.t. w must be zero,

$$\nabla_w \varepsilon = 0$$

$$\mathbb{E}_{\pi_\theta} [(Q^\theta(s, a) - Q_w(s, a)) \nabla_w Q_w(s, a)] = 0$$

$$\mathbb{E}_{\pi_\theta} [(Q^\theta(s, a) - Q_w(s, a)) \nabla_\theta \log \pi_\theta(s, a)] = 0$$

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

So $Q_w(s, a)$ can be substituted directly into the policy gradient,

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

Reducing Variance Using a Baseline

- We subtract a baseline function $B(s)$ from the policy gradient
- This can reduce variance, without changing expectation

$$\begin{aligned}\mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) B(s)] &= \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s, a) B(s) \\ &= \sum_{s \in \mathcal{S}} d^{\pi_\theta} B(s) \nabla_\theta \sum_{a \in \mathcal{A}} \pi_\theta(s, a) \\ &= 0\end{aligned}$$

- A good baseline is the state value function $B(s) = V^{\pi_\theta}(s)$
- So we can rewrite the policy gradient using the **advantage function** $A^{\pi_\theta}(s, a)$

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s)$$

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

Estimating the Advantage Function (1)

- The advantage function can significantly reduce variance of policy gradient
- So the critic should really estimate the advantage function
- For example, by estimating *both* $V^{\pi_\theta}(s)$ and $Q^{\pi_\theta}(s, a)$
- Using two function approximators and two parameter vectors,

$$V_v(s) \approx V^{\pi_\theta}(s)$$

$$Q_w(s, a) \approx Q^{\pi_\theta}(s, a)$$

$$A(s, a) = Q_w(s, a) - V_v(s)$$

- And updating *both* value functions by e.g. TD learning

Estimating the Advantage Function (2)

- For the true value function $V^{\pi_\theta}(s)$, the TD error δ^{π_θ}

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$$

- is an unbiased estimate of the advantage function

$$\begin{aligned}\mathbb{E}_{\pi_\theta} [\delta^{\pi_\theta} | s, a] &= \mathbb{E}_{\pi_\theta} [r + \gamma V^{\pi_\theta}(s') | s, a] - V^{\pi_\theta}(s) \\ &= Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \\ &= A^{\pi_\theta}(s, a)\end{aligned}$$

- So we can use the TD error to compute the policy gradient

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

- In practice we can use an approximate TD error

$$\delta_v = r + \gamma V_v(s') - V_v(s)$$

- This approach only requires one set of critic parameters v

Critics at Different Time-Scales

- Critic can estimate value function $V_\theta(s)$ from many targets at different time-scales From last lecture...
 - For MC, the target is the return v_t

$$\Delta\theta = \alpha(v_t - V_\theta(s))\phi(s)$$

- For TD(0), the target is the TD target $r + \gamma V(s')$

$$\Delta\theta = \alpha(r + \gamma V(s') - V_\theta(s))\phi(s)$$

- For forward-view TD(λ), the target is the λ -return v_t^λ

$$\Delta\theta = \alpha(v_t^\lambda - V_\theta(s))\phi(s)$$

- For backward-view TD(λ), we use eligibility traces

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

$$e_t = \gamma \lambda e_{t-1} + \phi(s_t)$$

$$\Delta\theta = \alpha \delta_t e_t$$

Actors at Different Time-Scales

- The policy gradient can also be estimated at many time-scales

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

- Monte-Carlo policy gradient uses error from complete return

$$\Delta\theta = \alpha(v_t - V_v(s_t)) \nabla_{\theta} \log \pi_{\theta}(s_t, a_t)$$

- Actor-critic policy gradient uses the one-step TD error

$$\Delta\theta = \alpha(r + \gamma V_v(s_{t+1}) - V_v(s_t)) \nabla_{\theta} \log \pi_{\theta}(s_t, a_t)$$

Policy Gradient with Eligibility Traces

- Just like forward-view $\text{TD}(\lambda)$, we can mix over time-scales

$$\Delta\theta = \alpha(v_t^\lambda - V_v(s_t)) \nabla_\theta \log \pi_\theta(s_t, a_t)$$

- where $v_t^\lambda - V_v(s_t)$ is a biased estimate of advantage fn
- Like backward-view $\text{TD}(\lambda)$, we can also use eligibility traces
 - By equivalence with $\text{TD}(\lambda)$, substituting $\phi(s) = \nabla_\theta \log \pi_\theta(s, a)$

$$\delta = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

$$e_{t+1} = \lambda e_t + \nabla_\theta \log \pi_\theta(s, a)$$

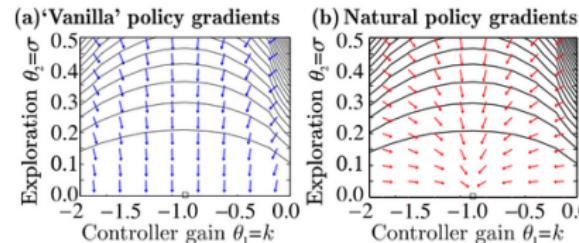
$$\Delta\theta = \alpha \delta e_t$$

- This update can be applied online, to incomplete sequences

Alternative Policy Gradient Directions

- Gradient ascent algorithms can follow *any* ascent direction
- A good ascent direction can significantly speed convergence
- Also, a policy can often be reparametrised without changing action probabilities
- For example, increasing score of all actions in a softmax policy
- The vanilla gradient is sensitive to these reparametrisations

Natural Policy Gradient



- The **natural policy gradient** is parametrisation independent
- It finds ascent direction that is closest to vanilla gradient, when changing policy by a small, fixed amount

$$\nabla_{\theta}^{nat} \pi_{\theta}(s, a) = G_{\theta}^{-1} \nabla_{\theta} \pi_{\theta}(s, a)$$

- where G_{θ} is the Fisher information matrix

$$G_{\theta} = \mathbb{E}_{\pi_{\theta}} \left[\nabla_{\theta} \log \pi_{\theta}(s, a) \nabla_{\theta} \log \pi_{\theta}(s, a)^T \right]$$

Natural Actor-Critic

- Using compatible function approximation,

$$\nabla_w A_w(s, a) = \nabla_\theta \log \pi_\theta(s, a)$$

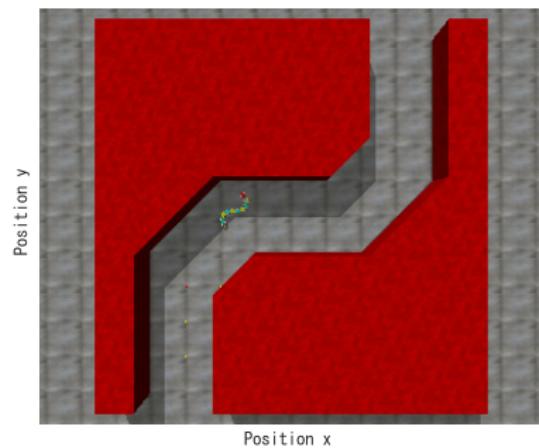
- So the natural policy gradient simplifies,

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) A^{\pi_\theta}(s, a)] \\ &= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a)^T w] \\ &= G_\theta w\end{aligned}$$

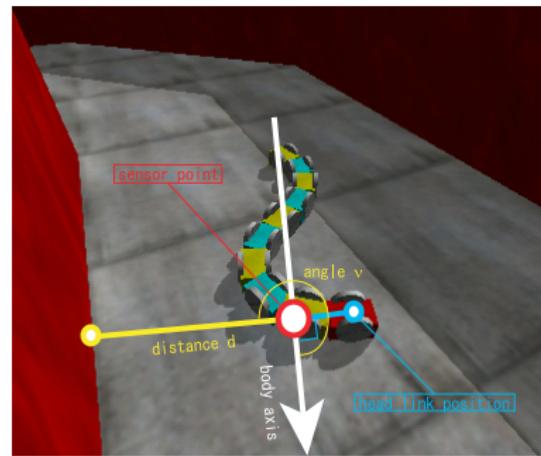
$$\nabla_\theta^{nat} J(\theta) = w$$

- i.e. update actor parameters in direction of critic parameters

Natural Actor Critic in Snake Domain

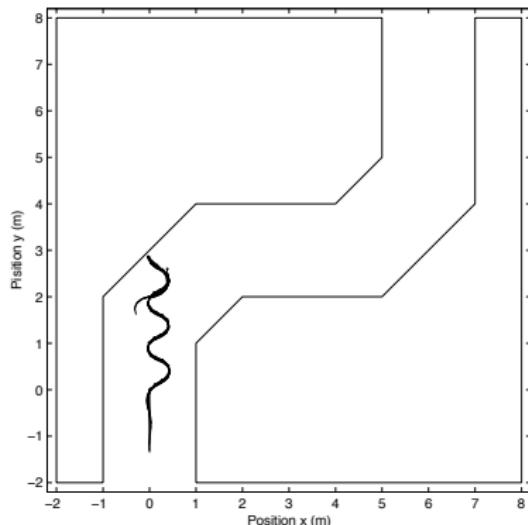


(a) Crank course

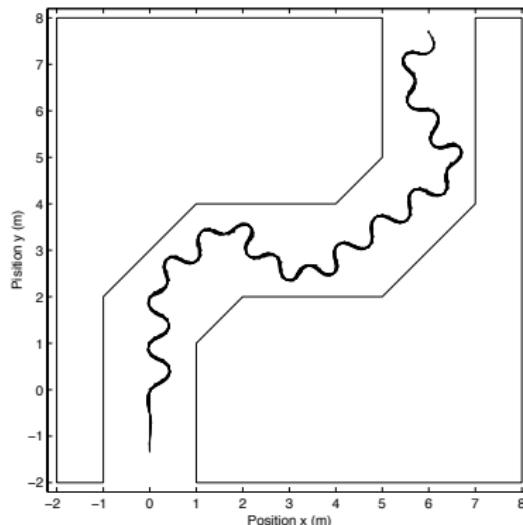


(b) Sensor setting

Natural Actor Critic in Snake Domain (2)

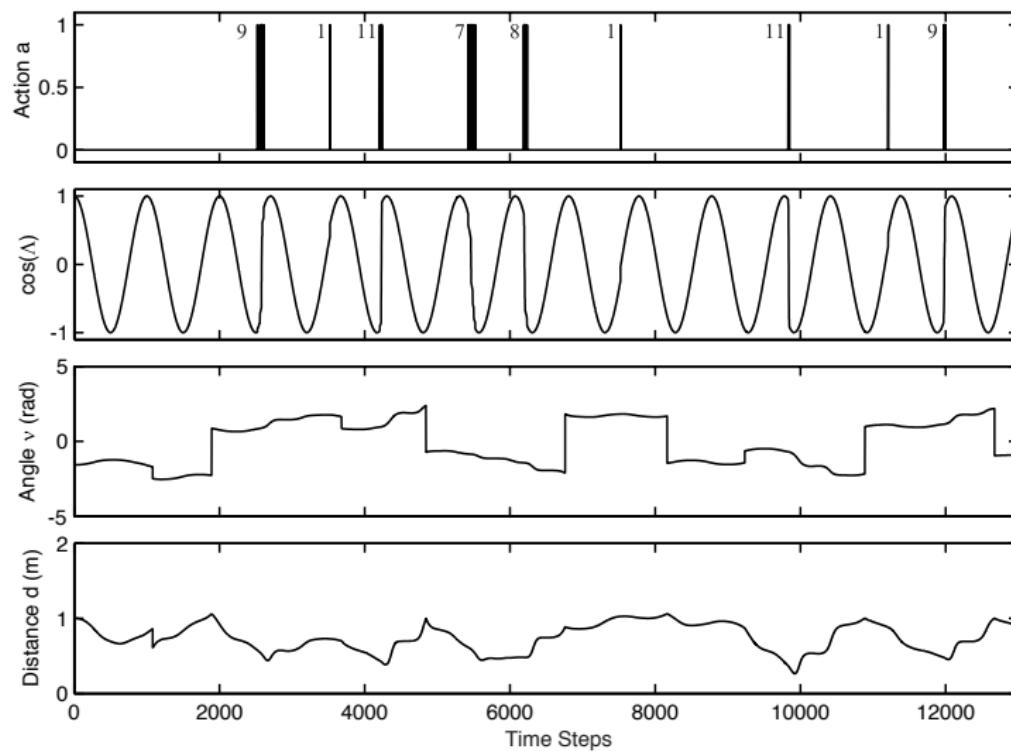


(a) Before learning



(b) After learning

Natural Actor Critic in Snake Domain (3)



Summary of Policy Gradient Algorithms

- The **policy gradient** has many equivalent forms

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) \ v_t] \quad \text{REINFORCE}$$

$$= \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) \ Q^w(s, a)] \quad \text{Q Actor-Critic}$$

$$= \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) \ A^w(s, a)] \quad \text{Advantage Actor-Critic}$$

$$= \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) \ \delta] \quad \text{TD Actor-Critic}$$

$$= \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) \ \delta e] \quad \text{TD}(\lambda) \text{ Actor-Critic}$$

$$G_{\theta}^{-1} \nabla_{\theta} J(\theta) = w \quad \text{Natural Actor-Critic}$$

- Each leads a stochastic gradient ascent algorithm
- Critic uses **policy evaluation** (e.g. MC or TD learning) to estimate $Q^{\pi}(s, a)$, $A^{\pi}(s, a)$ or $V^{\pi}(s)$

Lecture 8: Integrating Learning and Planning

David Silver

Outline

1 Introduction

2 Model-Based Reinforcement Learning

3 Integrated Architectures

4 Simulation-Based Search

Outline

1 Introduction

2 Model-Based Reinforcement Learning

3 Integrated Architectures

4 Simulation-Based Search

Model-Based Reinforcement Learning

- *Last lecture:* learn **policy** directly from experience
- *Previous lectures:* learn **value function** directly from experience
- *This lecture:* learn **model** directly from experience
- and use **planning** to construct a value function or policy
- Integrate learning and planning into a single architecture

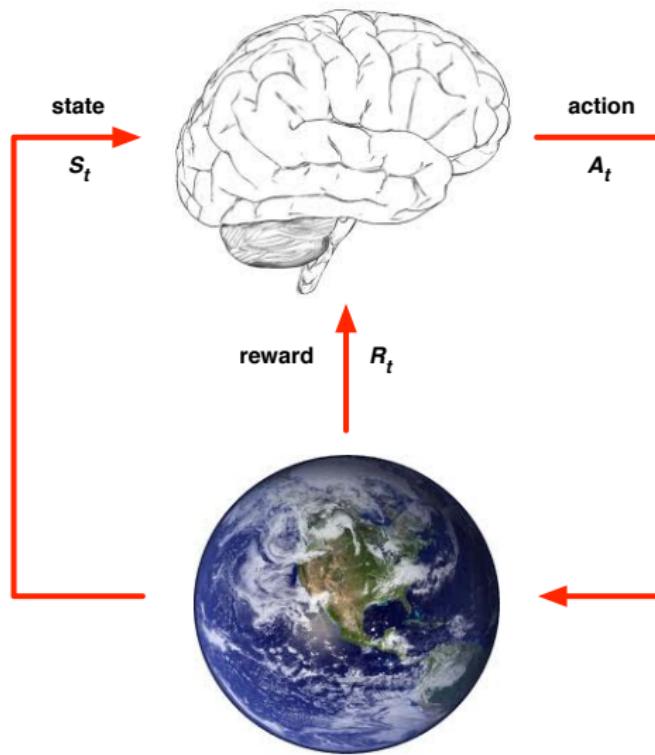
Model-Based and Model-Free RL

- Model-Free RL
 - No model
 - Learn value function (and/or policy) from experience

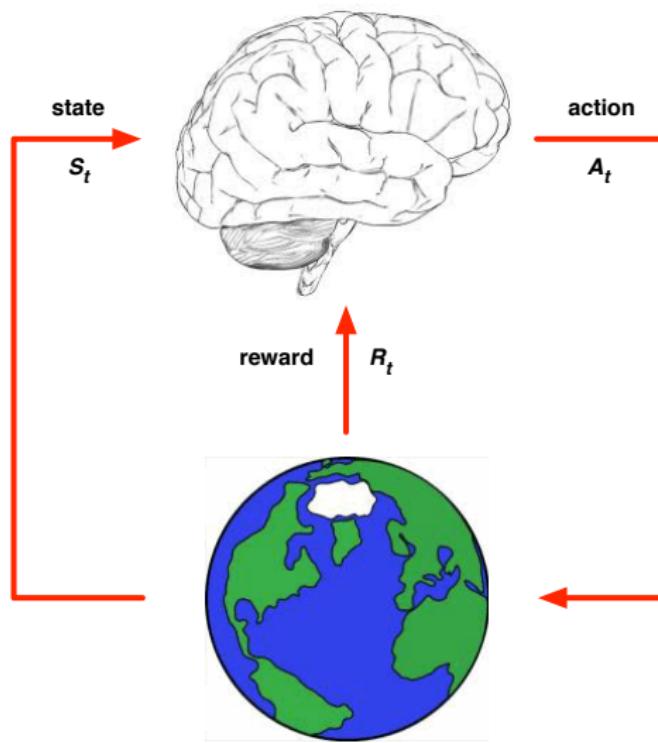
Model-Based and Model-Free RL

- Model-Free RL
 - No model
 - **Learn** value function (and/or policy) from experience
- Model-Based RL
 - Learn a model from experience
 - **Plan** value function (and/or policy) from model

Model-Free RL



Model-Based RL



Outline

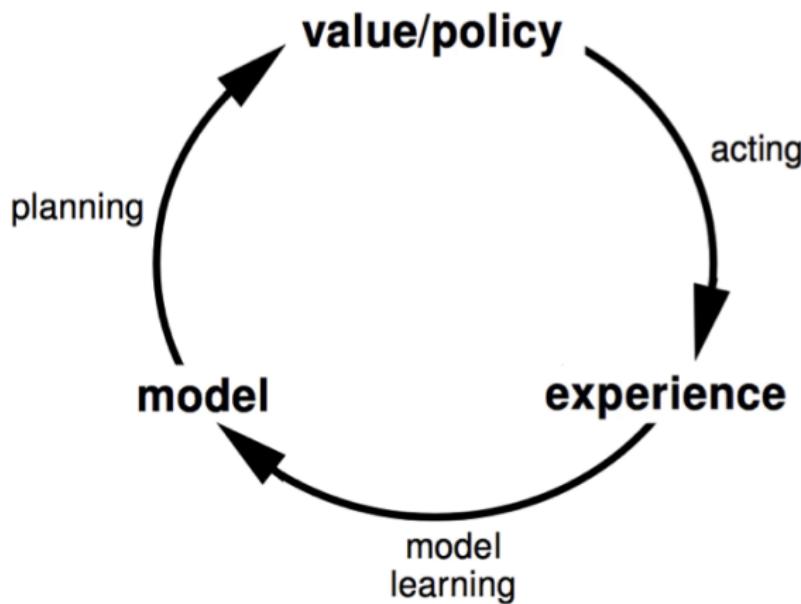
1 Introduction

2 Model-Based Reinforcement Learning

3 Integrated Architectures

4 Simulation-Based Search

Model-Based RL



Advantages of Model-Based RL

Advantages:

- Can efficiently learn model by supervised learning methods
- Can reason about model uncertainty

Disadvantages:

- First learn a model, then construct a value function
⇒ two sources of approximation error

What is a Model?

- A *model* \mathcal{M} is a representation of an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$, parametrized by η
- We will assume state space \mathcal{S} and action space \mathcal{A} are known
- So a model $\mathcal{M} = \langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$ represents state transitions $\mathcal{P}_\eta \approx \mathcal{P}$ and rewards $\mathcal{R}_\eta \approx \mathcal{R}$

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1} \mid S_t, A_t)$$

$$R_{t+1} = \mathcal{R}_\eta(R_{t+1} \mid S_t, A_t)$$

- Typically assume conditional independence between state transitions and rewards

$$\mathbb{P}[S_{t+1}, R_{t+1} \mid S_t, A_t] = \mathbb{P}[S_{t+1} \mid S_t, A_t] \mathbb{P}[R_{t+1} \mid S_t, A_t]$$

Model Learning

- Goal: estimate model \mathcal{M}_η from experience $\{S_1, A_1, R_2, \dots, S_T\}$
- This is a supervised learning problem

$$S_1, A_1 \rightarrow R_2, S_2$$

$$S_2, A_2 \rightarrow R_3, S_3$$

$$\vdots$$

$$S_{T-1}, A_{T-1} \rightarrow R_T, S_T$$

- Learning $s, a \rightarrow r$ is a *regression* problem
- Learning $s, a \rightarrow s'$ is a *density estimation* problem
- Pick loss function, e.g. mean-squared error, KL divergence, ...
- Find parameters η that minimise empirical loss

Examples of Models

- Table Lookup Model
- Linear Expectation Model
- Linear Gaussian Model
- Gaussian Process Model
- Deep Belief Network Model
- ...

Table Lookup Model

- Model is an explicit MDP, $\hat{\mathcal{P}}, \hat{\mathcal{R}}$
- Count visits $N(s, a)$ to each state action pair

$$\hat{\mathcal{P}}_{s,s'}^a = \frac{1}{N(s,a)} \sum_{t=1}^T \mathbf{1}(S_t, A_t, S_{t+1} = s, a, s')$$

$$\hat{\mathcal{R}}_s^a = \frac{1}{N(s,a)} \sum_{t=1}^T \mathbf{1}(S_t, A_t = s, a) R_t$$

- Alternatively
 - At each time-step t , record experience tuple $\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$
 - To sample model, randomly pick tuple matching $\langle s, a, \cdot, \cdot \rangle$

AB Example

Two states A, B ; no discounting; 8 episodes of experience

A, 0, B, 0

B, 1

B, 1

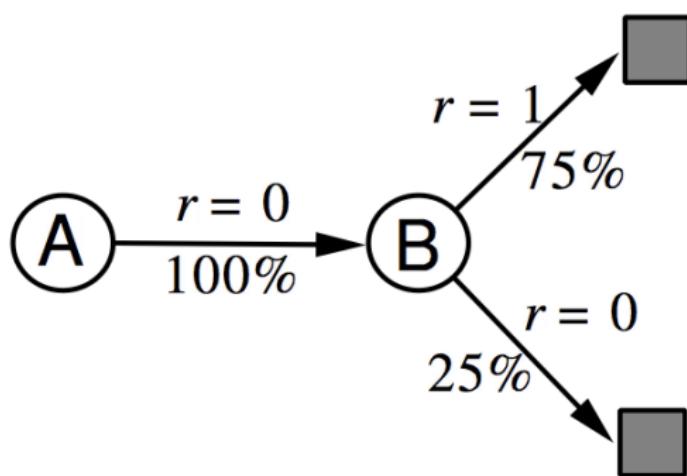
B, 1

B, 1

B, 1

B, 1

B, 0



We have constructed a **table lookup model** from the experience

Planning with a Model

- Given a model $\mathcal{M}_\eta = \langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$
- Solve the MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$
- Using favourite planning algorithm
 - Value iteration
 - Policy iteration
 - Tree search
 - ...

Sample-Based Planning

- A simple but powerful approach to planning
- Use the model **only** to generate samples
- **Sample** experience from model

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1} \mid S_t, A_t)$$

$$R_{t+1} = \mathcal{R}_\eta(R_{t+1} \mid S_t, A_t)$$

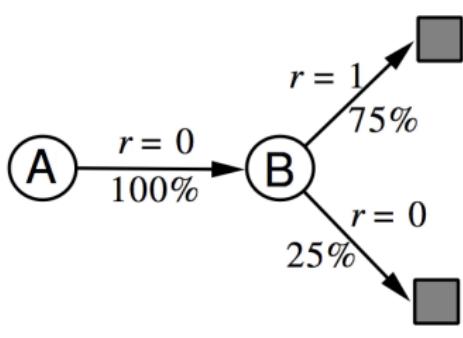
- Apply **model-free** RL to samples, e.g.:
 - Monte-Carlo control
 - Sarsa
 - Q-learning
- Sample-based planning methods are often more efficient

Back to the AB Example

- Construct a table-lookup model from real experience
- Apply model-free RL to sampled experience

Real experience

A, 0, B, 0
 B, 1
 B, 0



Sampled experience

B, 1
 B, 0
 B, 1
 A, 0, B, 1
 B, 1
 A, 0, B, 1
 B, 1
 B, 0

e.g. Monte-Carlo learning: $V(A) = 1, V(B) = 0.75$

Planning with an Inaccurate Model

- Given an imperfect model $\langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle \neq \langle \mathcal{P}, \mathcal{R} \rangle$
- Performance of model-based RL is limited to optimal policy for approximate MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$
- i.e. Model-based RL is only as good as the estimated model
- When the model is inaccurate, planning process will compute a suboptimal policy
- Solution 1: when model is wrong, use model-free RL
- Solution 2: reason explicitly about model uncertainty

Outline

1 Introduction

2 Model-Based Reinforcement Learning

3 Integrated Architectures

4 Simulation-Based Search

Real and Simulated Experience

We consider two sources of experience

Real experience Sampled from environment (true MDP)

$$S' \sim \mathcal{P}_{s,s'}^a$$

$$R = \mathcal{R}_s^a$$

Simulated experience Sampled from model (approximate MDP)

$$S' \sim \mathcal{P}_\eta(S' | S, A)$$

$$R = \mathcal{R}_\eta(R | S, A)$$

Integrating Learning and Planning

- Model-Free RL
 - No model
 - **Learn** value function (and/or policy) from real experience

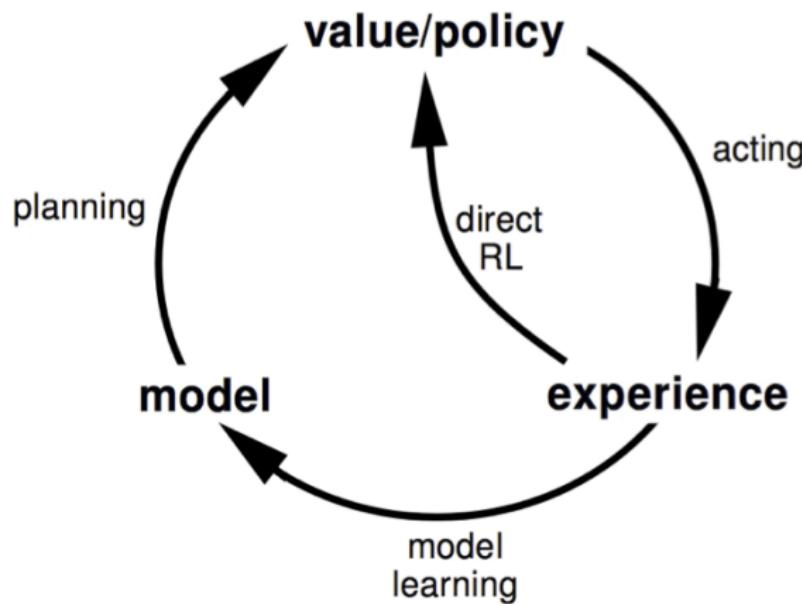
Integrating Learning and Planning

- Model-Free RL
 - No model
 - **Learn** value function (and/or policy) from real experience
- Model-Based RL (using Sample-Based Planning)
 - Learn a model from real experience
 - **Plan** value function (and/or policy) from simulated experience

Integrating Learning and Planning

- Model-Free RL
 - No model
 - **Learn** value function (and/or policy) from real experience
- Model-Based RL (using Sample-Based Planning)
 - Learn a model from real experience
 - **Plan** value function (and/or policy) from simulated experience
- Dyna
 - Learn a model from real experience
 - **Learn and plan** value function (and/or policy) from real and simulated experience

Dyna Architecture



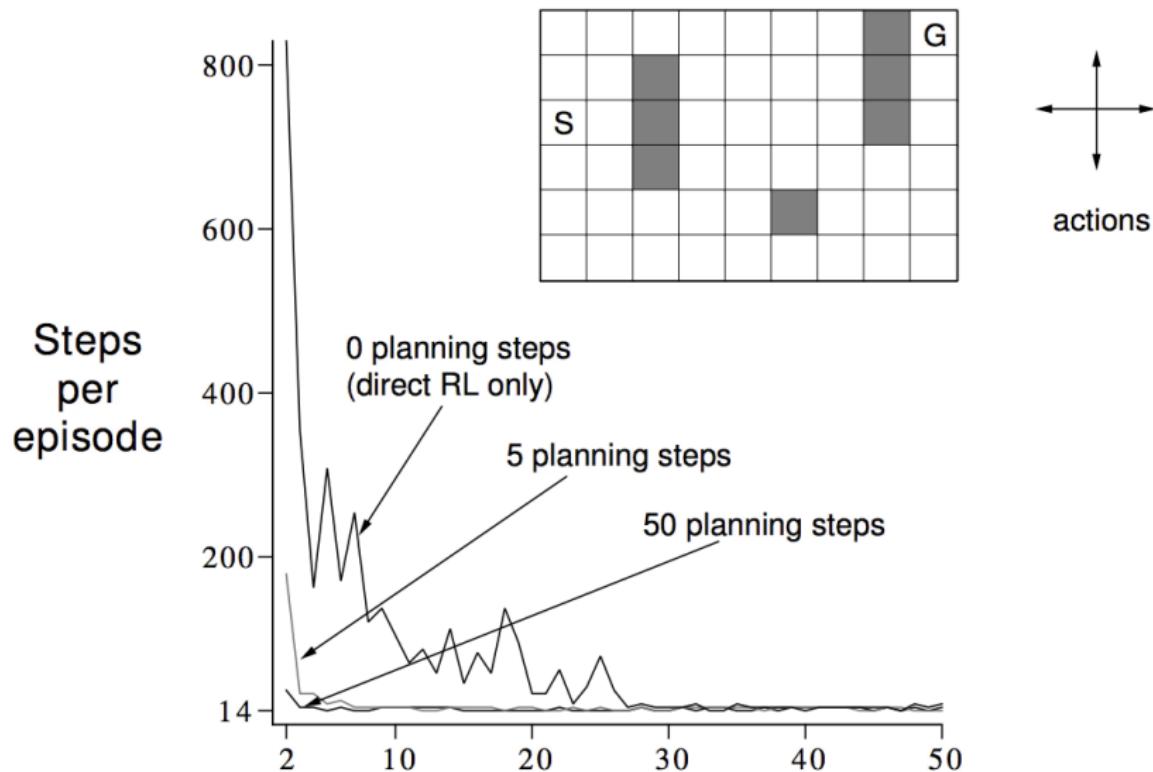
Dyna-Q Algorithm

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Do forever:

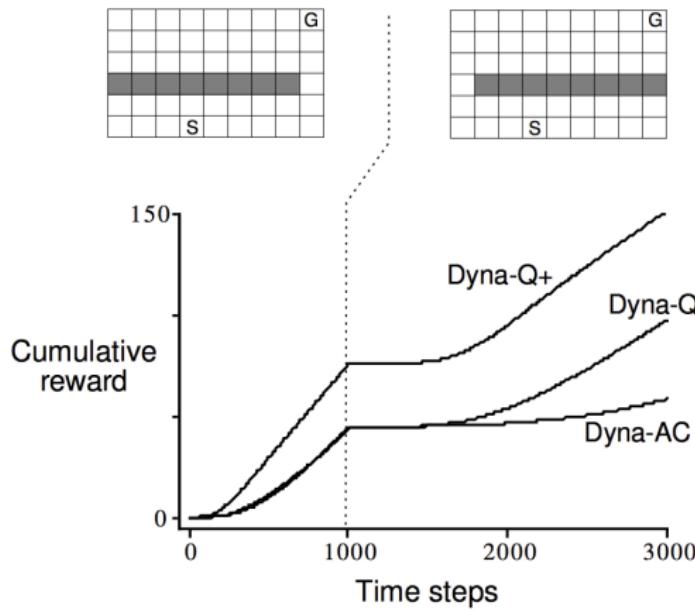
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon\text{-greedy}(S, Q)$
- (c) Execute action A ; observe resultant reward, R , and state, S'
- (d) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
- (e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Repeat n times:
 - $S \leftarrow$ random previously observed state
 - $A \leftarrow$ random action previously taken in S
 - $R, S' \leftarrow Model(S, A)$
 - $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

Dyna-Q on a Simple Maze



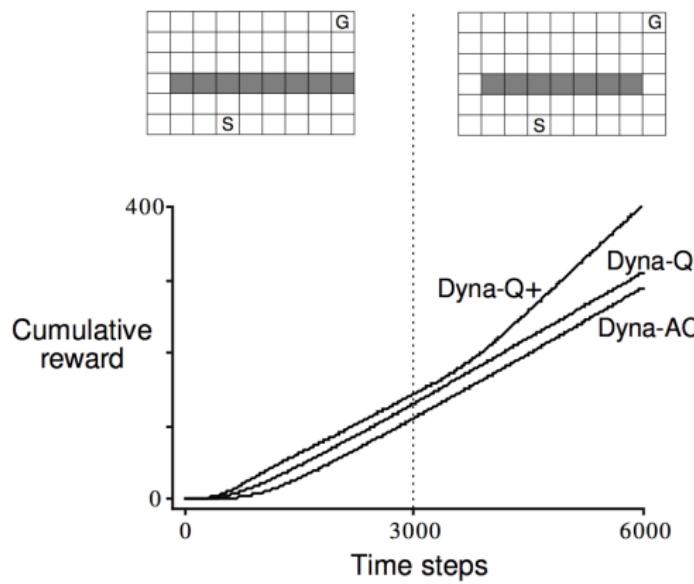
Dyna-Q with an Inaccurate Model

- The changed environment is **harder**



Dyna-Q with an Inaccurate Model (2)

- The changed environment is easier

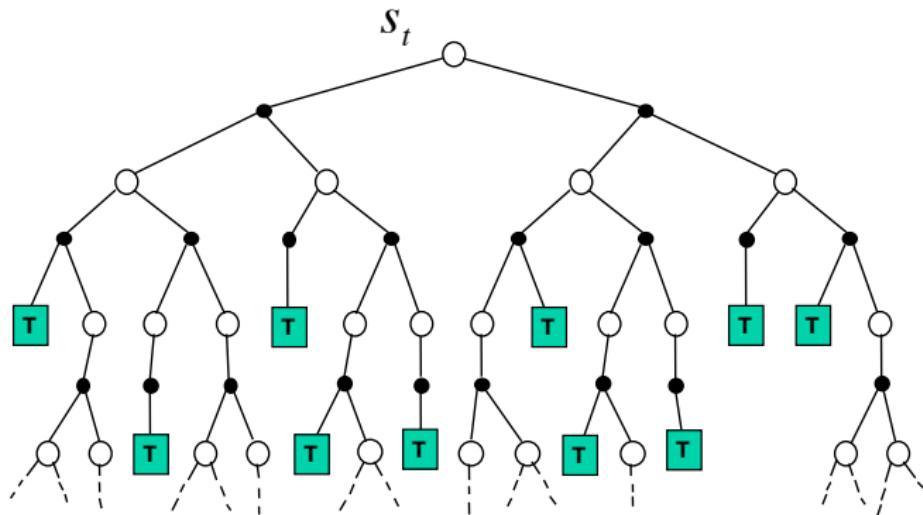


Outline

- 1 Introduction
- 2 Model-Based Reinforcement Learning
- 3 Integrated Architectures
- 4 Simulation-Based Search

Forward Search

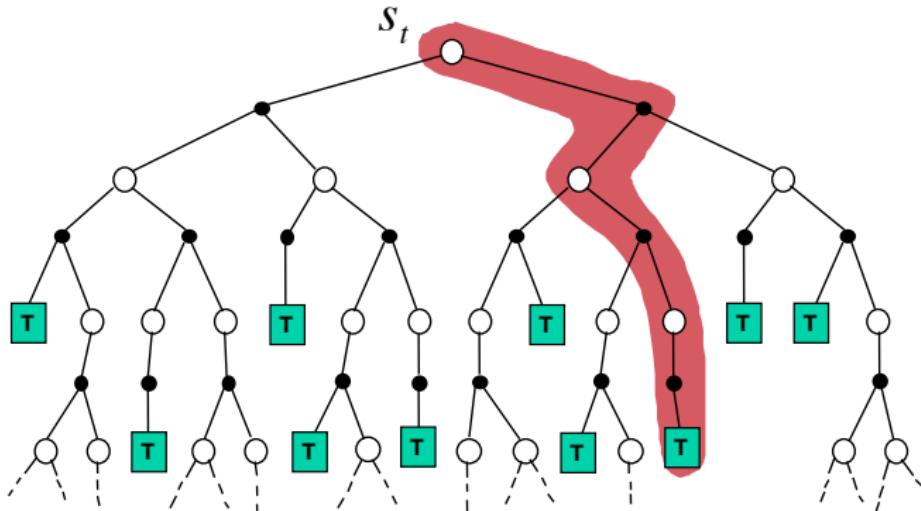
- Forward search algorithms select the best action by lookahead
- They build a search tree with the current state s_t at the root
- Using a model of the MDP to look ahead



- No need to solve whole MDP, just sub-MDP starting from now

Simulation-Based Search

- Forward search paradigm using sample-based planning
- Simulate episodes of experience from now with the model
- Apply model-free RL to simulated episodes



Simulation-Based Search (2)

- Simulate episodes of experience from now with the model

$$\{s_t^k, A_t^k, R_{t+1}^k, \dots, S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu$$

- Apply model-free RL to simulated episodes
 - Monte-Carlo control → Monte-Carlo search
 - Sarsa → TD search

Simple Monte-Carlo Search

- Given a model \mathcal{M}_ν and a **simulation policy** π
- For each action $a \in \mathcal{A}$
 - Simulate K episodes from current (real) state s_t

$$\{\mathbf{s}_t, \mathbf{a}, R_{t+1}^k, S_{t+1}^k, A_{t+1}^k, \dots, S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

- Evaluate actions by mean return (**Monte-Carlo evaluation**)

$$Q(\mathbf{s}_t, \mathbf{a}) = \frac{1}{K} \sum_{k=1}^K G_t \xrightarrow{P} q_\pi(s_t, a)$$

- Select current (real) action with maximum value

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_t, a)$$

Monte-Carlo Tree Search (Evaluation)

- Given a model \mathcal{M}_ν ,
- Simulate K episodes from current state s_t using current simulation policy π

$$\{\textcolor{red}{s_t}, A_t^k, R_{t+1}^k, S_{t+1}^k, \dots, S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

- Build a search tree containing visited states and actions
- Evaluate states $Q(s, a)$ by mean return of episodes from s, a

$$Q(\textcolor{red}{s}, \textcolor{red}{a}) = \frac{1}{N(s, a)} \sum_{k=1}^K \sum_{u=t}^T \mathbf{1}(S_u, A_u = s, a) G_u \xrightarrow{P} q_\pi(s, a)$$

- After search is finished, select current (real) action with maximum value in search tree

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_t, a)$$

Monte-Carlo Tree Search (Simulation)

- In MCTS, the simulation policy π improves
- Each simulation consists of two phases (in-tree, out-of-tree)
 - Tree policy (improves): pick actions to maximise $Q(S, A)$
 - Default policy (fixed): pick actions randomly
- Repeat (each simulation)
 - Evaluate states $Q(S, A)$ by Monte-Carlo evaluation
 - Improve tree policy, e.g. by ϵ – greedy(Q)
- Monte-Carlo control applied to simulated experience
- Converges on the optimal search tree, $Q(S, A) \rightarrow q_*(S, A)$

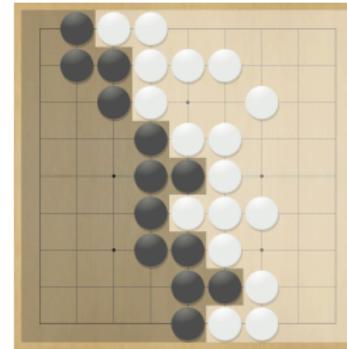
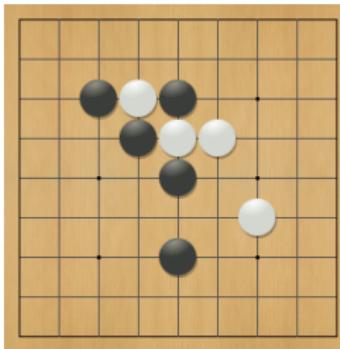
Case Study: the Game of Go

- The ancient oriental game of Go is 2500 years old
- Considered to be the hardest classic board game
- Considered a grand challenge task for AI
(John McCarthy)
- Traditional game-tree search has failed in Go



Rules of Go

- Usually played on 19x19, also 13x13 or 9x9 board
- Simple rules, complex strategy
- Black and white place down stones alternately
- Surrounded stones are captured and removed
- The player with more territory wins the game



Position Evaluation in Go

- How good is a position s ?
- Reward function (undiscounted):

$R_t = 0$ for all non-terminal steps $t < T$

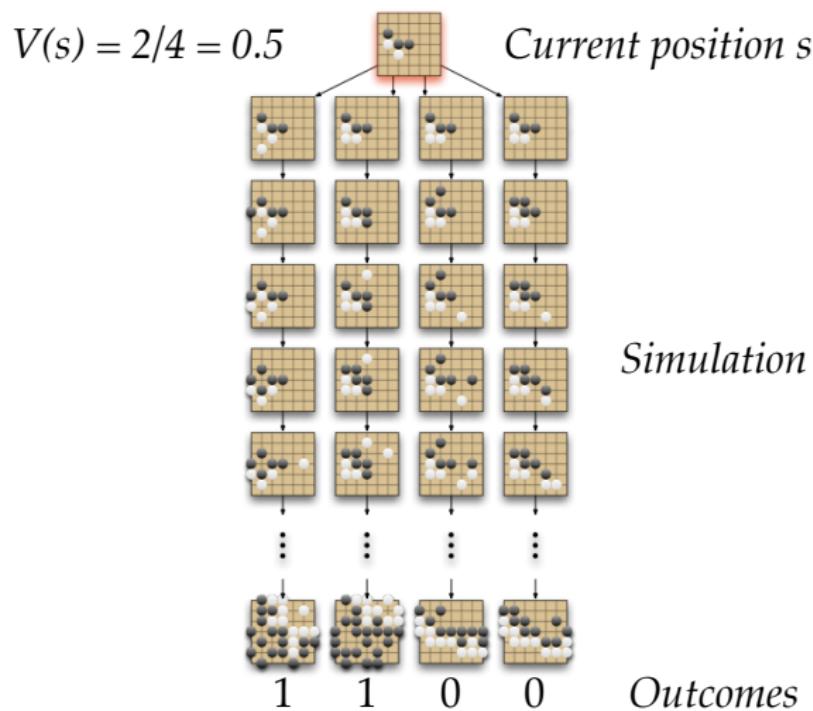
$$R_T = \begin{cases} 1 & \text{if Black wins} \\ 0 & \text{if White wins} \end{cases}$$

- Policy $\pi = \langle \pi_B, \pi_W \rangle$ selects moves for both players
- Value function (how good is position s):

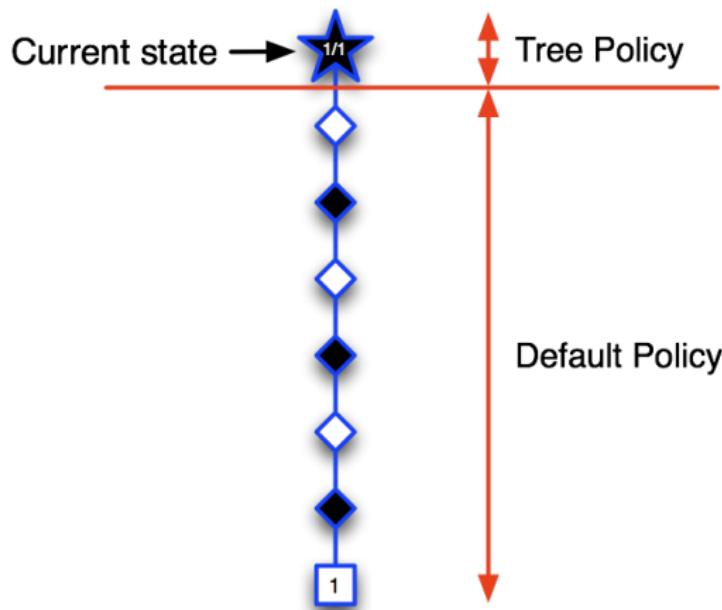
$$v_\pi(s) = \mathbb{E}_\pi [R_T \mid S = s] = \mathbb{P} [\text{Black wins} \mid S = s]$$

$$v_*(s) = \max_{\pi_B} \min_{\pi_W} v_\pi(s)$$

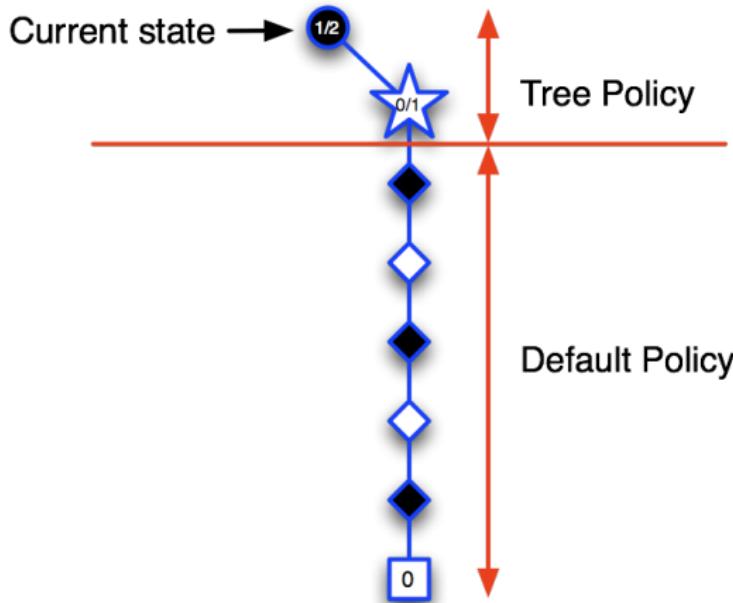
Monte-Carlo Evaluation in Go



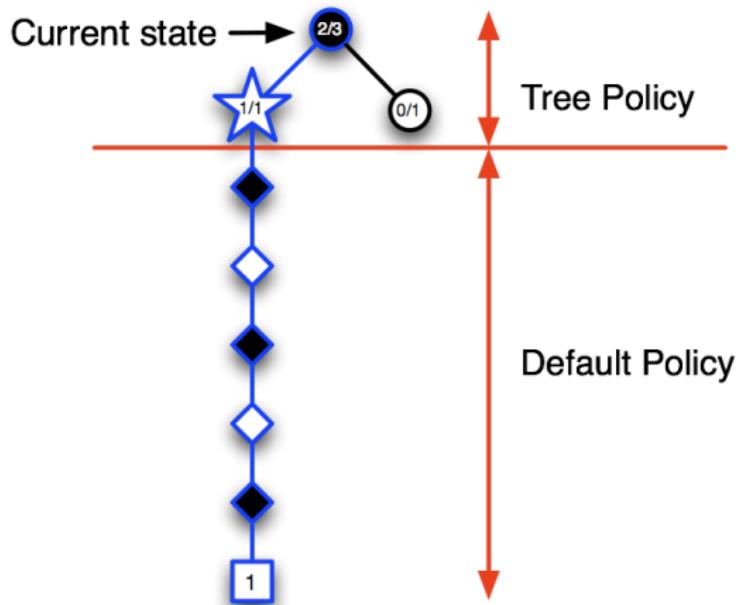
Applying Monte-Carlo Tree Search (1)



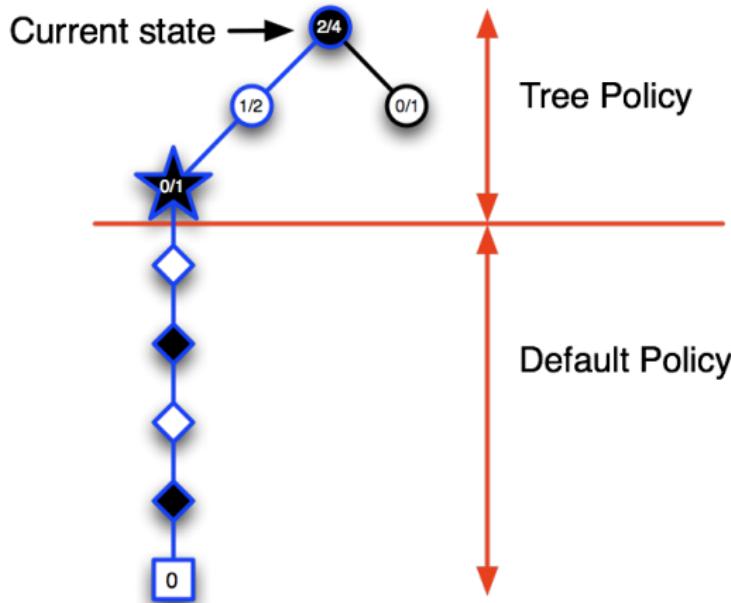
Applying Monte-Carlo Tree Search (2)



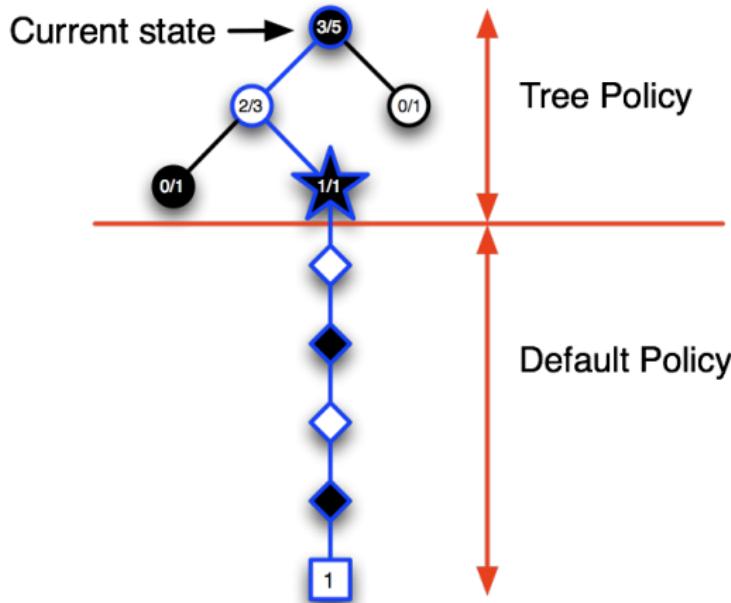
Applying Monte-Carlo Tree Search (3)



Applying Monte-Carlo Tree Search (4)



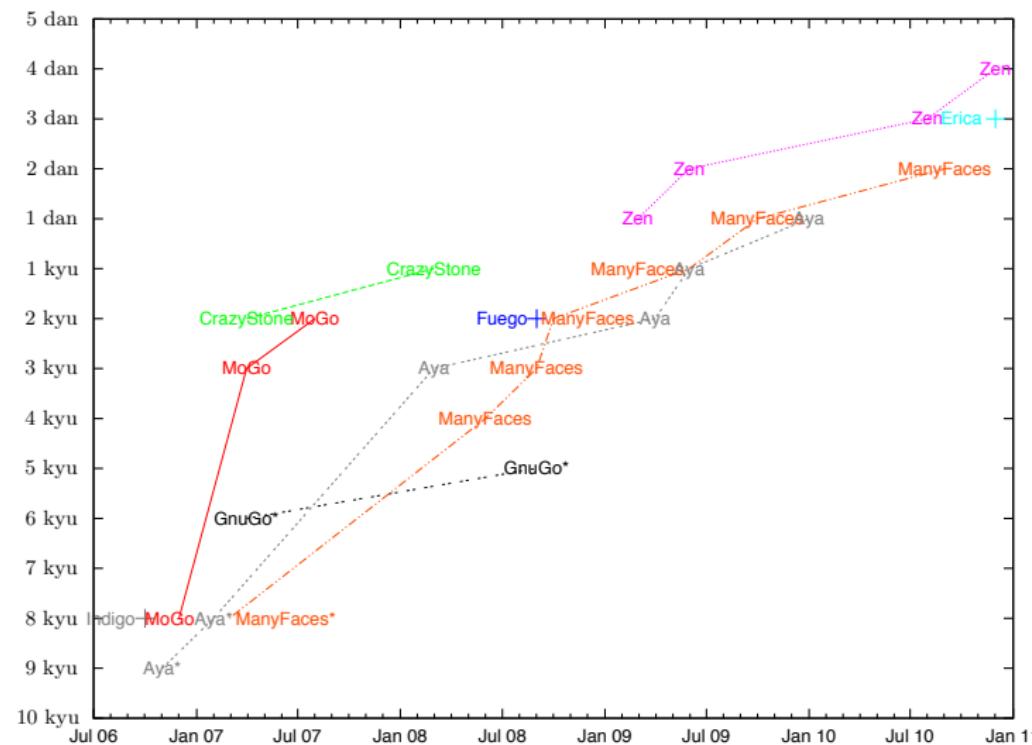
Applying Monte-Carlo Tree Search (5)



Advantages of MC Tree Search

- Highly selective best-first search
- Evaluates states *dynamically* (unlike e.g. DP)
- Uses sampling to break curse of dimensionality
- Works for “black-box” models (only requires samples)
- Computationally efficient, anytime, parallelisable

Example: MC Tree Search in Computer Go



Temporal-Difference Search

- Simulation-based search
- Using TD instead of MC (bootstrapping)
- MC tree search applies MC control to sub-MDP from now
- TD search applies Sarsa to sub-MDP from now

MC vs. TD search

- For model-free reinforcement learning, bootstrapping is helpful
 - TD learning reduces variance but increases bias
 - TD learning is usually more efficient than MC
 - $\text{TD}(\lambda)$ can be much more efficient than MC
- For simulation-based search, bootstrapping is also helpful
 - TD search reduces variance but increases bias
 - TD search is usually more efficient than MC search
 - $\text{TD}(\lambda)$ search can be much more efficient than MC search

TD Search

- Simulate episodes from the current (real) state s_t
- Estimate action-value function $Q(s, a)$
- For each step of simulation, update action-values by Sarsa

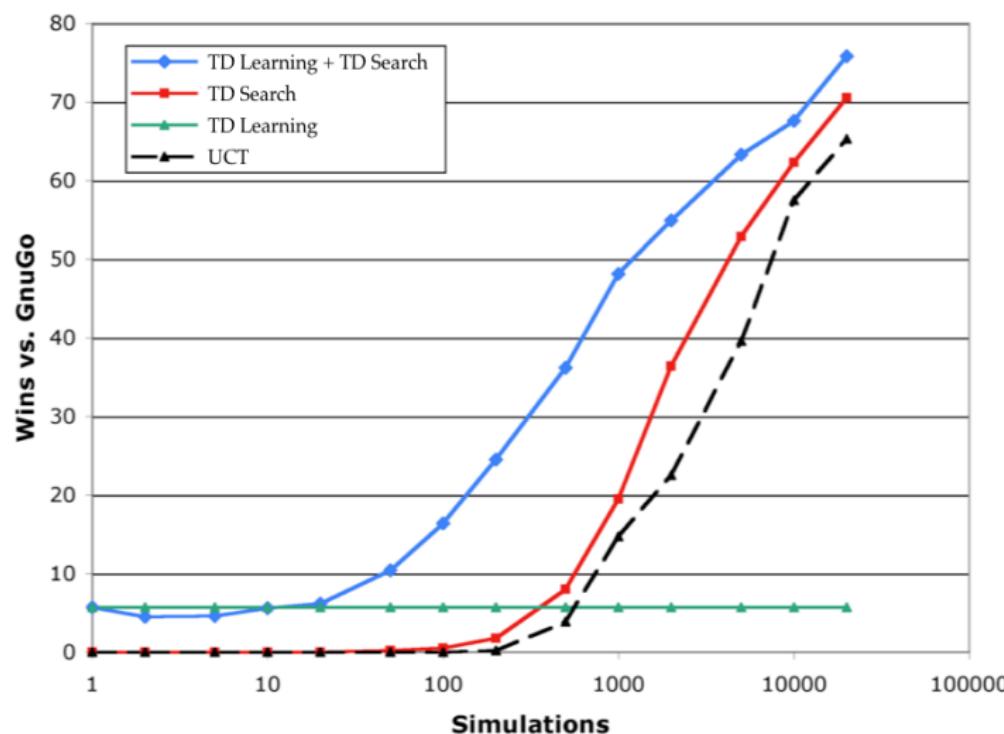
$$\Delta Q(S, A) = \alpha(R + \gamma Q(S', A') - Q(S, A))$$

- Select actions based on action-values $Q(s, a)$
 - e.g. ϵ -greedy
- May also use function approximation for Q

Dyna-2

- In Dyna-2, the agent stores two sets of feature weights
 - Long-term memory
 - Short-term (working) memory
- Long-term memory is updated from **real experience** using TD learning
 - General domain knowledge that applies to any episode
- Short-term memory is updated from **simulated experience** using TD search
 - Specific local knowledge about the current situation
- Overall value function is sum of long and short-term memories

Results of TD search in Go



Questions?

Lecture 9: Exploration and Exploitation

David Silver

Outline

1 Introduction

2 Multi-Armed Bandits

3 Contextual Bandits

4 MDPs

Exploration vs. Exploitation Dilemma

- Online decision-making involves a fundamental choice:
 - Exploitation** Make the best decision given current information
 - Exploration** Gather more information
- The best long-term strategy may involve short-term sacrifices
- Gather enough information to make the best overall decisions

Examples

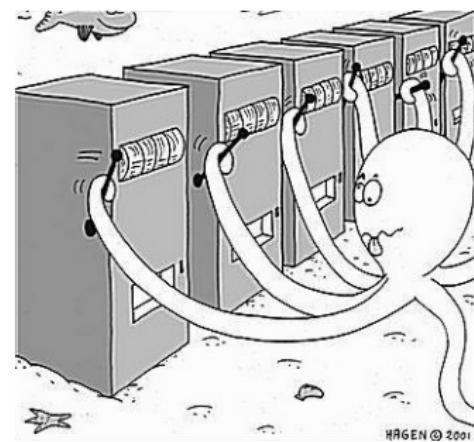
- Restaurant Selection
 - Exploitation Go to your favourite restaurant
 - Exploration Try a new restaurant
- Online Banner Advertisements
 - Exploitation Show the most successful advert
 - Exploration Show a different advert
- Oil Drilling
 - Exploitation Drill at the best known location
 - Exploration Drill at a new location
- Game Playing
 - Exploitation Play the move you believe is best
 - Exploration Play an experimental move

Principles

- Naive Exploration
 - Add noise to greedy policy (e.g. ϵ -greedy)
- Optimistic Initialisation
 - Assume the best until proven otherwise
- Optimism in the Face of Uncertainty
 - Prefer actions with uncertain values
- Probability Matching
 - Select actions according to probability they are best
- Information State Search
 - Lookahead search incorporating value of information

The Multi-Armed Bandit

- A multi-armed bandit is a tuple $\langle \mathcal{A}, \mathcal{R} \rangle$
- \mathcal{A} is a known set of m actions (or “arms”)
- $\mathcal{R}^a(r) = \mathbb{P}[r|a]$ is an unknown probability distribution over rewards
- At each step t the agent selects an action $a_t \in \mathcal{A}$
- The environment generates a reward $r_t \sim \mathcal{R}^{a_t}$
- The goal is to maximise cumulative reward $\sum_{\tau=1}^t r_\tau$



Regret

- The *action-value* is the mean reward for action a ,

$$Q(a) = \mathbb{E}[r|a]$$

- The *optimal value* V^* is

$$V^* = Q(a^*) = \max_{a \in \mathcal{A}} Q(a)$$

- The *regret* is the opportunity loss for one step

$$l_t = \mathbb{E}[V^* - Q(a_t)]$$

- The *total regret* is the total opportunity loss

$$L_t = \mathbb{E} \left[\sum_{\tau=1}^t V^* - Q(a_\tau) \right]$$

- Maximise cumulative reward \equiv minimise total regret

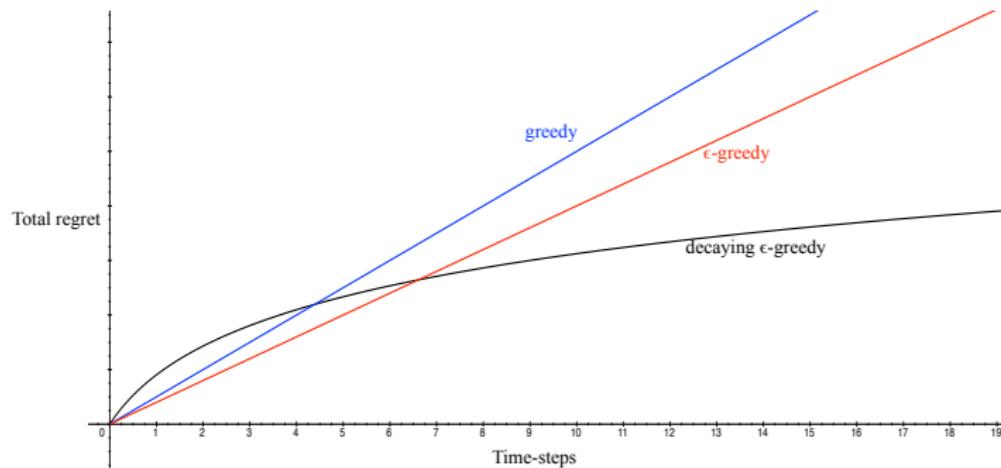
Counting Regret

- The *count* $N_t(a)$ is expected number of selections for action a
- The *gap* Δ_a is the difference in value between action a and optimal action a^* , $\Delta_a = V^* - Q(a)$
- Regret is a function of gaps and the counts

$$\begin{aligned} L_t &= \mathbb{E} \left[\sum_{\tau=1}^t V^* - Q(a_\tau) \right] \\ &= \sum_{a \in \mathcal{A}} \mathbb{E}[N_t(a)] (V^* - Q(a)) \\ &= \sum_{a \in \mathcal{A}} \mathbb{E}[N_t(a)] \Delta_a \end{aligned}$$

- A good algorithm ensures small counts for large gaps
- Problem: gaps are not known!

Linear or Sublinear Regret



- If an algorithm **forever** explores it will have linear total regret
- If an algorithm **never** explores it will have linear total regret
- Is it possible to achieve sublinear total regret?

Greedy Algorithm

- We consider algorithms that estimate $\hat{Q}_t(a) \approx Q(a)$
- Estimate the value of each action by Monte-Carlo evaluation

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{t=1}^T r_t \mathbf{1}(a_t = a)$$

- The *greedy* algorithm selects action with highest value

$$a_t^* = \operatorname{argmax}_{a \in \mathcal{A}} \hat{Q}_t(a)$$

- Greedy can lock onto a suboptimal action forever
- \Rightarrow Greedy has linear total regret

ϵ -Greedy Algorithm

- The ϵ -greedy algorithm continues to explore forever
 - With probability $1 - \epsilon$ select $a = \operatorname{argmax}_{a \in \mathcal{A}} \hat{Q}(a)$
 - With probability ϵ select a random action
- Constant ϵ ensures minimum regret

$$I_t \geq \frac{\epsilon}{\mathcal{A}} \sum_{a \in \mathcal{A}} \Delta_a$$

- $\Rightarrow \epsilon$ -greedy has linear total regret

Optimistic Initialisation

- Simple and practical idea: initialise $Q(a)$ to high value
- Update action value by incremental Monte-Carlo evaluation
- Starting with $N(a) > 0$

$$\hat{Q}_t(a_t) = \hat{Q}_{t-1} + \frac{1}{N_t(a_t)}(r_t - \hat{Q}_{t-1})$$

- Encourages systematic exploration early on
- But can still lock onto suboptimal action
- \Rightarrow greedy + optimistic initialisation has linear total regret
- \Rightarrow ϵ -greedy + optimistic initialisation has linear total regret

Decaying ϵ_t -Greedy Algorithm

- Pick a decay schedule for $\epsilon_1, \epsilon_2, \dots$
- Consider the following schedule

$$c > 0$$

$$d = \min_{a|\Delta_a > 0} \Delta_i$$

$$\epsilon_t = \min \left\{ 1, \frac{c|\mathcal{A}|}{d^2 t} \right\}$$

- Decaying ϵ_t -greedy has *logarithmic* asymptotic total regret!
- Unfortunately, schedule requires advance knowledge of gaps
- Goal: find an algorithm with sublinear regret for any multi-armed bandit (without knowledge of \mathcal{R})

Lower Bound

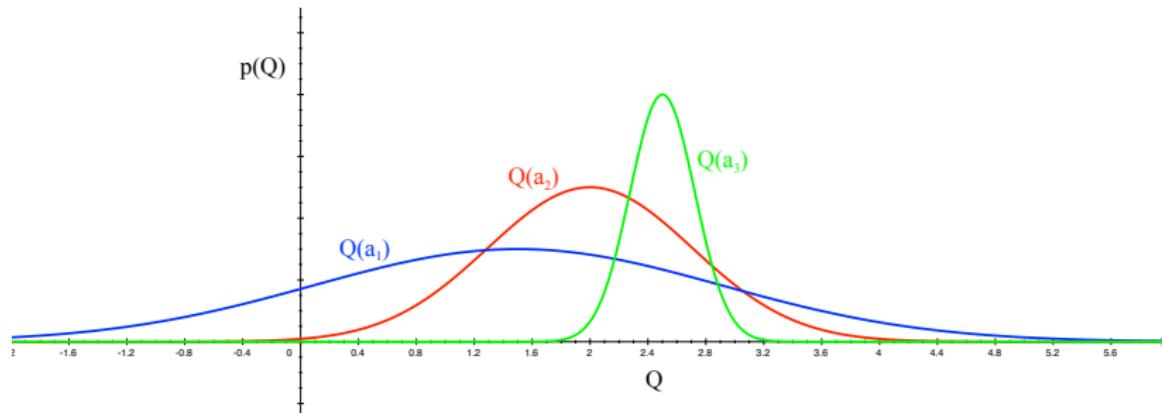
- The performance of any algorithm is determined by similarity between optimal arm and other arms
- Hard problems have similar-looking arms with different means
- This is described formally by the gap Δ_a and the similarity in distributions $KL(\mathcal{R}^a || \mathcal{R}^{a*})$

Theorem (Lai and Robbins)

Asymptotic total regret is at least logarithmic in number of steps

$$\lim_{t \rightarrow \infty} L_t \geq \log t \sum_{a | \Delta_a > 0} \frac{\Delta_a}{KL(\mathcal{R}^a || \mathcal{R}^{a*})}$$

Optimism in the Face of Uncertainty



- Which action should we pick?
- The more uncertain we are about an action-value
- The more important it is to explore that action
- It could turn out to be the best action

Optimism in the Face of Uncertainty (2)



- After picking **blue** action
- We are less uncertain about the value
- And more likely to pick another action
- Until we home in on best action

Upper Confidence Bounds

- Estimate an upper confidence $\hat{U}_t(a)$ for each action value
- Such that $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$ with high probability
- This depends on the number of times $N(a)$ has been selected
 - Small $N_t(a) \Rightarrow$ large $\hat{U}_t(a)$ (estimated value is uncertain)
 - Large $N_t(a) \Rightarrow$ small $\hat{U}_t(a)$ (estimated value is accurate)
- Select action maximising Upper Confidence Bound (UCB)

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} \hat{Q}_t(a) + \hat{U}_t(a)$$

Hoeffding's Inequality

Theorem (Hoeffding's Inequality)

Let X_1, \dots, X_t be i.i.d. random variables in $[0,1]$, and let $\bar{X}_t = \frac{1}{t} \sum_{\tau=1}^t X_\tau$ be the sample mean. Then

$$\mathbb{P} [\mathbb{E}[X] > \bar{X}_t + u] \leq e^{-2tu^2}$$

- We will apply Hoeffding's Inequality to rewards of the bandit
- conditioned on selecting action a

$$\mathbb{P} [Q(a) > \hat{Q}_t(a) + U_t(a)] \leq e^{-2N_t(a)U_t(a)^2}$$

Calculating Upper Confidence Bounds

- Pick a probability p that true value exceeds UCB
- Now solve for $U_t(a)$

$$e^{-2N_t(a)U_t(a)^2} = p$$

$$U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

- Reduce p as we observe more rewards, e.g. $p = t^{-4}$
- Ensures we select optimal action as $t \rightarrow \infty$

$$U_t(a) = \sqrt{\frac{2 \log t}{N_t(a)}}$$

UCB1

- This leads to the UCB1 algorithm

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(a) + \sqrt{\frac{2 \log t}{N_t(a)}}$$

Theorem

The UCB algorithm achieves logarithmic asymptotic total regret

$$\lim_{t \rightarrow \infty} L_t \leq 8 \log t \sum_{a | \Delta_a > 0} \Delta_a$$

└ Multi-Armed Bandits

└ Upper Confidence Bound

Example: UCB vs. ϵ -Greedy On 10-armed Bandit

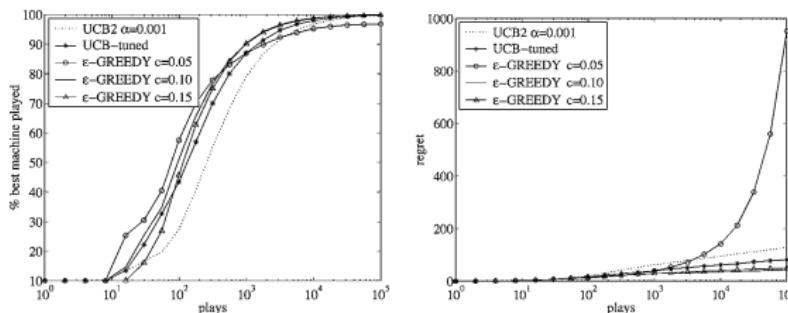


Figure 9. Comparison on distribution 11 (10 machines with parameters 0.9, 0.6, ..., 0.6).

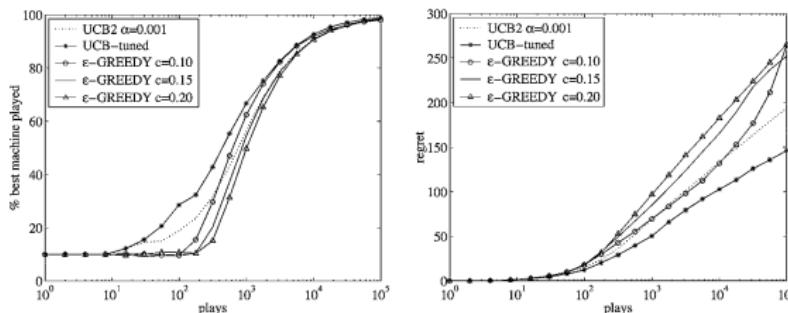


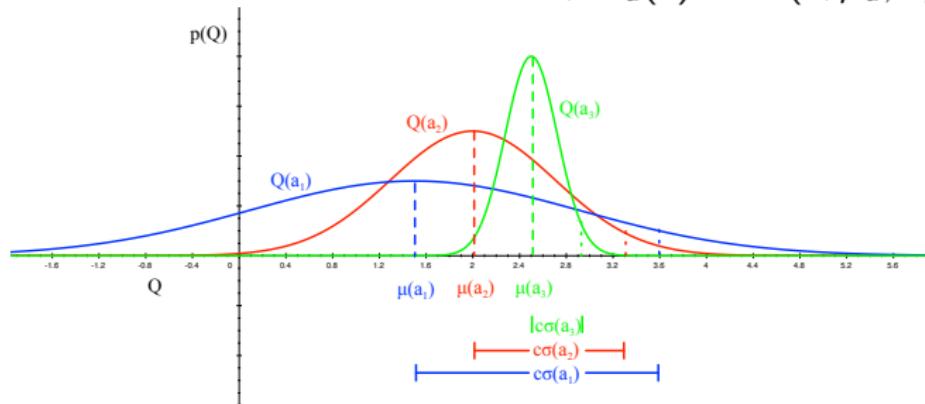
Figure 10. Comparison on distribution 12 (10 machines with parameters 0.9, 0.8, 0.8, 0.8, 0.7, 0.7, 0.7, 0.6, 0.6, 0.6).

Bayesian Bandits

- So far we have made no assumptions about the reward distribution \mathcal{R}
 - Except bounds on rewards
- **Bayesian bandits** exploit prior knowledge of rewards, $p[\mathcal{R}]$
- They compute posterior distribution of rewards $p[\mathcal{R} \mid h_t]$
 - where $h_t = a_1, r_1, \dots, a_{t-1}, r_{t-1}$ is the history
- Use posterior to guide exploration
 - Upper confidence bounds (Bayesian UCB)
 - Probability matching (Thompson sampling)
- Better performance if prior knowledge is accurate

Bayesian UCB Example: Independent Gaussians

- Assume reward distribution is Gaussian, $\mathcal{R}_a(r) = \mathcal{N}(r; \mu_a, \sigma_a^2)$



- Compute Gaussian posterior over μ_a and σ_a^2 (by Bayes law)

$$p[\mu_a, \sigma_a^2 | h_t] \propto p[\mu_a, \sigma_a^2] \prod_{t \mid a_t=a} \mathcal{N}(r_t; \mu_a, \sigma_a^2)$$

- Pick action that maximises standard deviation of $Q(a)$

$$a_t = \operatorname{argmax}_a \mu_a + c\sigma_a/\sqrt{N(a)}$$

Probability Matching

- Probability matching selects action a according to probability that a is the optimal action

$$\pi(a \mid h_t) = \mathbb{P} [Q(a) > Q(a'), \forall a' \neq a \mid h_t]$$

- Probability matching is optimistic in the face of uncertainty
 - Uncertain actions have higher probability of being max
- Can be difficult to compute analytically from posterior

Thompson Sampling

- Thompson sampling implements probability matching

$$\begin{aligned}\pi(a \mid h_t) &= \mathbb{P} [Q(a) > Q(a'), \forall a' \neq a \mid h_t] \\ &= \mathbb{E}_{\mathcal{R} \mid h_t} \left[\mathbf{1}(a = \operatorname{argmax}_{a \in \mathcal{A}} Q(a)) \right]\end{aligned}$$

- Use Bayes law to compute posterior distribution $p[\mathcal{R} \mid h_t]$
- Sample a reward distribution \mathcal{R} from posterior
- Compute action-value function $Q(a) = \mathbb{E} [\mathcal{R}_a]$
- Select action maximising value on sample, $a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(a)$
- Thompson sampling achieves Lai and Robbins lower bound!

Value of Information

- Exploration is useful because it gains information
- Can we quantify the value of information?
 - How much reward a decision-maker would be prepared to pay in order to have that information, prior to making a decision
 - Long-term reward after getting information - immediate reward
- Information gain is higher in uncertain situations
- Therefore it makes sense to explore uncertain situations more
- If we know value of information, we can trade-off exploration and exploitation *optimally*

Information State Space

- We have viewed bandits as *one-step* decision-making problems
- Can also view as *sequential* decision-making problems
- At each step there is an *information state* \tilde{s}
 - \tilde{s} is a statistic of the history, $\tilde{s}_t = f(h_t)$
 - summarising all information accumulated so far
- Each action a causes a transition to a new information state \tilde{s}' (by adding information), with probability $\tilde{P}_{\tilde{s}, \tilde{s}'}^a$
- This defines MDP $\tilde{\mathcal{M}}$ in augmented information state space

$$\tilde{\mathcal{M}} = \langle \tilde{\mathcal{S}}, \mathcal{A}, \tilde{\mathcal{P}}, \mathcal{R}, \gamma \rangle$$

Example: Bernoulli Bandits

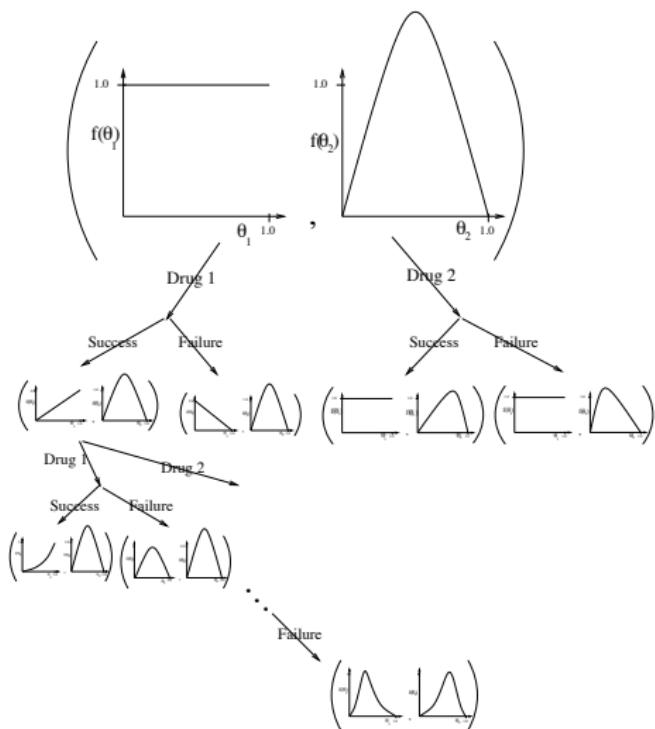
- Consider a Bernoulli bandit, such that $\mathcal{R}^a = \mathcal{B}(\mu_a)$
- e.g. Win or lose a game with probability μ_a
- Want to find which arm has the highest μ_a
- The information state is $\tilde{s} = \langle \alpha, \beta \rangle$
 - α_a counts the pulls of arm a where reward was 0
 - β_a counts the pulls of arm a where reward was 1

Solving Information State Space Bandits

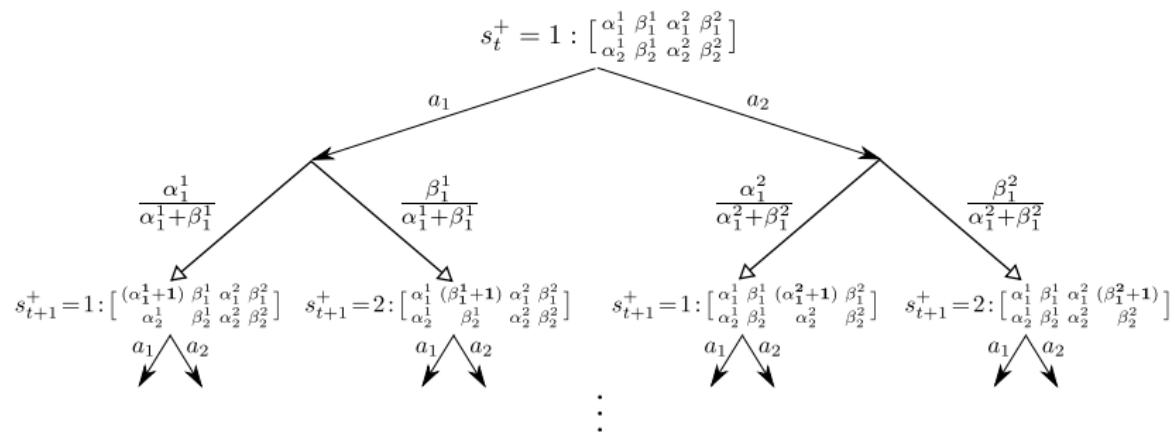
- We now have an infinite MDP over information states
- This MDP can be solved by reinforcement learning
- Model-free reinforcement learning
 - e.g. Q-learning (Duff, 1994)
- Bayesian model-based reinforcement learning
 - e.g. Gittins indices (Gittins, 1979)
 - This approach is known as *Bayes-adaptive RL*
 - Finds Bayes-optimal exploration/exploitation trade-off with respect to prior distribution

Bayes-Adaptive Bernoulli Bandits

- Start with $Beta(\alpha_a, \beta_a)$ prior over reward function \mathcal{R}^a
- Each time a is selected, update posterior for \mathcal{R}^a
 - $Beta(\alpha_a + 1, \beta_a)$ if $r = 0$
 - $Beta(\alpha_a, \beta_a + 1)$ if $r = 1$
- This defines transition function $\tilde{\mathcal{P}}$ for the Bayes-adaptive MDP
- Information state $\langle \alpha, \beta \rangle$ corresponds to reward model $Beta(\alpha, \beta)$
- Each state transition corresponds to a Bayesian model update



Bayes-Adaptive MDP for Bernoulli Bandits



Gittins Indices for Bernoulli Bandits

- Bayes-adaptive MDP can be solved by dynamic programming
- The solution is known as the *Gittins index*
- Exact solution to Bayes-adaptive MDP is typically intractable
 - Information state space is too large
- Recent idea: apply simulation-based search (Guez et al. 2012)
 - Forward search in information state space
 - Using simulations from current information state

Contextual Bandits

- A contextual bandit is a tuple $\langle \mathcal{A}, \mathcal{S}, \mathcal{R} \rangle$
- \mathcal{A} is a known set of actions (or “arms”)
- $\mathcal{S} = \mathbb{P}[s]$ is an unknown distribution over states (or “contexts”)
- $\mathcal{R}_s^a(r) = \mathbb{P}[r|s, a]$ is an unknown probability distribution over rewards
- At each step t
 - Environment generates state $s_t \sim \mathcal{S}$
 - Agent selects action $a_t \in \mathcal{A}$
 - Environment generates reward $r_t \sim \mathcal{R}_{s_t}^{a_t}$
- Goal is to maximise cumulative reward $\sum_{\tau=1}^t r_\tau$



Linear Regression

- Action-value function is expected reward for state s and action a

$$Q(s, a) = \mathbb{E}[r|s, a]$$

- Estimate value function with a linear function approximator

$$Q_\theta(s, a) = \phi(s, a)^\top \theta \approx Q(s, a)$$

- Estimate parameters by least squares regression

$$A_t = \sum_{\tau=1}^t \phi(s_\tau, a_\tau) \phi(s_\tau, a_\tau)^\top$$

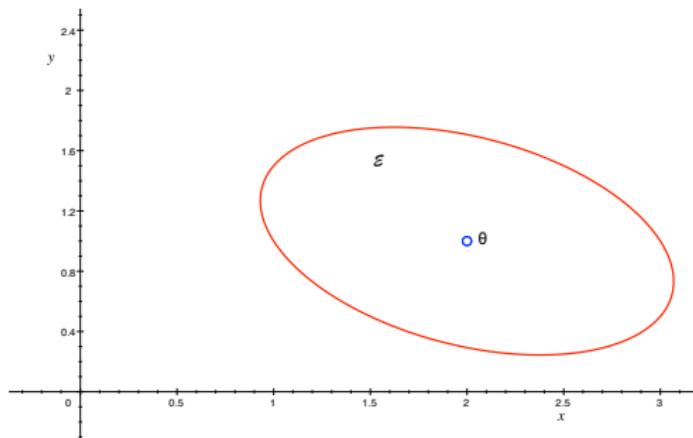
$$b_t = \sum_{\tau=1}^t \phi(s_\tau, a_\tau) r_\tau$$

$$\theta_t = A_t^{-1} b_t$$

Linear Upper Confidence Bounds

- Least squares regression estimates the mean action-value $Q_\theta(s, a)$
- But it can also estimate the variance of the action-value $\sigma_\theta^2(s, a)$
- i.e. the uncertainty due to parameter estimation error
- Add on a bonus for uncertainty, $U_\theta(s, a) = c\sigma$
- i.e. define UCB to be c standard deviations above the mean

Geometric Interpretation



- Define confidence ellipsoid \mathcal{E}_t around parameters θ_t
- Such that \mathcal{E}_t includes true parameters θ^* with high probability
- Use this ellipsoid to estimate the uncertainty of action values
- Pick parameters within ellipsoid that maximise action value

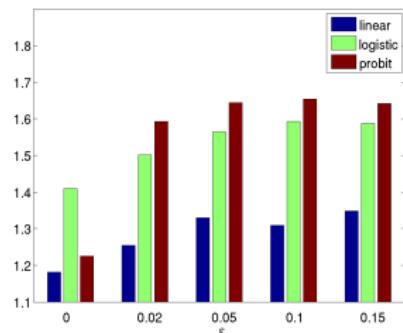
$$\operatorname{argmax}_{\theta \in \mathcal{E}} Q_\theta(s, a)$$

Calculating Linear Upper Confidence Bounds

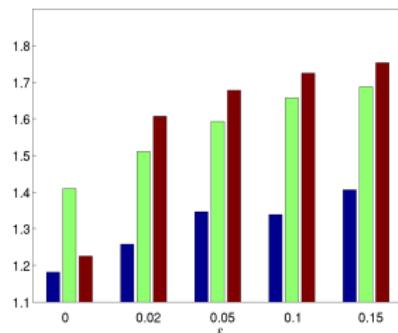
- For least squares regression, parameter covariance is A^{-1}
- Action-value is linear in features, $Q_\theta(s, a) = \phi(s, a)^\top \theta$
- So action-value variance is quadratic,
 $\sigma_\theta^2(s, a) = \phi(s, a)^\top A^{-1} \phi(s, a)$
- Upper confidence bound is $Q_\theta(s, a) + c \sqrt{\phi(s, a)^\top A^{-1} \phi(s, a)}$
- Select action maximising upper confidence bound

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q_\theta(s_t, a) + c \sqrt{\phi(s_t, a)^\top A_t^{-1} \phi(s_t, a)}$$

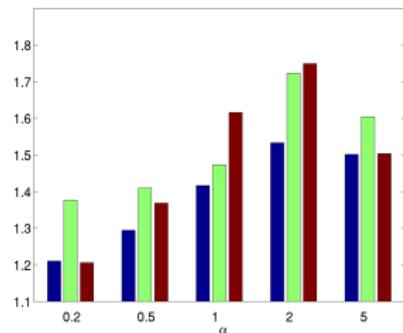
Example: Linear UCB for Selecting Front Page News



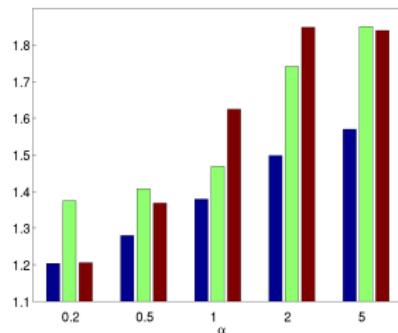
(a)



(b)



(c)



(d)

Exploration/Exploitation Principles to MDPs

The same principles for exploration/exploitation apply to MDPs

- Naive Exploration
- Optimistic Initialisation
- Optimism in the Face of Uncertainty
- Probability Matching
- Information State Search

Optimistic Initialisation: Model-Free RL

- Initialise action-value function $Q(s, a)$ to $\frac{r_{max}}{1-\gamma}$
- Run favourite model-free RL algorithm
 - Monte-Carlo control
 - Sarsa
 - Q-learning
 - ...
- Encourages systematic exploration of states and actions

Optimistic Initialisation: Model-Based RL

- Construct an **optimistic** model of the MDP
- Initialise transitions to **go to heaven**
 - (i.e. transition to terminal state with r_{max} reward)
- Solve optimistic MDP by favourite planning algorithm
 - policy iteration
 - value iteration
 - tree search
 - ...
- Encourages systematic exploration of states and actions
- e.g. RMax algorithm (Brafman and Tennenholz)

Upper Confidence Bounds: Model-Free RL

- Maximise UCB on action-value function $Q^\pi(s, a)$

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_t, a) + U(s_t, a)$$

- Estimate uncertainty in policy evaluation (easy)
- Ignores uncertainty from policy improvement
- Maximise UCB on optimal action-value function $Q^*(s, a)$

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_t, a) + U_1(s_t, a) + U_2(s_t, a)$$

- Estimate uncertainty in policy evaluation (easy)
- plus uncertainty from policy improvement (hard)

Bayesian Model-Based RL

- Maintain posterior distribution over MDP models
- Estimate both transitions and rewards, $p[\mathcal{P}, \mathcal{R} \mid h_t]$
 - where $h_t = s_1, a_1, r_2, \dots, s_t$ is the history
- Use posterior to guide exploration
 - Upper confidence bounds (Bayesian UCB)
 - Probability matching (Thompson sampling)

Thompson Sampling: Model-Based RL

- Thompson sampling implements probability matching

$$\begin{aligned}\pi(s, a \mid h_t) &= \mathbb{P} [Q^*(s, a) > Q^*(s, a'), \forall a' \neq a \mid h_t] \\ &= \mathbb{E}_{\mathcal{P}, \mathcal{R} \mid h_t} \left[\mathbf{1}(a = \operatorname{argmax}_{a \in \mathcal{A}} Q^*(s, a)) \right]\end{aligned}$$

- Use Bayes law to compute posterior distribution $p[\mathcal{P}, \mathcal{R} \mid h_t]$
- Sample** an MDP \mathcal{P}, \mathcal{R} from posterior
- Solve MDP using favourite planning algorithm to get $Q^*(s, a)$
- Select optimal action for sample MDP, $a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q^*(s_t, a)$

Information State Search in MDPs

- MDPs can be augmented to include information state
- Now the augmented state is $\langle s, \tilde{s} \rangle$
 - where s is original state within MDP
 - and \tilde{s} is a statistic of the history (accumulated information)
- Each action a causes a transition
 - to a new state s' with probability $\mathcal{P}_{s,s'}^a$
 - to a new information state \tilde{s}'
- Defines MDP $\tilde{\mathcal{M}}$ in augmented information state space

$$\tilde{\mathcal{M}} = \langle \tilde{\mathcal{S}}, \mathcal{A}, \tilde{\mathcal{P}}, \mathcal{R}, \gamma \rangle$$

Bayes Adaptive MDPs

- Posterior distribution over MDP model is an information state

$$\tilde{s}_t = \mathbb{P}[\mathcal{P}, \mathcal{R} | h_t]$$

- Augmented MDP over $\langle s, \tilde{s} \rangle$ is called **Bayes-adaptive MDP**
- Solve this MDP to find optimal exploration/exploitation trade-off (with respect to prior)
- However, Bayes-adaptive MDP is typically enormous
- Simulation-based search has proven effective (Guez et al.)

Conclusion

- Have covered several principles for exploration/exploitation
 - Naive methods such as ϵ -greedy
 - Optimistic initialisation
 - Upper confidence bounds
 - Probability matching
 - Information state search
- Each principle was developed in bandit setting
- But same principles also apply to MDP setting

Lecture 10: Classic Games

David Silver

Outline

- 1 State of the Art
- 2 Game Theory
- 3 Minimax Search
- 4 Self-Play Reinforcement Learning
- 5 Combining Reinforcement Learning and Minimax Search
- 6 Reinforcement Learning in Imperfect-Information Games
- 7 Conclusions

Why Study Classic Games?

- Simple rules, deep concepts
- Studied for hundreds or thousands of years
- Meaningful IQ test
- *Drosophila* of artificial intelligence
- Microcosms encapsulating real world issues
- Games are fun!

AI in Games: State of the Art

| Program | Level of Play | Program to Achieve Level |
|--------------------|---------------|--|
| Checkers | Perfect | <i>Chinook</i> |
| Chess | Superhuman | <i>Deep Blue</i> |
| Othello | Superhuman | <i>Logistello</i> |
| Backgammon | Superhuman | <i>TD-Gammon</i> |
| Scrabble | Superhuman | <i>Maven</i> |
| Go | Grandmaster | <i>MoGo</i> ¹ , <i>Crazy Stone</i> ² , <i>Zen</i> ³ |
| Poker ⁴ | Superhuman | <i>Polaris</i> |

¹ 9×9 ² 9×9 and 19×19 ³ 19×19 ⁴Heads-up Limit Texas Hold'em

RL in Games: State of the Art

| Program | Level of Play | RL Program to Achieve Level |
|--------------------|----------------------|---|
| Checkers | Perfect | <i>Chinook</i> |
| Chess | International Master | <i>KnightCap / Meep</i> |
| Othello | Superhuman | <i>Logistello</i> |
| Backgammon | Superhuman | <i>TD-Gammon</i> |
| Scrabble | Superhuman | <i>Maven</i> |
| Go | Grandmaster | <i>MoGo¹, Crazy Stone², Zen³</i> |
| Poker ⁴ | Superhuman | <i>SmooCT</i> |

¹ 9×9 ² 9×9 and 19×19 ³ 19×19 ⁴Heads-up Limit Texas Hold'em

Optimality in Games

- What is the optimal policy π^i for i th player?
- If all other players fix their policies π^{-i}
- **Best response** $\pi_*^i(\pi^{-i})$ is optimal policy against those policies
- **Nash equilibrium** is a joint policy for all players

$$\pi^i = \pi_*^i(\pi^{-i})$$

- such that every player's policy is a best response
- i.e. no player would choose to deviate from Nash

Single-Agent and Self-Play Reinforcement Learning

- Best response is solution to single-agent RL problem
 - Other players become part of the environment
 - Game is reduced to an MDP
 - Best response is optimal policy for this MDP
- Nash equilibrium is fixed-point of self-play RL
 - Experience is generated by playing games between agents

$$a_1 \sim \pi^1, a_2 \sim \pi^2, \dots$$

- Each agent learns best response to other players
- One player's policy determines another player's environment
- All players are adapting to each other

Two-Player Zero-Sum Games

We will focus on a special class of games:

- A **two-player game** has two (alternating) players
 - We will name player 1 *white* and player 2 *black*
- A **zero sum game** has equal and opposite rewards for black and white

$$R^1 + R^2 = 0$$

We consider methods for finding Nash equilibria in these games

- Game tree search (i.e. planning)
- Self-play reinforcement learning

Perfect and Imperfect Information Games

- A **perfect information** or **Markov** game is fully observed
 - Chess
 - Checkers
 - Othello
 - Backgammon
 - Go
- An **imperfect information** game is partially observed
 - Scrabble
 - Poker
- We focus first on perfect information games

Minimax

- A **value function** defines the expected total reward given joint policies $\pi = \langle \pi^1, \pi^2 \rangle$

$$v_\pi(s) = \mathbb{E}_\pi [G_t \mid S_t = s]$$

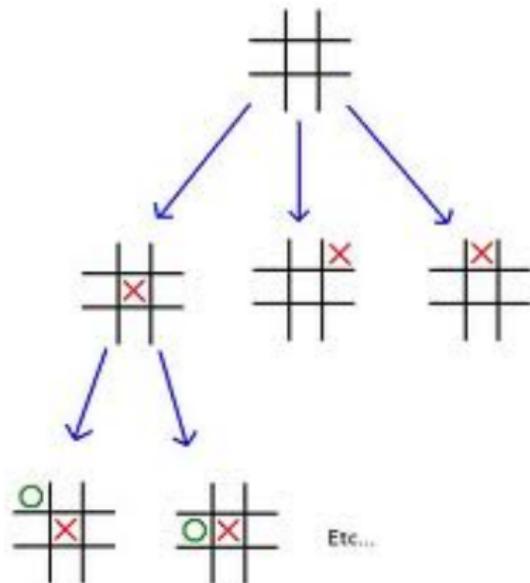
- A **minimax** value function maximizes white's expected return while minimizing black's expected return

$$v_*(s) = \max_{\pi^1} \min_{\pi^2} v_\pi(s)$$

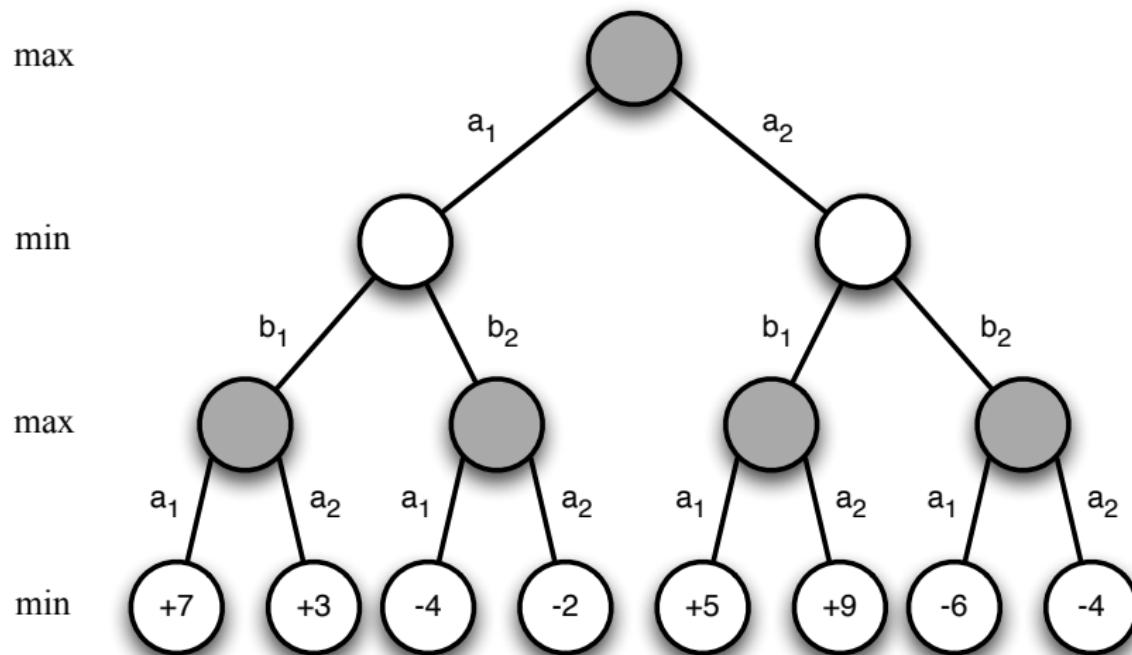
- A **minimax** policy is a joint policy $\pi = \langle \pi^1, \pi^2 \rangle$ that achieves the minimax values
- There is a unique minimax value function
- A minimax policy is a Nash equilibrium

Minimax Search

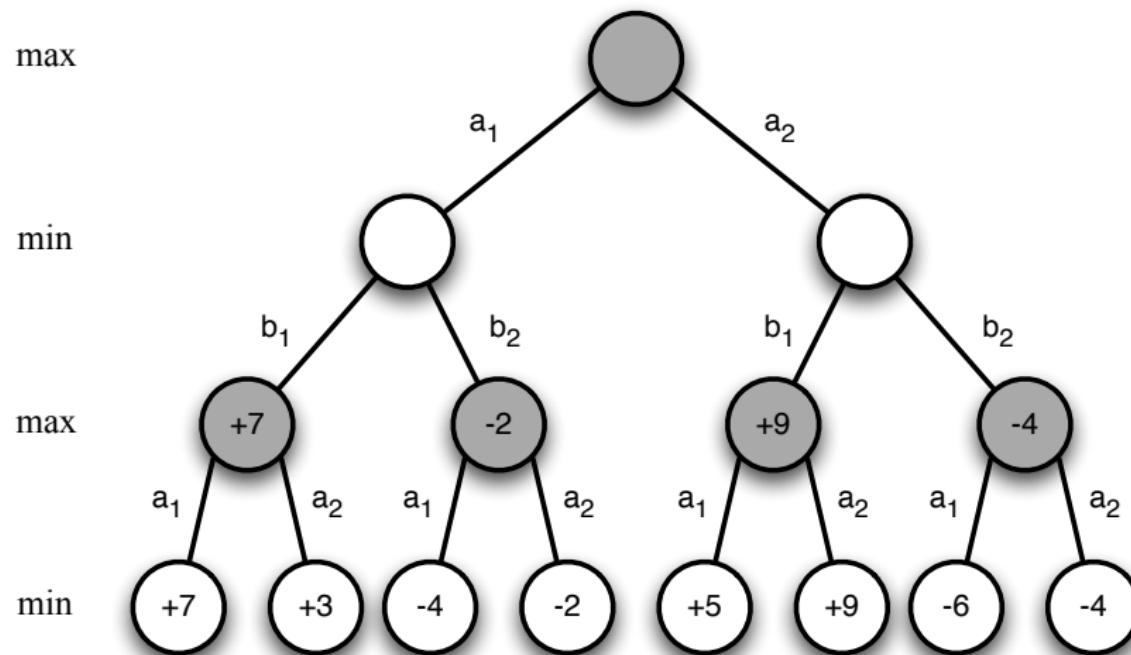
- Minimax values can be found by depth-first game-tree search
- Introduced by Claude Shannon: *Programming a Computer for Playing Chess*
- Ran on paper!



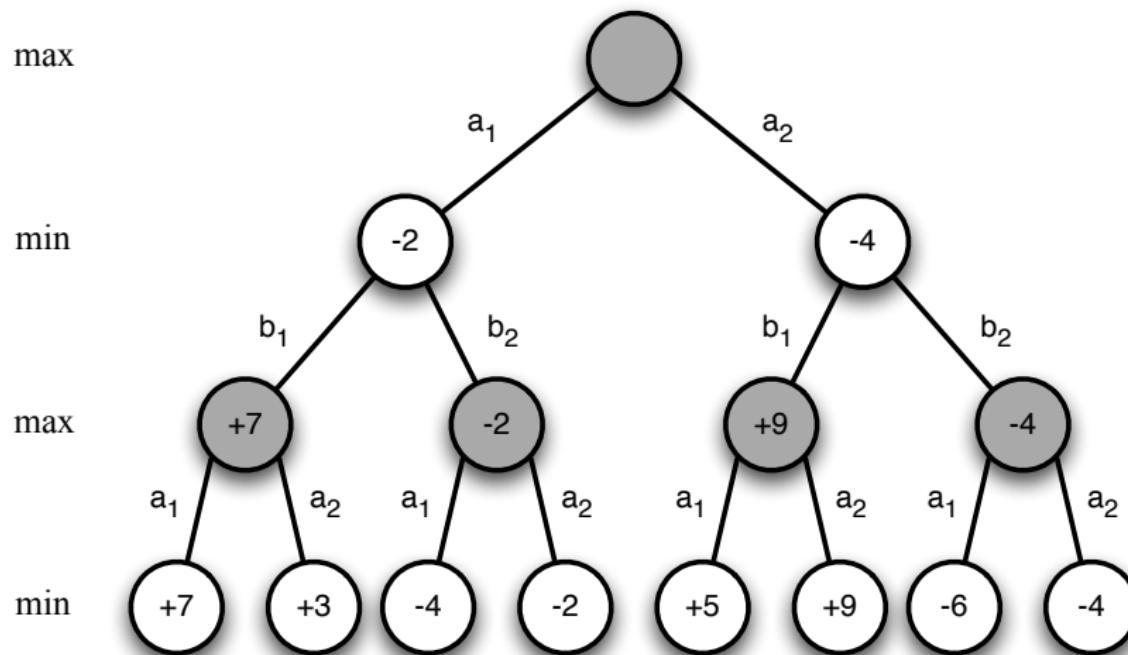
Minimax Search Example



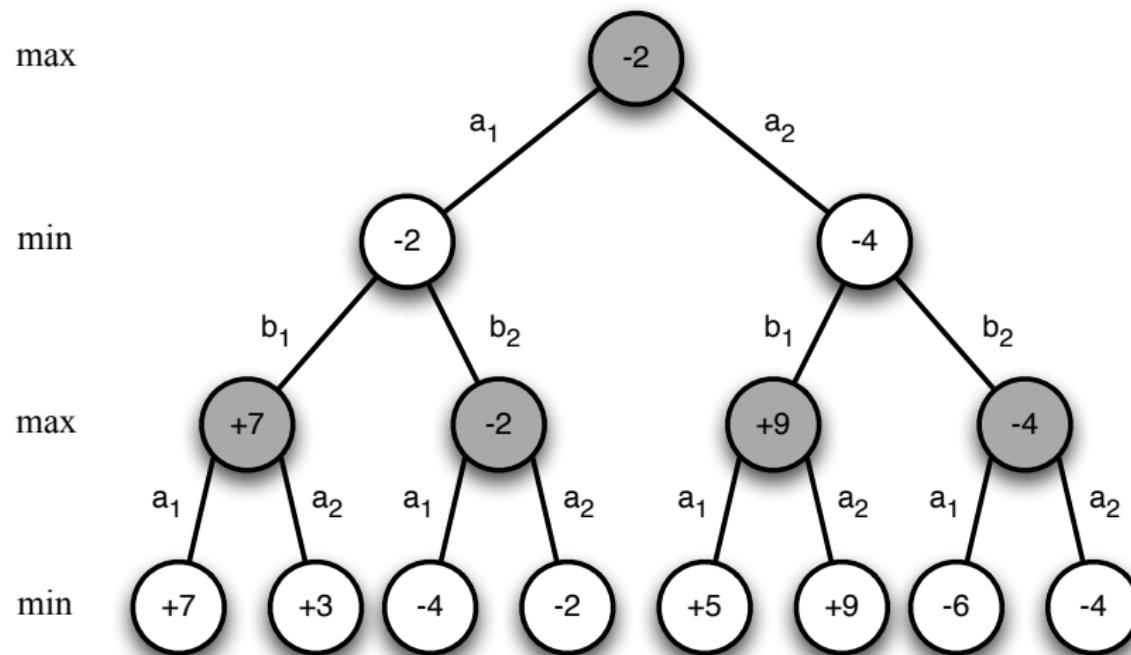
Minimax Search Example



Minimax Search Example



Minimax Search Example

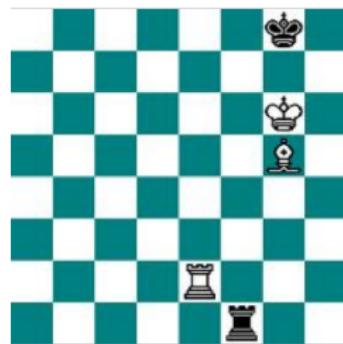


Value Function in Minimax Search

- Search tree grows exponentially
- Impractical to search to the end of the game
- Instead use value function approximator $v(s, w) \approx v_*(s)$
 - aka *evaluation function, heuristic function*
- Use value function to estimate minimax value at leaf nodes
- Minimax search run to fixed depth with respect to leaf values

Binary-Linear Value Function

- Binary feature vector $\mathbf{x}(s)$: e.g. one feature per piece
- Weight vector \mathbf{w} : e.g. value of each piece
- Position is evaluated by summing weights of active features



$$v(s, \mathbf{w}) = \mathbf{x}(s) \cdot \mathbf{w} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix} \cdot \begin{bmatrix} +5 \\ +3 \\ +1 \\ -5 \\ -3 \\ -1 \\ \vdots \end{bmatrix} = \begin{array}{c} \text{King} \\ \text{Queen} \\ \text{Bishop} \\ \text{Knight} \\ \text{Rook} \\ \text{Pawn} \end{array}$$

$$v(s, \mathbf{w}) = 5 + 3 - 5 = 3$$

Deep Blue

- Knowledge
 - 8000 handcrafted chess features
 - Binary-linear value function
 - Weights largely hand-tuned by human experts
- Search
 - High performance parallel alpha-beta search
 - 480 special-purpose VLSI chess processors
 - Searched 200 million positions/second
 - Looked ahead 16-40 ply
- Results
 - Defeated human champion Garry Kasparov 4-2 (1997)
 - Most watched event in internet history

Chinook

- Knowledge
 - Binary-linear value function
 - 21 knowledge-based features (position, mobility, ...)
 - x4 phases of the game
- Search
 - High performance alpha-beta search
 - Retrograde analysis
 - Search backward from won positions
 - Store all winning positions in lookup tables
 - Plays perfectly from last n checkers
- Results
 - Defeated Marion Tinsley in world championship 1994
 - won 2 games but Tinsley withdrew for health reasons
 - Chinook solved Checkers in 2007
 - perfect play against God

Self-Play Temporal-Difference Learning

- Apply value-based RL algorithms to games of self-play
- MC: update value function towards the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - v(S_t, \mathbf{w})) \nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

- TD(0): update value function towards successor value $v(S_{t+1})$

$$\Delta \mathbf{w} = \alpha(v(S_{t+1}, \mathbf{w}) - v(S_t, \mathbf{w})) \nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

- TD(λ): update value function towards the λ -return G_t^λ

$$\Delta \mathbf{w} = \alpha(G_t^\lambda - v(S_t, \mathbf{w})) \nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

Policy Improvement with Afterstates

- For deterministic games it is sufficient to estimate $v_*(s)$
- This is because we can efficiently evaluate the **afterstate**

$$q_*(s, a) = v_*(\text{succ}(s, a))$$

- Rules of the game define the successor state $\text{succ}(s, a)$
- Actions are selected e.g. by min/maximising afterstate value

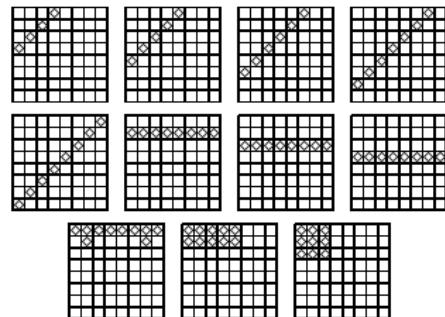
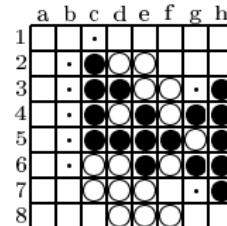
$$A_t = \operatorname{argmax}_a v_*(\text{succ}(S_t, a)) \quad \text{for white}$$

$$A_t = \operatorname{argmin}_a v_*(\text{succ}(S_t, a)) \quad \text{for black}$$

- This improves joint policy for both players

Self-Play TD in Othello: *Logistello*

- Logistello created its own features
- Start with raw input features, e.g.
“black stone at C1?”
- Construct new features by
conjunction/disjunction
- Created 1.5 million features in
different configurations
- Binary-linear value function using
these features



Reinforcement Learning in Logistello

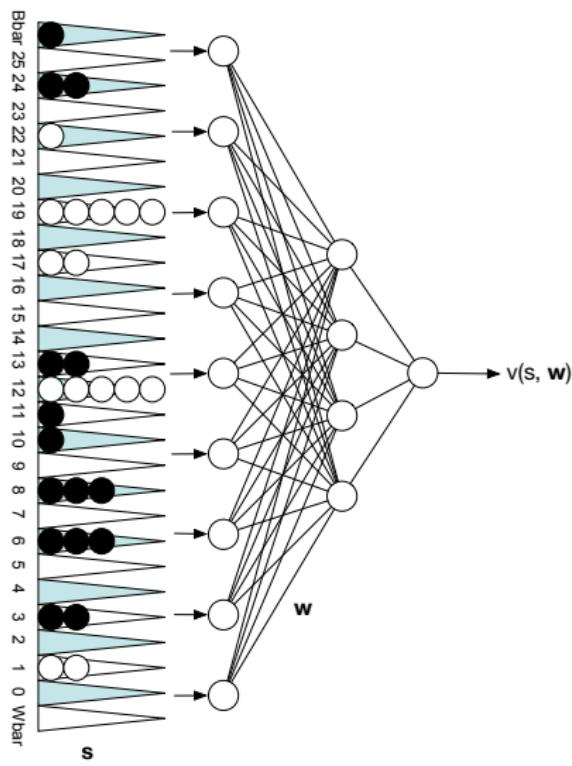
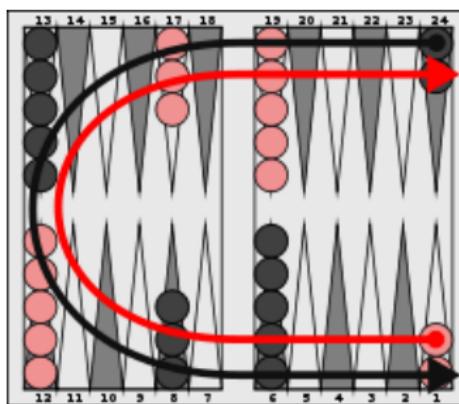
Logistello used generalised policy iteration

- Generate batch of self-play games from current policy
- Evaluate policies using Monte-Carlo (regress to outcomes)
- Greedy policy improvement to generate new players

Results

- Defeated World Champion Takeshi Murukami 6-0

TD Gammon: Non-Linear Value Function Approximation



Self-Play TD in Backgammon: *TD-Gammon*

- Initialised with random weights
- Trained by games of self-play
- Using non-linear temporal-difference learning

$$\delta_t = v(S_{t+1}, \mathbf{w}) - v(S_t, \mathbf{w})$$

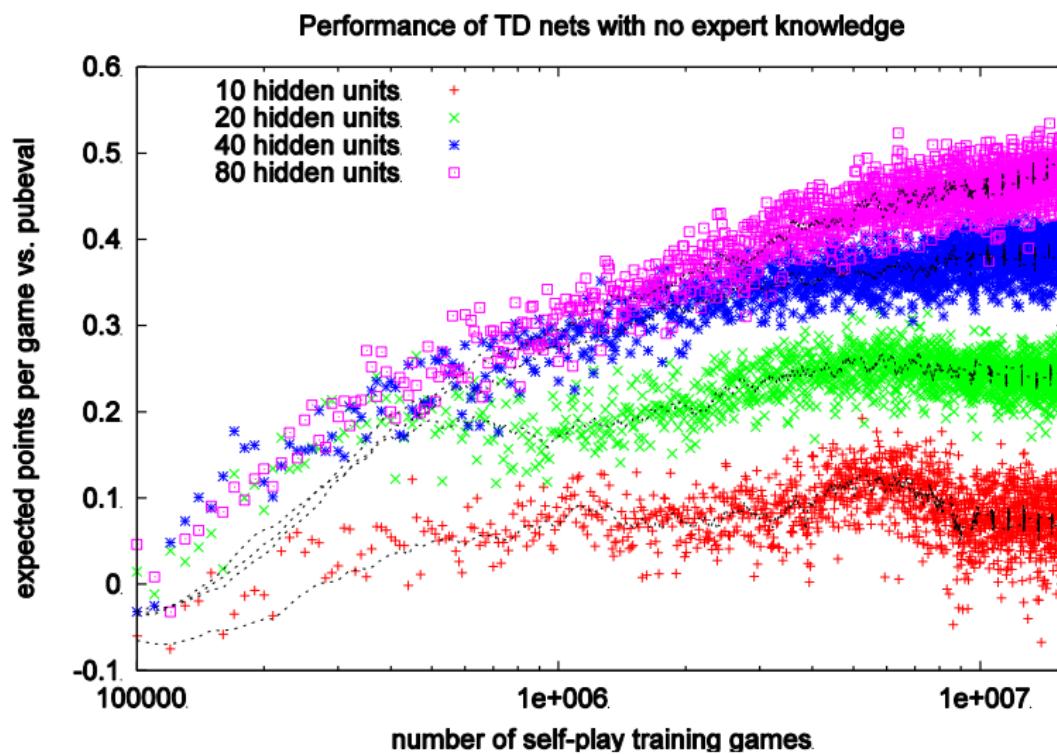
$$\Delta \mathbf{w} = \alpha \delta_t \nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

- Greedy policy improvement (no exploration)
- Algorithm always converged in practice
- Not true for other games

TD Gammon: Results

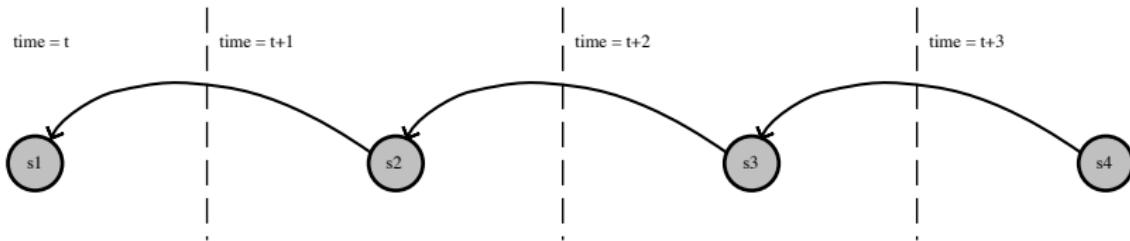
- Zero expert knowledge \implies strong intermediate play
- Hand-crafted features \implies advanced level of play (1991)
- 2-ply search \implies strong master play (1993)
- 3-ply search \implies superhuman play (1998)
- Defeated world champion Luigi Villa 7-1 (1992)

New TD-Gammon Results



Simple TD

- TD: update value towards successor value



- Value function approximator $v(s, \mathbf{w})$ with parameters \mathbf{w}
- Value function backed up from raw value at next state

$$v(S_t, \mathbf{w}) \leftarrow v(S_{t+1}, \mathbf{w})$$

- First learn value function by TD learning
- Then use value function in minimax search (no learning)

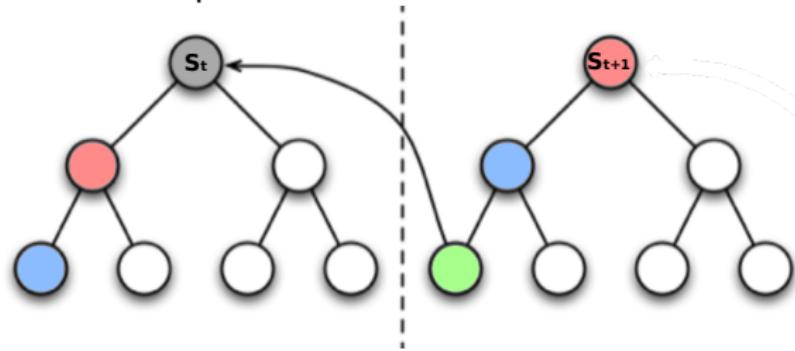
$$v_+(S_t, \mathbf{w}) = \underset{s \in \text{leaves}(S_t)}{\text{minimax}} v(s, \mathbf{w})$$

Simple TD: Results

- Othello: superhuman performance in *Logistello*
- Backgammon: superhuman performance in *TD-Gammon*
- Chess: poor performance
- Checkers: poor performance
- In chess tactics seem necessary to find signal in position
- e.g. hard to find checkmates without search
- Can we learn directly from minimax search values?

TD Root

- TD root: update value towards successor search value



- Search value is computed at root position S_t

$$v_+(S_t, \mathbf{w}) = \underset{s \in \text{leaves}(S_t)}{\text{minimax}} v(s, \mathbf{w})$$

- Value function backed up from *search value* at next state

$$v(S_t, \mathbf{w}) \leftarrow v_+(S_{t+1}, \mathbf{w}) = v(l_+(S_{t+1}), \mathbf{w})$$

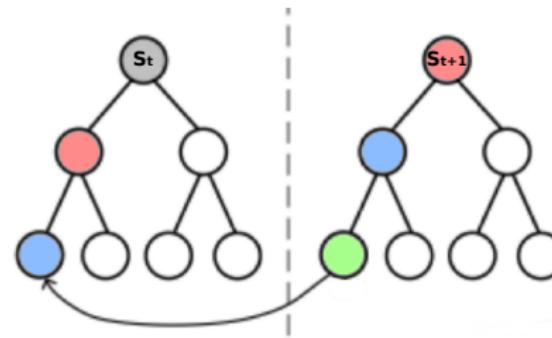
- Where $l_+(s)$ is the leaf node achieving minimax value from s

TD Root in Checkers: *Samuel's Player*

- First ever TD learning algorithm (*Samuel 1959*)
- Applied to a Checkers program that learned by self-play
- Defeated an amateur human player
- Also used other ideas we might now consider strange

TD Leaf

- TD leaf: update search value towards successor search value



- Search value computed at current and next step

$$v_+(S_t, \mathbf{w}) = \underset{s \in \text{leaves}(S_t)}{\text{minimax}} v(s, \mathbf{w}), \quad v_+(S_{t+1}, \mathbf{w}) = \underset{s \in \text{leaves}(S_{t+1})}{\text{minimax}} v(s, \mathbf{w})$$

- Search value at step t backed up from *search value* at $t + 1$

$$\begin{aligned} v_+(S_t, \mathbf{w}) &\leftarrow v_+(S_{t+1}, \mathbf{w}) \\ \implies v(I_+(S_t), \mathbf{w}) &\leftarrow v(I_+(S_{t+1}), \mathbf{w}) \end{aligned}$$

TD leaf in Chess: *Knightcap*

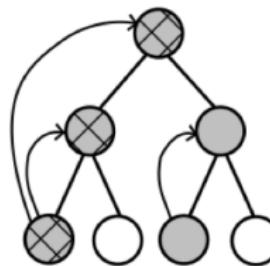
- Learning
 - *Knightcap* trained against expert opponent
 - Starting from standard piece values only
 - Learnt weights using TD leaf
- Search
 - Alpha-beta search with standard enhancements
- Results
 - Achieved master level play after a small number of games
 - Was not effective in self-play
 - Was not effective without starting from good weights

TD leaf in Checkers: *Chinook*

- Original Chinook used hand-tuned weights
- Later version was trained by self-play
- Using TD leaf to adjust weights
 - Except material weights which were kept fixed
- Self-play weights performed \geq hand-tuned weights
- i.e. learning to play at superhuman level

Treestrap

- Treestrap: update search values towards deeper search values



- Minimax search value computed at *all* nodes $s \in \text{nodes}(S_t)$
- Value backed up from search value, at same step, for all nodes

$$\begin{aligned} v(s, \mathbf{w}) &\leftarrow v_+(s, \mathbf{w}) \\ \implies v(s, \mathbf{w}) &\leftarrow v(l_+(s), \mathbf{w}) \end{aligned}$$

Treestrap in Chess: *Meep*

- Binary linear value function with 2000 features
- Starting from random initial weights (no prior knowledge)
- Weights adjusted by TreeStrap
- Won 13/15 vs. international masters
- Effective in self-play
- Effective from random initial weights

Simulation-Based Search

- Self-play reinforcement learning can replace search
- Simulate games of self-play from root state S_t
- Apply RL to simulated experience
 - Monte-Carlo Control \implies Monte-Carlo Tree Search
 - Most effective variant is UCT algorithm
 - Balance exploration/exploitation in each node using UCB
 - Self-play UCT converges on minimax values
 - Perfect information, zero-sum, 2-player games
 - Imperfect information: see next section

Performance of MCTS in Games

- MCTS is best performing method in many challenging games
 - Go (last lecture)
 - Hex
 - Lines of Action
 - Amazons
- In many games simple Monte-Carlo search is enough
 - Scrabble
 - Backgammon

Simple Monte-Carlo Search in Maven

■ Learning

- Maven evaluates moves by $score + v(rack)$
- Binary-linear value function of rack
- Using one, two and three letter features
- Q???????, QU??????, III????
- Learnt by Monte-Carlo policy iteration (cf. Logistello)

■ Search

- Roll-out moves by imagining n steps of self-play
- Evaluate resulting position by $score + v(rack)$
- Score move by average evaluation in rollouts
- Select and play highest scoring move
- Specialised endgame search using B^*

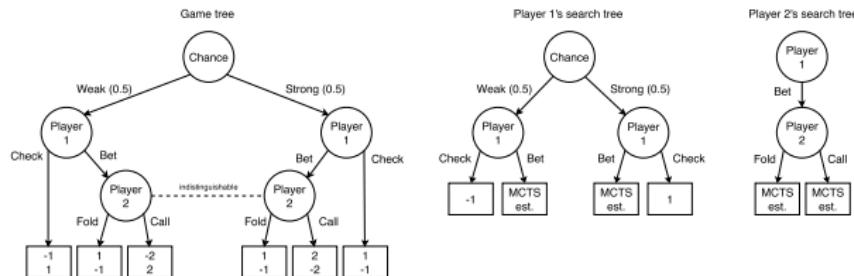
Maven: Results

- Maven beat world champion Adam Logan 9-5
- Here Maven predicted endgame to finish with MOUTHPART
- Analysis showed Maven had error rate of 3 points per game

| | | | | | | | | | | | | | |
|----------------|----------------|----------------|----------------|-----------------|----------------|----------------|----------------|----------------|-----------------|----------------|----------------|----------------|----------------|
| M ₃ | O ₁ | U ₁ | T ₁ | H ₄ | | R ₁ | R ₁ | T ₁ | | 2L | | 3W | |
| R ₁ | E ₁ | | | | 3L | | | | Q ₁₀ | | | 2W | |
| T ₁ | | 2W | | | | 2L | | 2L | U ₁ | | G ₂ | | |
| H ₄ | U ₁ | R ₁ | T ₁ | | | 2L | | | R ₁ | 2W | R ₁ | 2L | |
| H ₁ | E ₁ | O ₁ | H ₁ | | | | | | I ₁ | S ₁ | E ₁ | L ₁ | |
| | 3L | D ₂ | O ₁ | Z ₁₀ | Y ₄ | | | 3L | P ₃ | R ₁ | X ₈ | E ₁ | |
| | E ₁ | | | E ₁ | | 2L | J ₈ | R ₁ | H ₄ | S ₁ | I ₁ | | |
| I ₁ | R ₁ | M ₃ | B ₃ | C ₃ | R ₁ | Y ₄ | Y ₄ | | N ₁ | 2L | E ₁ | 3W | |
| H ₄ | E ₁ | | | R ₁ | | 2L | | K ₅ | | 2L | | | |
| | 3L | H ₁ | F ₄ | 3L | L ₁ | | | B ₃ | | | 3L | | |
| | D ₂ | E ₁ | | O ₁ | | | O ₁ | R ₁ | | | | | |
| 2L | D ₂ | E ₁ | V ₄ | I ₁ | R ₁ | H ₁ | C ₃ | E ₁ | S ₁ | 2W | | 2L | |
| | D ₂ | | G ₂ | | G ₂ | O ₁ | 2L | | | 2W | | | |
| 2W | | | H ₁ | 3L | F ₄ | | 3L | | | | 2W | | |
| P ₃ | I ₁ | L ₁ | I ₁ | S ₁ | | T ₁ | U ₁ | T ₁ | O ₁ | R ₁ | I ₁ | A ₁ | L ₁ |

Game-Tree Search in Imperfect Information Games

- Players have different information states and therefore separate search trees



- There is one node for each information state
 - summarising what a player knows
 - e.g. the cards they have seen
- Many real states may share the same information state
- May also aggregate states e.g. with similar value

Solution Methods for Imperfect Information Games

Information-state game tree may be solved by:

- Iterative forward-search methods
 - e.g. Counterfactual regret minimization
 - “Perfect” play in Poker (heads-up limit Hold'em)
- Self-play reinforcement learning
- e.g. Smooth UCT
 - 3 silver medals in two- and three-player Poker (limit Hold'em)
 - Outperformed massive-scale forward-search agents

Smooth UCT Search

- Apply MCTS to information-state game tree
- Variant of UCT, inspired by game-theoretic Fictitious Play
 - Agents learn against and respond to opponents' average behaviour
- Extract average strategy from nodes' action counts,
$$\pi_{avg}(a|s) = \frac{N(s,a)}{N(s)}$$
.
- At each node, pick actions according to

$$A \sim \begin{cases} \text{UCT}(S), & \text{with probability } \eta \\ \pi_{avg}(\cdot|S), & \text{with probability } 1 - \eta \end{cases}$$

- Empirically, in variants of Poker:
 - Naive MCTS diverged
 - Smooth UCT converged to Nash equilibrium

RL in Games: A Successful Recipe

| Program | Input features | Value Fn | RL | Training | Search |
|--------------------------------|-------------------------------------|----------------|-----------------|--------------------|--------------------|
| Chess <i>Meep</i> | Binary <i>Pieces, pawns, ...</i> | Linear | TreeStrap | Self-Play / Expert | $\alpha\beta$ |
| Checkers <i>Chinook</i> | Binary <i>Pieces, ...</i> | Linear | TD leaf | Self-Play | $\alpha\beta$ |
| Othello <i>Logistello</i> | Binary <i>Disc configs</i> | Linear | MC | Self-Play | $\alpha\beta$ |
| Backgammon <i>TD Gammon</i> | Binary <i>Num checkers</i> | Neural network | TD(λ) | Self-Play | $\alpha\beta / MC$ |
| Go <i>MoGo</i> | Binary <i>Stone patterns</i> | Linear | TD | Self-Play | MCTS |
| Scrabble <i>Maven</i> | Binary <i>Letters on rack</i> | Linear | MC | Self-Play | MC search |
| Limit Hold'em <i>SmooCT</i> | Binary <i>Card abstraction</i> | Linear | MCTS | Self-Play | - |