Reinforcement Learning

NGUYEN DO VAN, PHD

Reinforcement Learning

- Introduction
- Markov Decision Process
- Dynamic Programming
- Online Learning
- Function Approximation
- Exploration and Exploitation

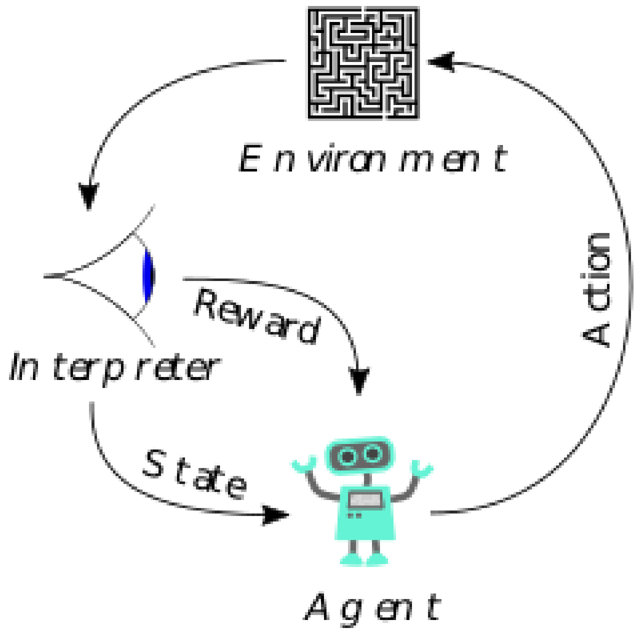
Introduction

Intelligent agents learning and acting

Sequence of decision, reward

Reinforcement learning: What is it?

- Making good decision to do new task: fundamental challenge in AI, ML
- Learn to make good sequence of decisions
- Intelligent agents learning and acting
  - Learning by trial-and-error, in real time
  - Improve with experience
  - Inspired by psychology:
    - Agents + environment
    - Agents select action to maximize *cumulative* rewards



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

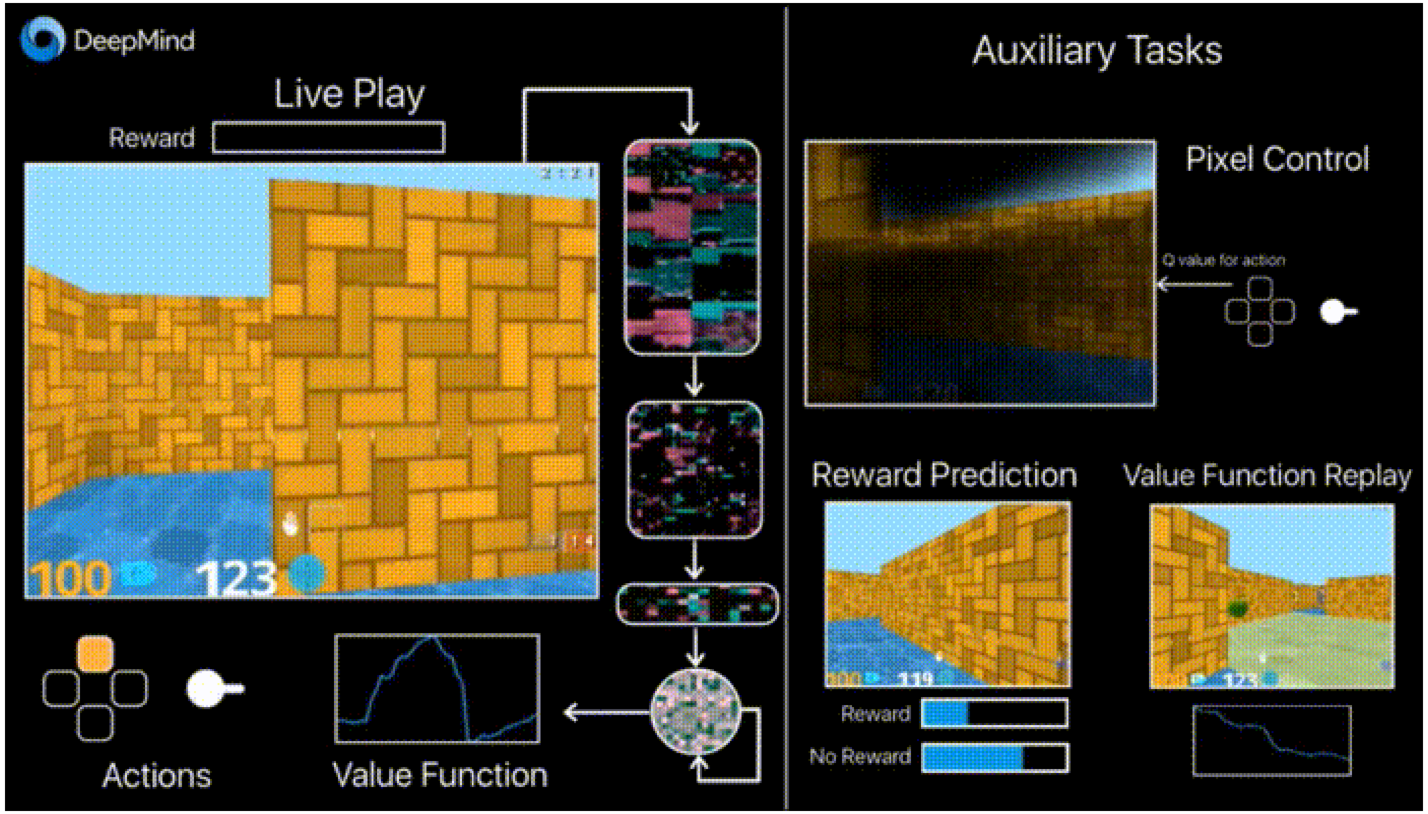
RL Applications

- Multi-disciplinary Conference on Reinforcement Learning and Decision Making (RLDM2017)
  - Robotics
  - Video games
  - Conversational systems
  - Medical intervention
  - Algorithm improvement
  - Improvisational theatre
  - Autonomous driving
  - Prosthetic arm control
  - Financial trading
  - Query completion

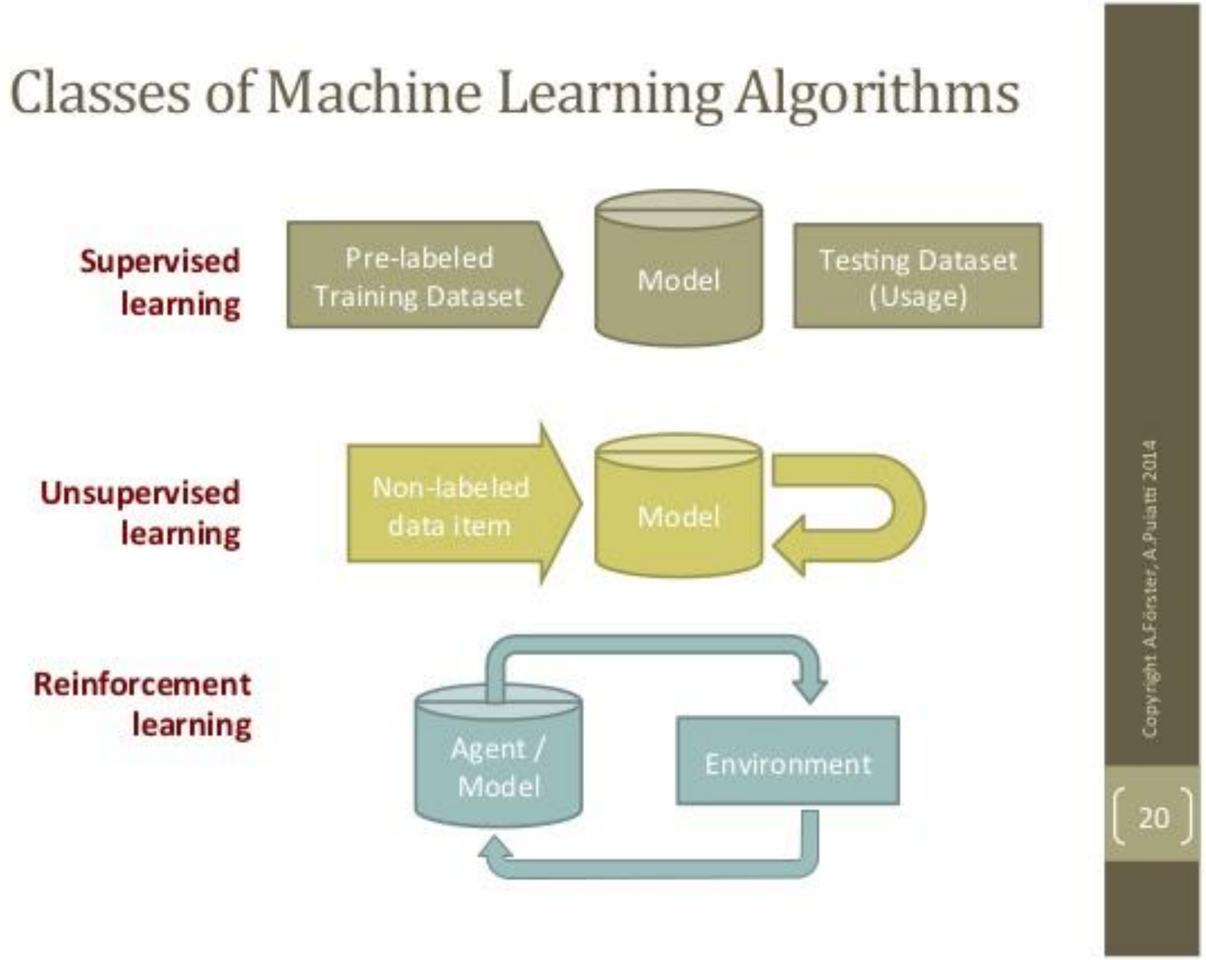
Robotics

https://www.youtube.com/watch?v=ZBFwe1gF0FU

Gaming


DeepMind
Live Play
Reward
Auxiliary Tasks
Pixel Control
Q value for action
Reward Prediction
Value Function Replay
Reward
No Reward
Actions
Value Function

RL vs supervised and unsupervised learning



Practical and technical challenges:

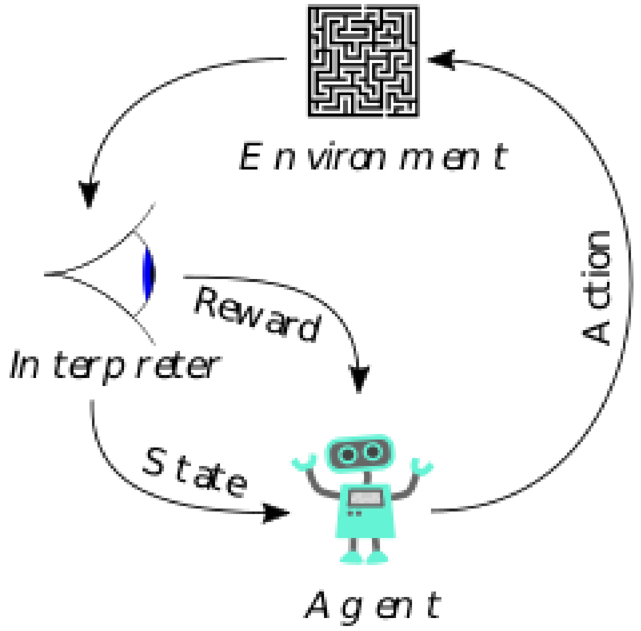
- Need to access to the environment
- Jointly learning AND planning from correlated sample
- Data distribution changes with action choice

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 maximize cumulative reward
- Example:
  - Robot Navigation: (-) Crash wall, (+) reaching target…
  - Control power station: (+) producing power, (-) exceeding safety thresholds
  - Games: (+) Wining game, Killing enemy, collecting bloods, (-) mine

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 environment step



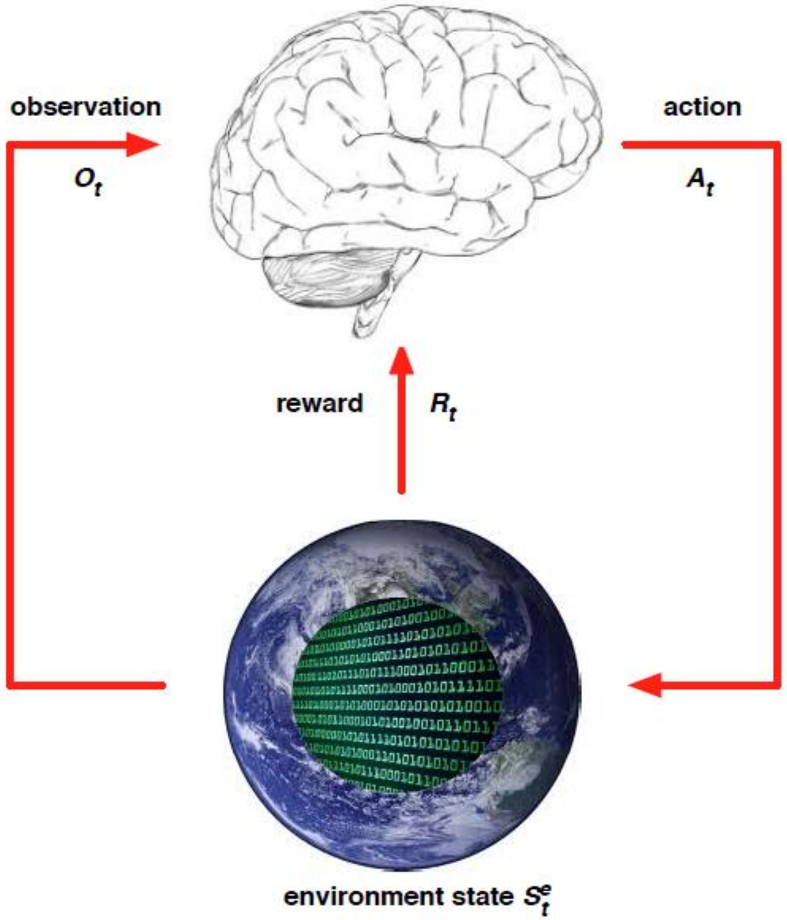
History and State

- History is the sequence of observations, actions and rewards

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

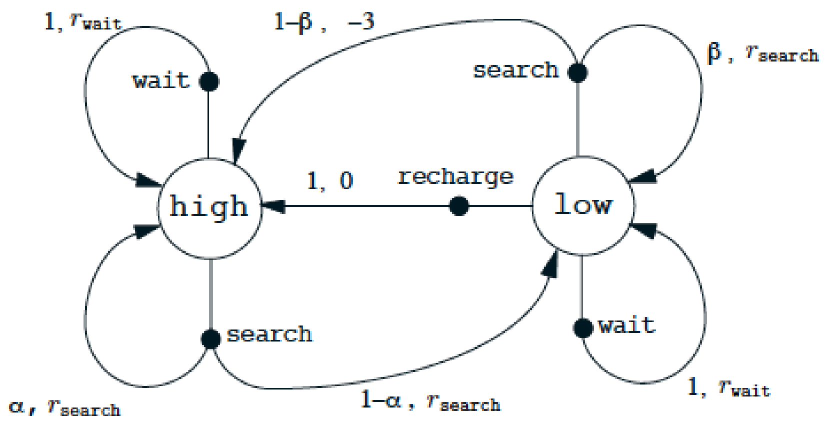
- State: the information to determine state in a trajectory
  - $S_t = f(H_t)$
  - Environment State: private representation of the environment
  - Agent State: agent internal representation
  - Information State (Markov Property): useful information from the history

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



Fully and Partially Observable Environments

- Full observation:
  - Agent fully observes environment state
  $$O_t = S_t^a = S_t^e$$
  - Agent State = environment state = information state
  - Markov Decision Process (detail later)
- Partially observability: agent indirectly or partially observes environment
  - Robot with first view cameras
  - Agent state differ from environment state
  - Agent must construct its own state representation

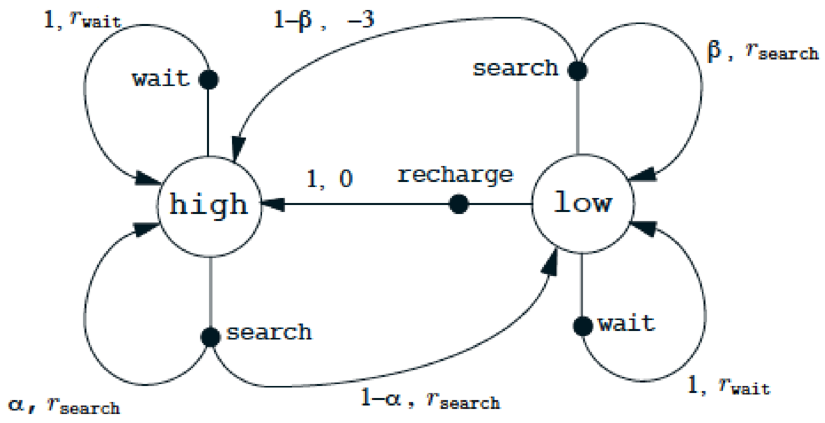


Major Component of an RL agent

- Policy - maps current state to action
- Value function - prediction of value for each state and action
- Model - agent’s representation of the environment.

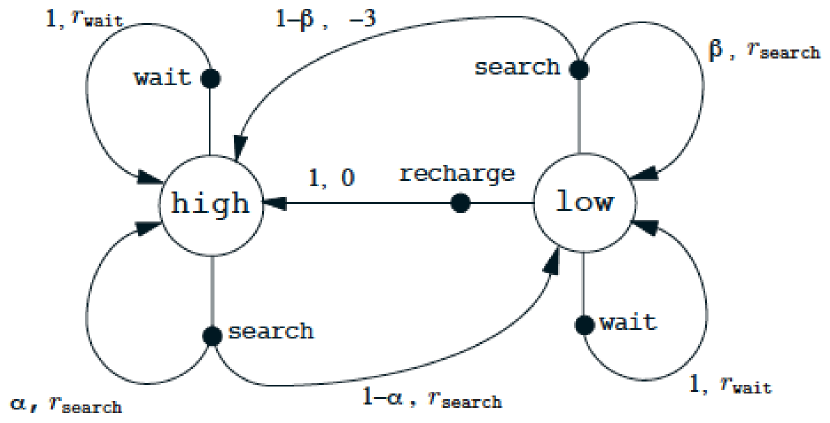
Policy

- Policy: agent's behavior, how is act in the environment
- Map from state to action
- Deterministic policy: $a = \pi(s)$
- Stochastic: $\pi(a|s) = \mathbb{P}[A_t = a | S_t = s]$



Value Function

- Value Function: a prediction of future reward (how many, how much future reward the agents expect)
- Used to evaluate the goodness/badness of state
- Agent select action to chose the best state based on value function (with maximized expected reward)



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

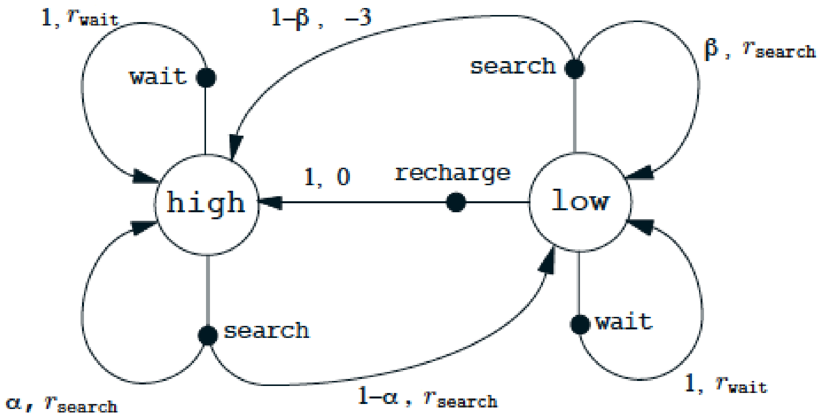
Model

- To model environments, predict what the environments will do
- P: to predict the next state

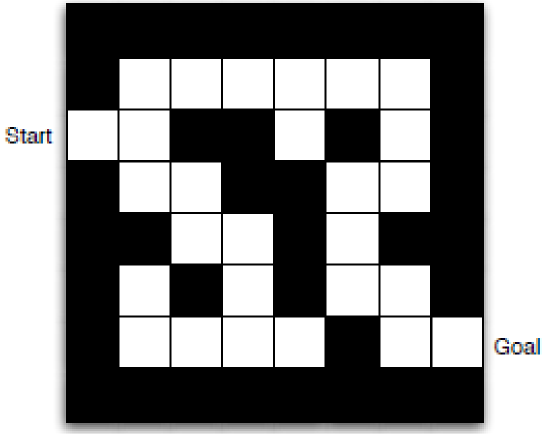
$$\mathcal{P}_{ss'}^{a} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$$

- R: to predict immediate (not future) reward

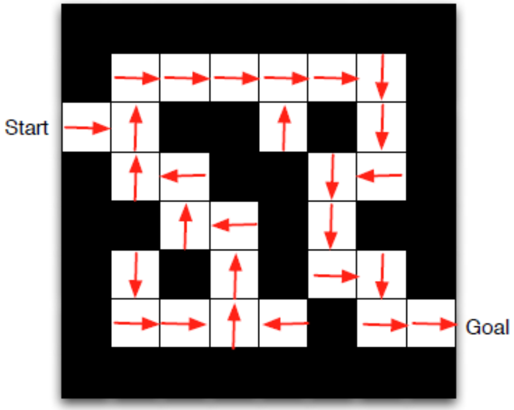
$$\mathcal{R}_{s}^{a} = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$$



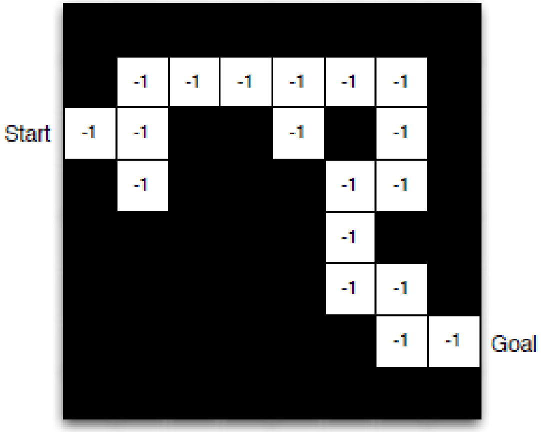
Maze Example



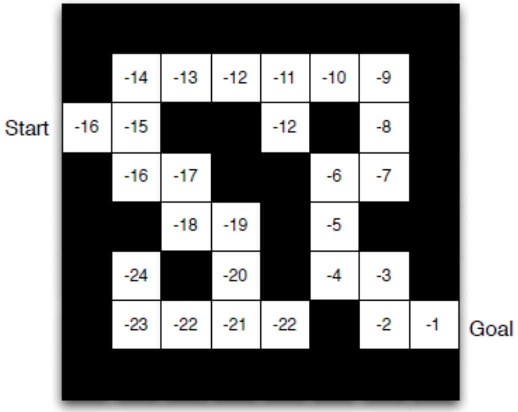
Rewards: -1 per time-step
Actions: N, E, S, W
States: Agent's location



Policy



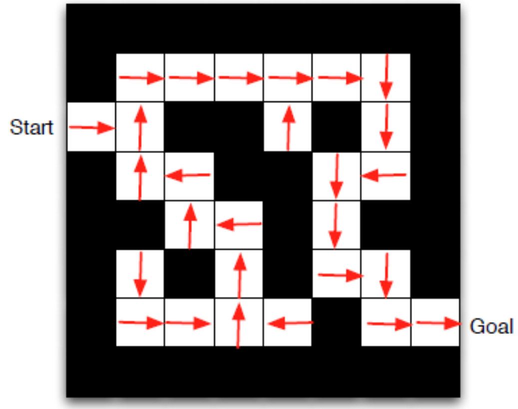
Model



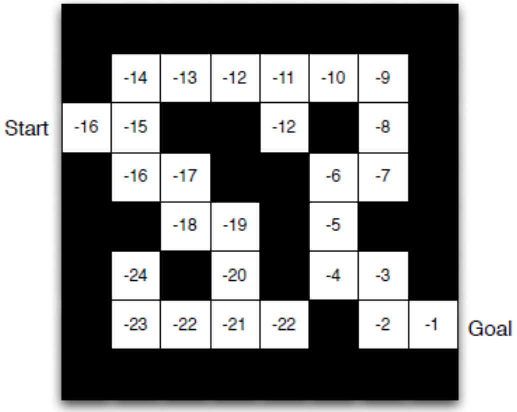
Value function

Categorizing Reinforcement Learning Agents

- Agents Action:
  - Value Based: Value function, no policy
  - Policy Based: Policy, no value function
  - Actor Critic: Both Policy and Value Function
- Modelling environment
  - Model Free: interacting directly environments
  - Model Based: Learn and model environments



Policy

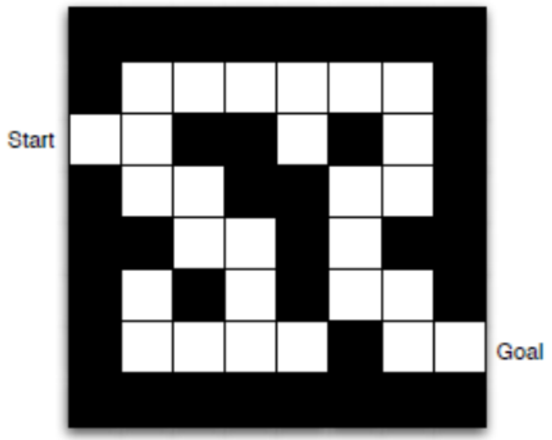


Value function

Learning and Planning

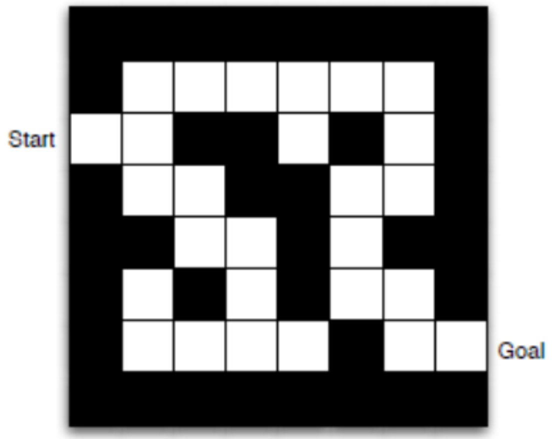
Sequence Decision Making

- Reinforcement Learning
  - Environments is initially unknown
  - Agent interacts with the environment
  - Agent improves policies
- Planning
  - Models of environment are known
  - Action by functional computation
  - Agent improve policies



Exploration and Exploitation

- Solve problem in trial-error learning
- Agents must learn to have good policies
- Agents learn from acting with their environments
- Reward may not response each step, it may be at the end of games
- Exploration: discovering the environment
- Exploitation: planning with maximal reward
- Trading between exploration and exploitation



Recap on RL introduction

- Sequence of decision, reward
- State, fully observation, partially observation
- Main components: Policy, Value Function, Model
- Categorizing RL agents
- Learning and Planning
- Discuss application of reinforcement learning

Markov Decision Process

Markov decision process: Model of finite-state environment

Bellman Equation

Dynamic Programming

Markov Decision Process (Model of the environment)

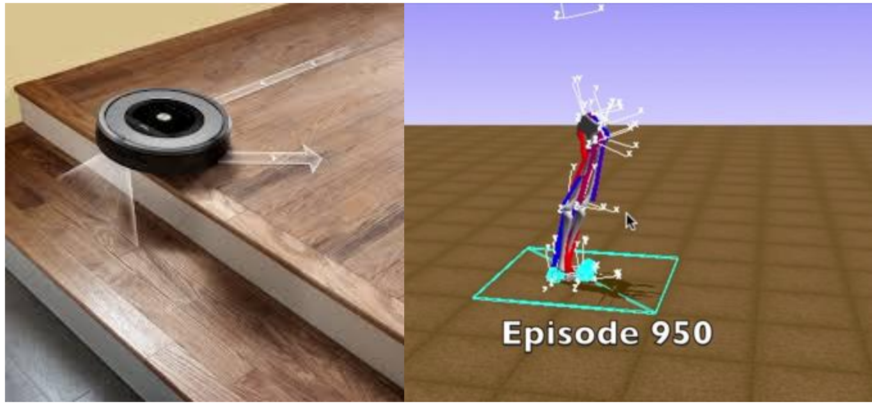
- Terminologies:

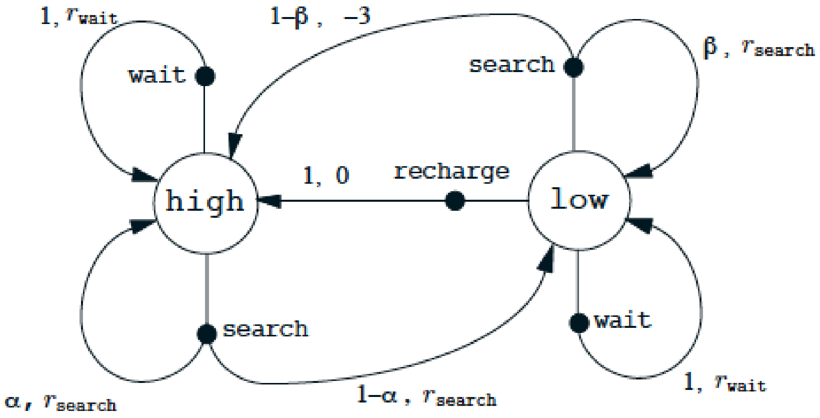
In a Markov Decision Process:

s, s'	states
a	action
r	reward
$\mathcal{S}$	set of all nonterminal states
$\mathcal{S}^+$	set of all states, including the terminal state
$\mathcal{A}(s)$	set of all actions possible in state s
$\mathcal{R}$	set of all possible rewards

t	discrete time step
$T, T(t)$	final time step of an episode, or of the episode including time t
A_t	action at time t
S_t	state at time t, typically due, stochastically, to S_{t-1} and A_{t-1}

$p(s', r \mid s, a)$	probability of transition to state s' with reward r, from state s and action a
$p(s' \mid s, a)$	probability of transition to state s', from state s taking action a

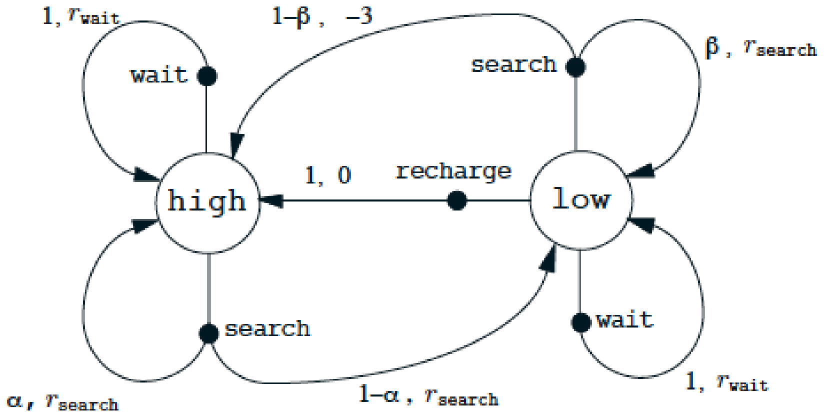




Markov Decision Process

- Markov property: The distribution over future states **depends only on the present state and action**, not on any other previous event.

$$p(s', r | s, a) \doteq \Pr\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\},$$



- Maximize return
  - Episodic task: consider return over finite horizon (e.g. games, maze).

$$U_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_T$$

  - Continuing task: consider return over infinite horizon (e.g. juggling, balancing).

$$U_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} \ldots = \sum_{k=0:\infty} \gamma^k r_{t+k}$$

How we get good decision?

- Defining behavior: the policy
  - Policy: defines the action-selection strategy at every state

π	policy, decision-making rule
$\pi(s)$	action taken in state s under *deterministic* policy π
$\pi(a\|s)$	probability of taking action a in state s under *stochastic* policy π

  - Goals: finds the policy that maximizes expected total reward

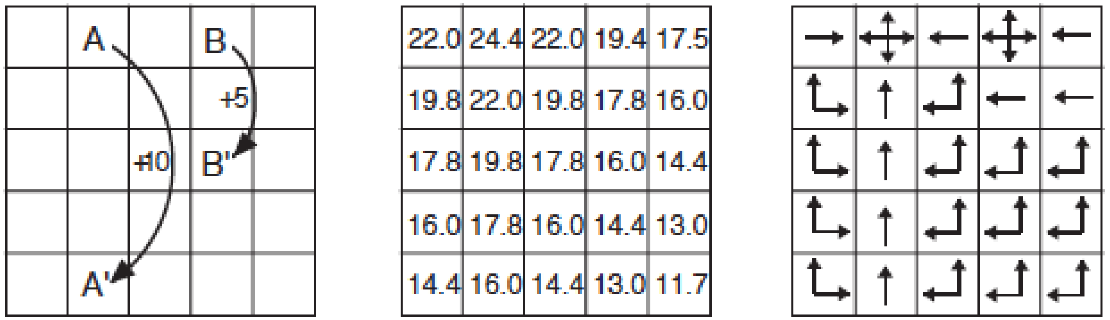
$$\text{argmax}_{\pi} E_{\pi} [r_0 + r_1 + \ldots + r_T \mid s_0]$$

Value functions

- The expected return of a policy for a state is call value function

$$V^{\pi}(s) = E_{\pi}\,[r_t + r_{t+t} + \ldots + r_T \mid s_t = s\,]$$

- Strategy to find optimal policy
  - Enumerate the space of all policies
  - Estimate the expected return of each one
  - Keep the policy that has maximum expected return



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

Gridworld v_* π_*

Gridworld example
- Reward to Off grid: -1
- Reward to On grid: 0
- Reward exception at A, B

Value functions

- Value of a policy

$$V^{\pi}(s) = E_{\pi}[r_t + r_{t+1} + \ldots + r_T \mid s_t = s]$$

$$V^{\pi}(s) = E_{\pi}[r_t] + E_{\pi}[r_{t+1} + \ldots + r_T \mid s_t = s]$$

$$V^{\pi}(s) = \underbrace{\sum_{a \in A} \pi(s,a) R(s,a)}_{\text{Immediate reward}} + \underbrace{E_{\pi}[r_{t+1} + \ldots + r_T \mid s_t = s]}_{\text{Future expected sum of rewards}}$$

$$V^{\pi}(s) = \sum_{a \in A} \pi(s,a) R(s,a) + \underbrace{\sum_{a \in A} \pi(s,a) \sum_{s' \in S} T(s,a,s')}_{\text{Expectation over 1-step transition}} E_{\pi}[r_{t+1} + \ldots + r_T \mid s_{t+1} = s']$$

Note: $T(s,a,s') = p(s'|s,a)$

$$V^{\pi}(s) = \sum_{a \in A} \pi(s,a) R(s,a) + \sum_{a \in A} \pi(s,a) \sum_{s' \in S} T(s,a,s') \underbrace{V^{\pi}(s')}_{\text{By definition}}$$

Bellman’s equation

- State value function (for a fixed policy with discount)

$$V^{\pi}(s) = \sum_{a \in A} \pi(s,a)\, [\, \underbrace{R(s,a)}_{\textit{Immediate}} + \underbrace{\gamma \sum_{s' \in S} T(s,a,s') V^{\pi}(s')\,]}_{\textit{Future expected sum of rewards}}$$

- State-action value function (Q-function)

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') [\sum_{a' \in A} \pi(s',a') Q^{\pi}(s',a')]$$

- When S is a finite set of states, this is a system of linear equations (one per state)
- Belman’s equation in matrix form: $V^{\pi} = R^{\pi} + \gamma\, T^{\pi}\, V^{\pi}$

$$Q^{\pi}(s,a) = r(s,a) + \gamma \sum_{s' \in S} p(s'|a,s) V^{\pi}(s')$$

Optimal Value, Q and policy

- Optimal V: the highest possible value for each s under any possible policy
- Satisfies the bellman Equation: $V^*(s) = \max_a \left[r(s,a) + \gamma \sum_{s' \in S} p(s'|a,s) V^*(s') \right]$
- Optimal Q-function: $Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in S} p(s'|a,s) V^*(s')$
- Optimal policy: $\pi^*(s,a) = \arg\max_a Q^*(s,a)$

Dynamic Programming (DP)

- Assuming full knowledge of Markov Decision Process
- It is used for planning in an MDP
- For prediction
  - Input: MDP (S,A,P,R,γ) and policy π
  - Output: value function v_π
- For controlling
  - Input: MDP (S,A,P,R,γ) and policy π
  - Output: Optimal value function v_* and optimal policy π_*

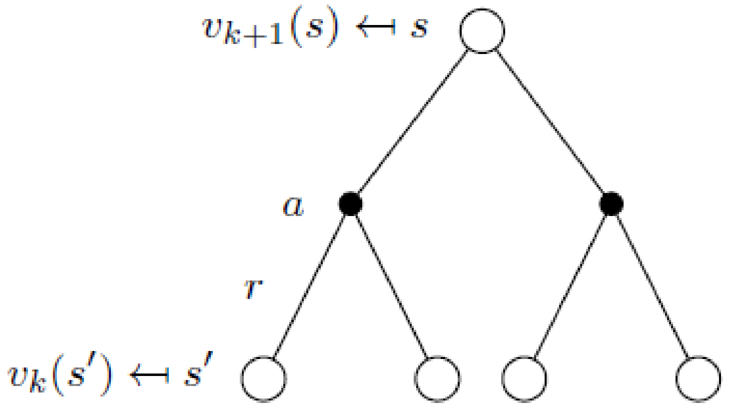
DP: Iterative Policy Evaluation

- Main idea of Dynamic Programming: turn Bellman equations to update rules
- Problem: evaluate a given policy π
- Iterative policy evaluation: Fix policy

Iterative policy evaluation

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

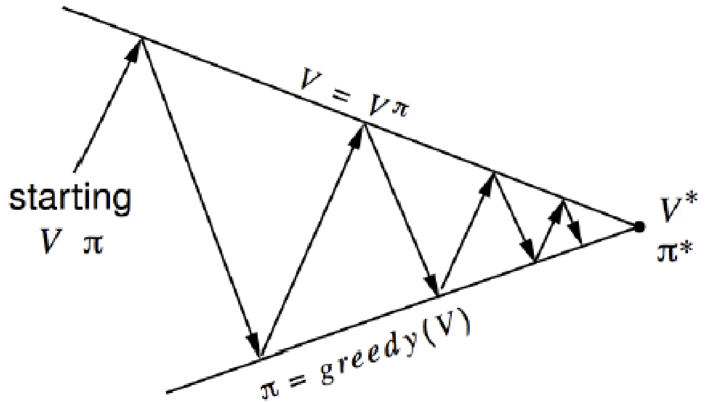
Bellman eq: $V^\pi = R^\pi + \gamma P^\pi V^\pi$



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

$$\mathbf{v}^{k+1} = \boldsymbol{\mathcal{R}}^{\boldsymbol{\pi}} + \gamma \boldsymbol{\mathcal{P}}^{\boldsymbol{\pi}} \mathbf{v}^k$$

DP: Improving a Policy



- Finding a good policy: Policy iteration

$$\pi_0 \xrightarrow{\mathrm{E}} v_{\pi_0} \xrightarrow{\mathrm{I}} \pi_1 \xrightarrow{\mathrm{E}} v_{\pi_1} \xrightarrow{\mathrm{I}} \pi_2 \xrightarrow{\mathrm{E}} \cdots \xrightarrow{\mathrm{I}} \pi_* \xrightarrow{\mathrm{E}} v_*,$$

- Start with an initial policy π_0 (e.g. random)
- Repeat:
  - Compute V^π, using iterative policy evaluation.
  - Compute a new policy π' that is greedy with respect to V^π
- Terminate when $\pi = \pi'$

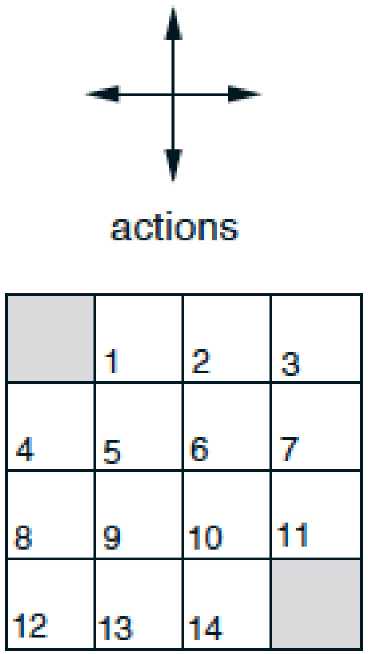
Policy iteration (using iterative policy evaluation)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Gridworld example



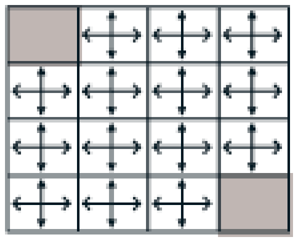
$R = -1$
on all transitions

v_k for the Random Policy

Greedy Policy w.r.t. v_k

$k = 0$

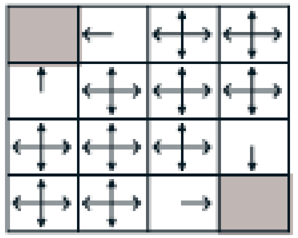
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



random policy

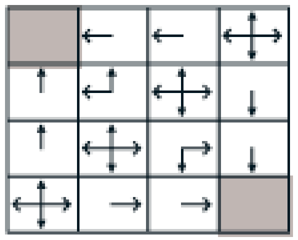
$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

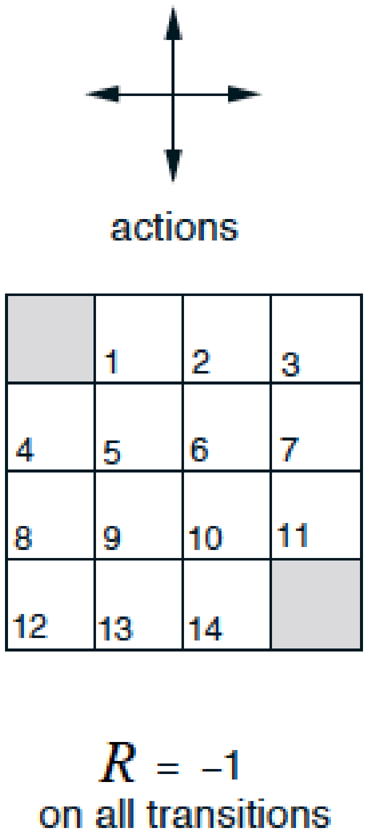


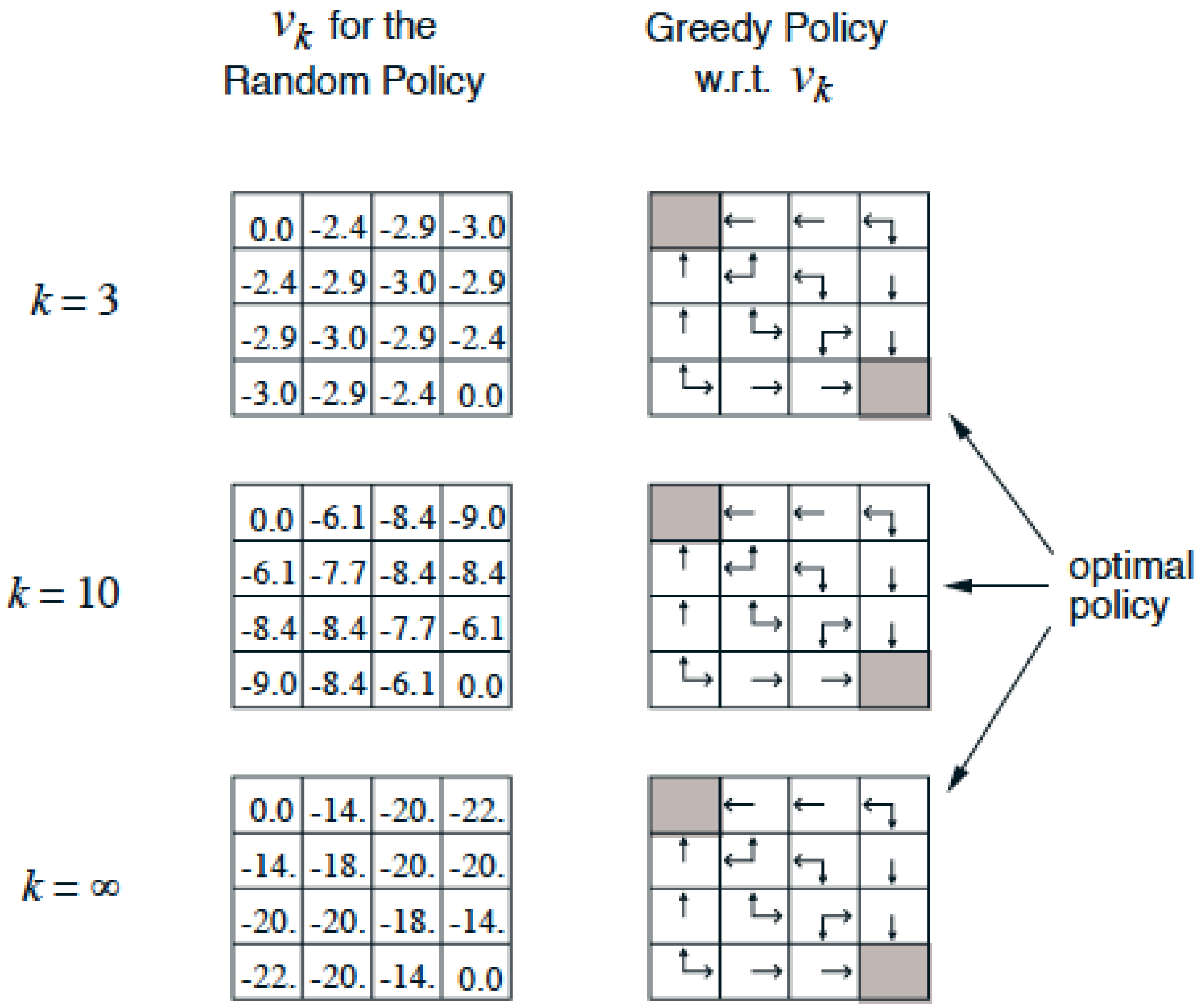
$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0



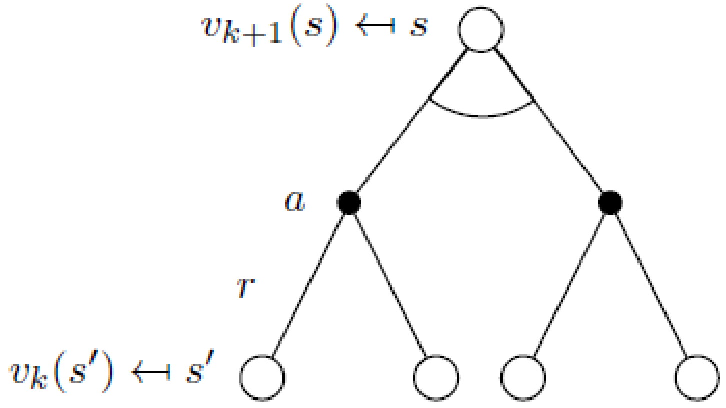
Gridworld example





DP: Value Iteration

- Finding a good policy: Value iteration
  - Drawback of policy iteration: evaluate policy also needs iteration
  - Main idea: Turn the Bellman optimality equation into an iterative update rule (same policy evaluation)



Value iteration

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$

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

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

DP: Pros and Cons

- Rarely use Dynamic programming in real applications
  - To calculate we must access environment model, fully observe with knowledge of environment.
  - Extending to continues actions and state
- However:
- Mathematically exact, expressible and analyzable
  - Good deals for small problem.
  - Stable, simple and fast

Visualization and Codes

- https://cs.stanford.edu/people/karpathy/reinforcejs/index.html

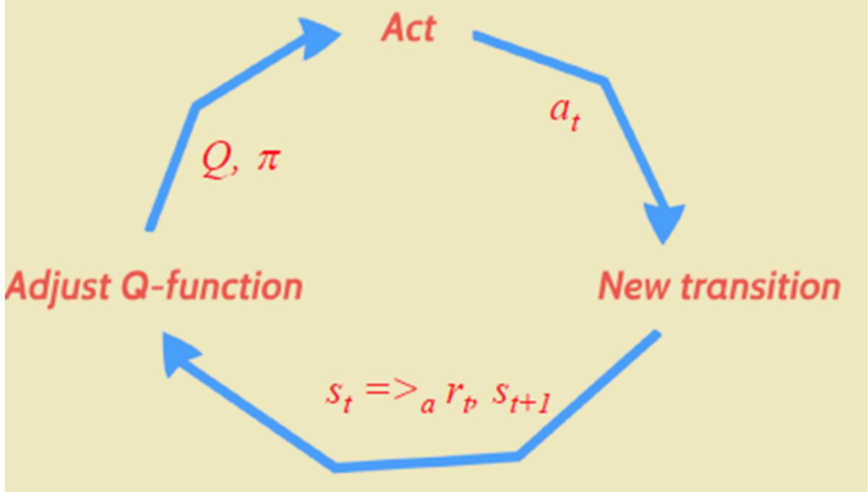
Online Learning

Model-free Reinforcement Learning

Partially observable environment, Monte Carlo, TD, Q-Learning

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

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

- Total discounted reward

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

- Value function is expected return

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

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

State Visit Monte-Carlo Policy Evaluation

- To evaluate state s
- At time-step t that state s is visited in an episode
  - Visiting state s: first or every time-step
- Increase counter N(s) = N(s) + 1
- Increate total return S(s) = S(s) + G_t
- Value is estimated by mean return V(s) = S(s)/N(s)
- By law of large number $V(s) \rightarrow v_\pi(s)$ as $N(s) \rightarrow \infty$

Incremental Monte-Carlo Updates

- Learning from experience
- Update V(s) incrementally after full game $S_1, A_1, R_2, ..., S_T$
- For each state S_t, with actual return G_t

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

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

- With learning rate

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

Temporal-Difference Learning

- Model-free: no knowledge of MDP
- Do not wait for episodes, learn from incomplete episode by bootstrapping
- Update value $V(s_t)$ toward estimated return $R_{t+1} + \gamma V(S_{t+1})$

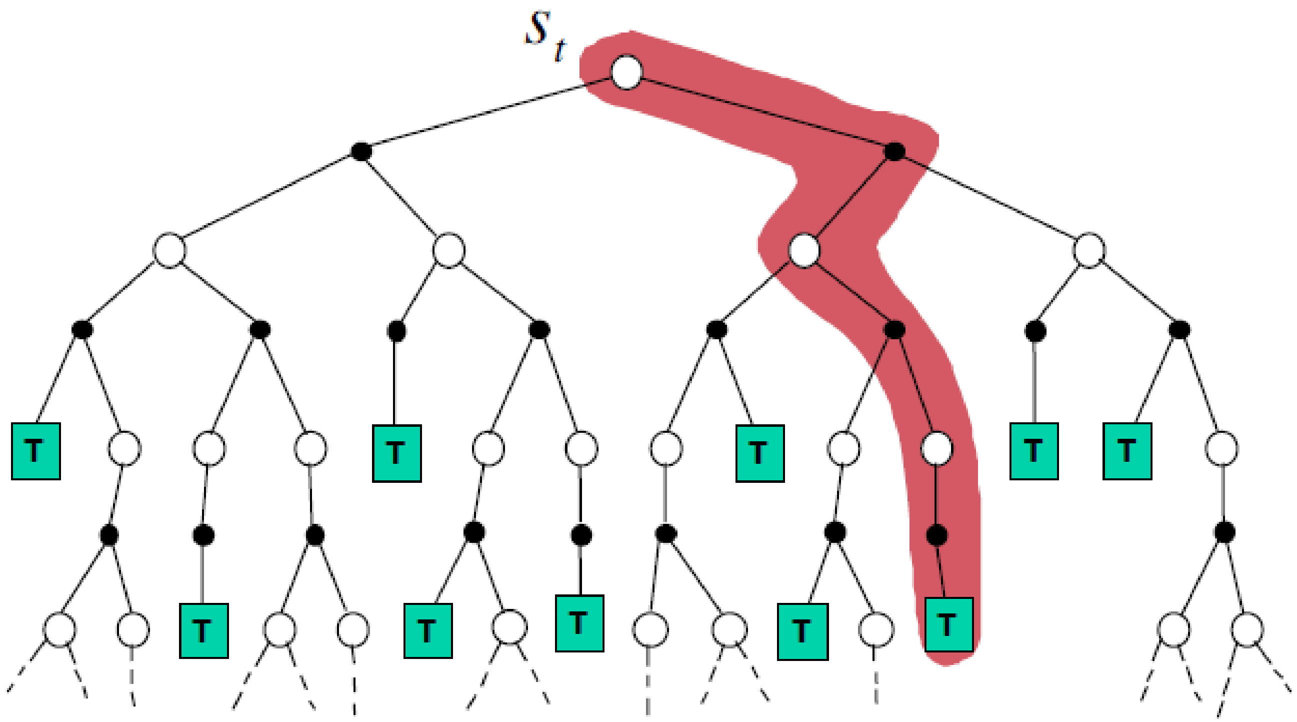
$$V(S_t) \leftarrow V(S_t) + \alpha \underbrace{(\overbrace{R_{t+1} + \gamma V(S_{t+1})}^{\text{TD Target}} - V(S_t))}_{\text{TD error}}$$

Monte-Carlo and Temporal Difference

- 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

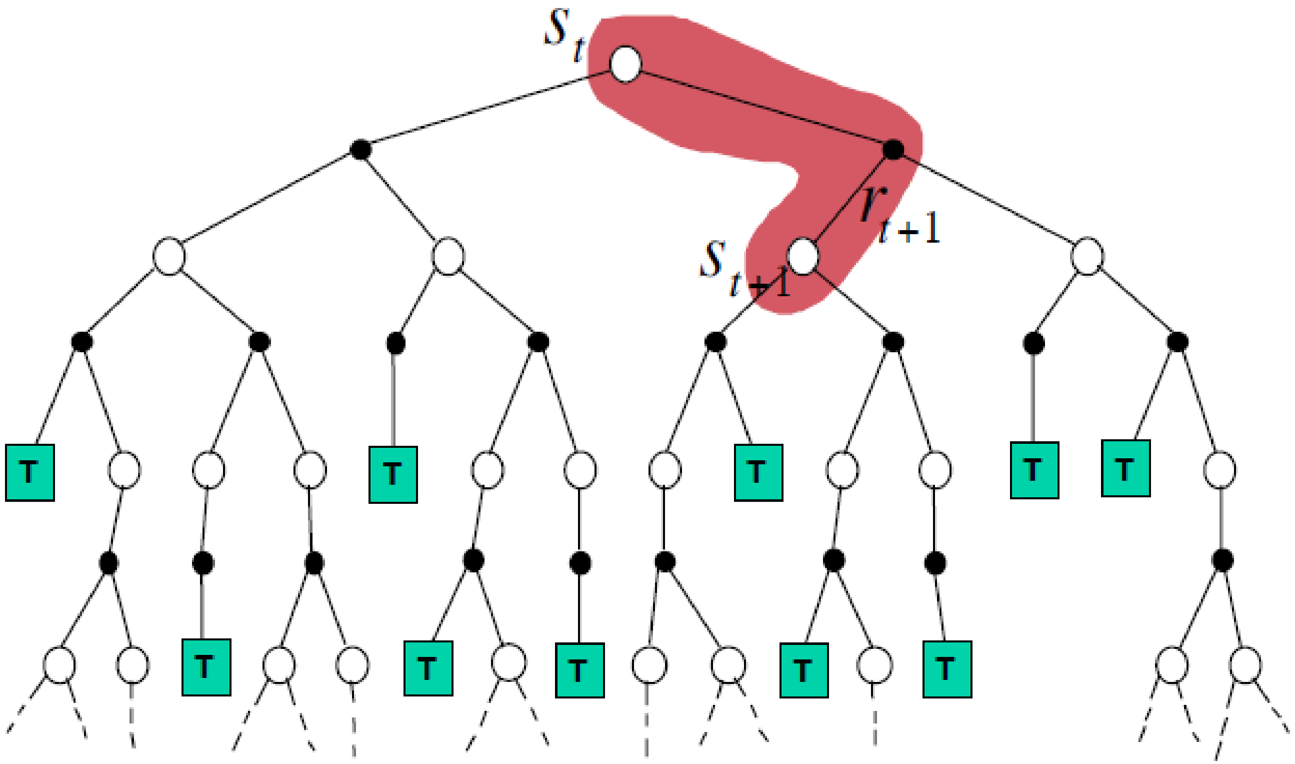
Monte-Carlo Backup

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



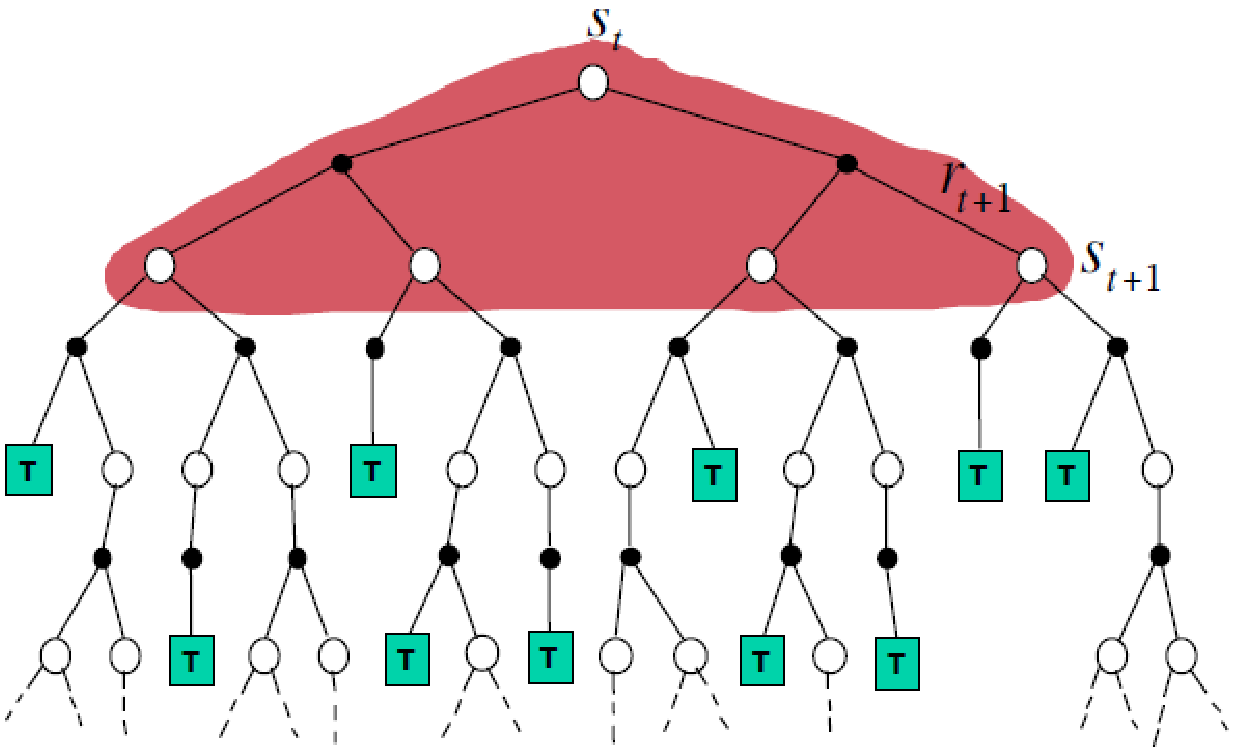
Temporal-Difference Backup

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



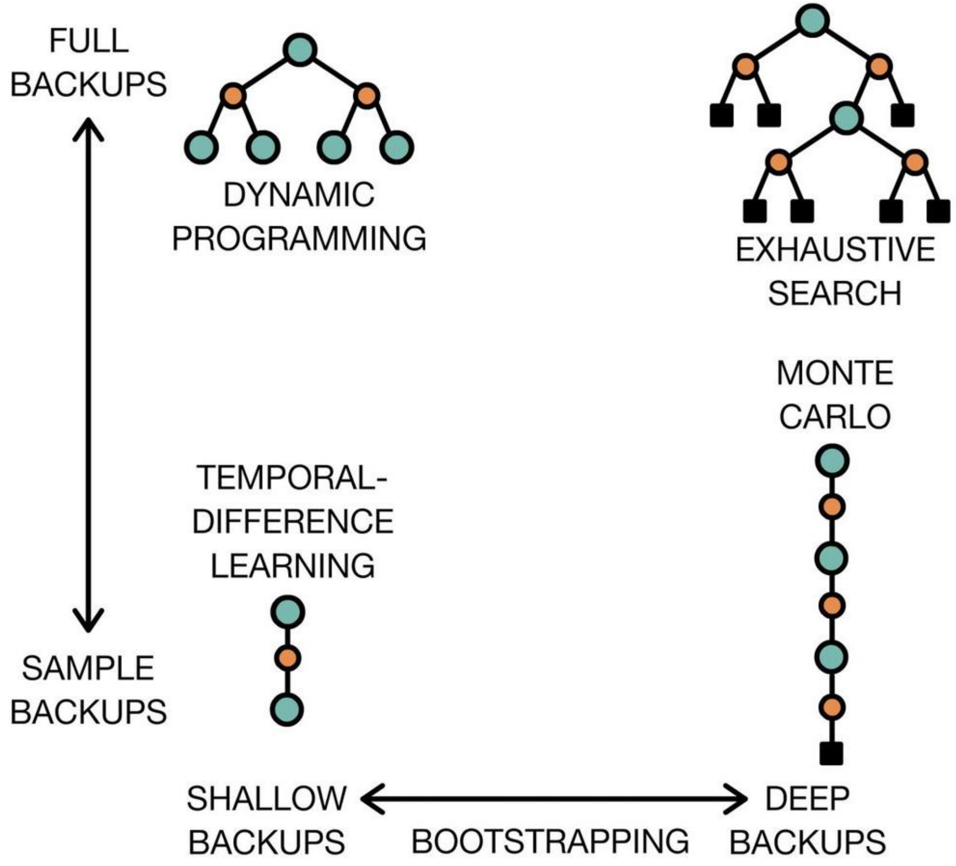
Dynamic Programming Backup

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



Bootstrapping and Sampling

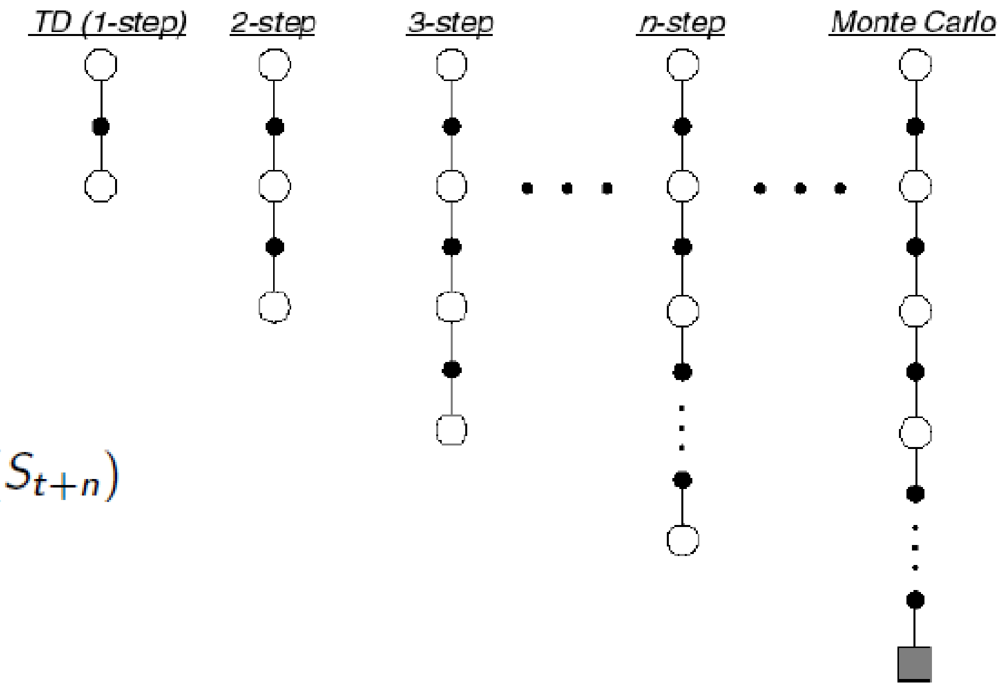
- Bootstrapping: update involves an estimate
  - MC does not bootstrap
  - DP bootstraps
  - TD bootstraps
- Sampling: update samples an expectation
  - MC samples
  - DP does not sample
  - TD samples



N-step prediction

- n-step return

$$
\begin{array}{lll}
n = 1 & (TD) & G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1}) \\
n = 2 & & G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \\
\vdots & & \vdots \\
n = \infty & (MC) & G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T
\end{array}
$$



- Define n-step return

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

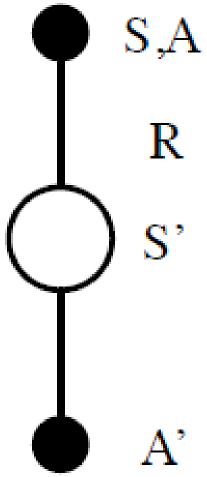
- n-step temporal-difference learning

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

On-policy Learning

- Advantage of TD:
  - Lower variance
  - Online
  - Incomplete sequence
- Sarsa:
  - Apply TD to Q(S,A)
  - Use policy improvement eg ϵ-greedy
  - Update every time-step

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



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

Sarsa Algorithm

Initialize any Q(s,a) and Q (terminate-state,null) =0
Repeat (for each episode)
    Initialize S
    Choose A from S using Q (eg ϵ-greedy)
    Repeat (for steps of episode)
        Take A, observe R, S'
        Chose A' from S' using Q (eg ϵ-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
        $S \leftarrow S';\ A \leftarrow A';$
Until S is terminal

Off-Policy Learning

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

$$\{S_1, A_1, R_2, \ldots, S_t\} \sim \mu$$

- Advantages:
  - Learning from observing human or other agents
  - Reuse experience generated from old policies $\pi_1, \pi_2, \pi_3, \ldots, \pi_{t-1}$
  - Learn about optimal policy while following exploratory policy
  - Learn about multiple policies while following one policy

Q-Learning

- Off-policy learning action-value Q(s,a)
- No importance sampling is required
- Off policy: Next action is chosen by $A_{t+1} \sim \mu(\cdot|S_t)$
- Q-Learning: choose alternative successor $A' \sim \pi(\cdot|S_t)$
- Update Q(St,At) towards value of alternative action

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

- Improve policy by greedy

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

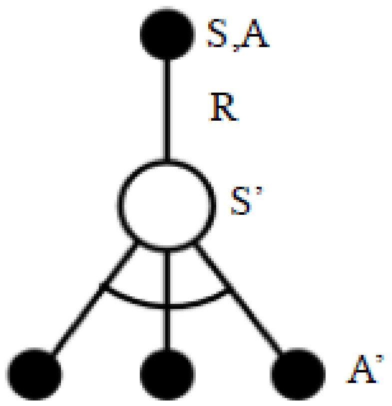
Q-Learning

Update equation

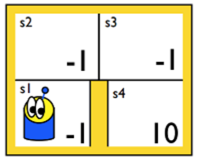
$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma \max_{a'} Q(S',a') - Q(S,A)\right)$$

Algorithm

Initialize $Q(s,a)$ arbitrarily
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'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$;
  until s is terminal



Q-Learning Table version



$\alpha = .7$

	↑	↓	←	→
S_1	0	0	0	0
S_2	0	0	0	0
S_3	0	0	0	0
S_4	0	0	0	0

Q-Table

Visualization and Codes

- https://cs.stanford.edu/people/karpathy/reinforcejs/index.html
- https://github.com/awjuliani/DeepRL-Agents/blob/master/Q-Table.ipynb
- https://gym.openai.com/

More on RL

Approximation Function

Exploration and Exploitation

Challenge Discussion

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

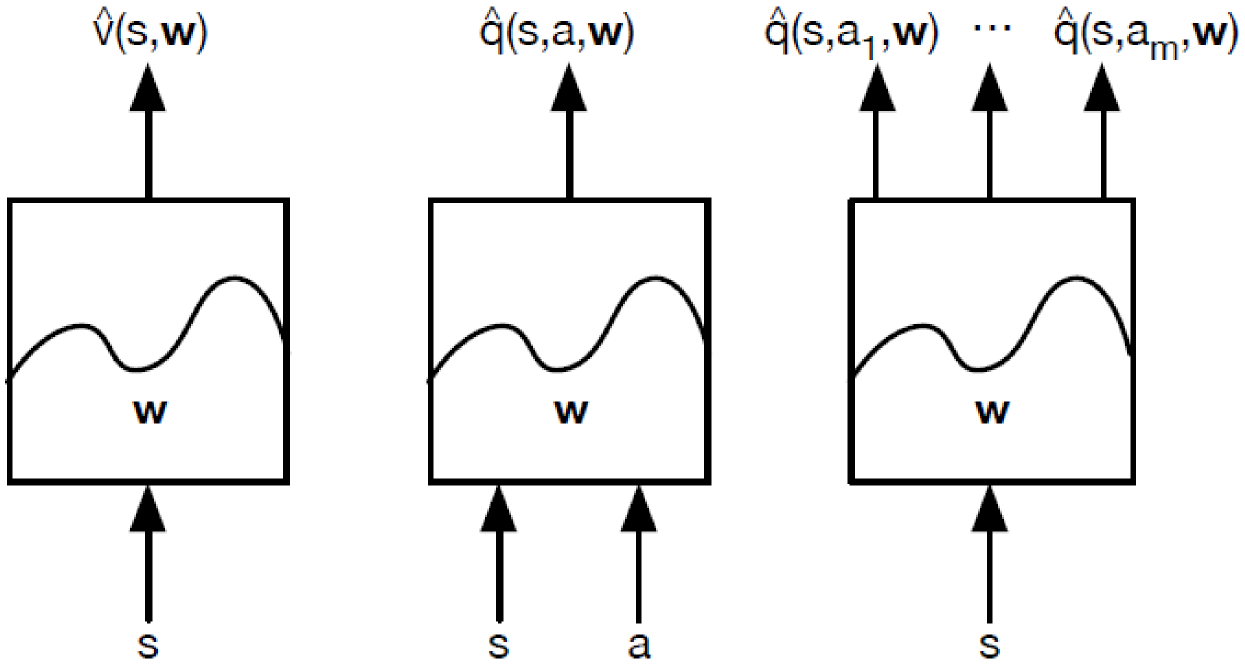
$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

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

  - Generalize from seen states to unseen states

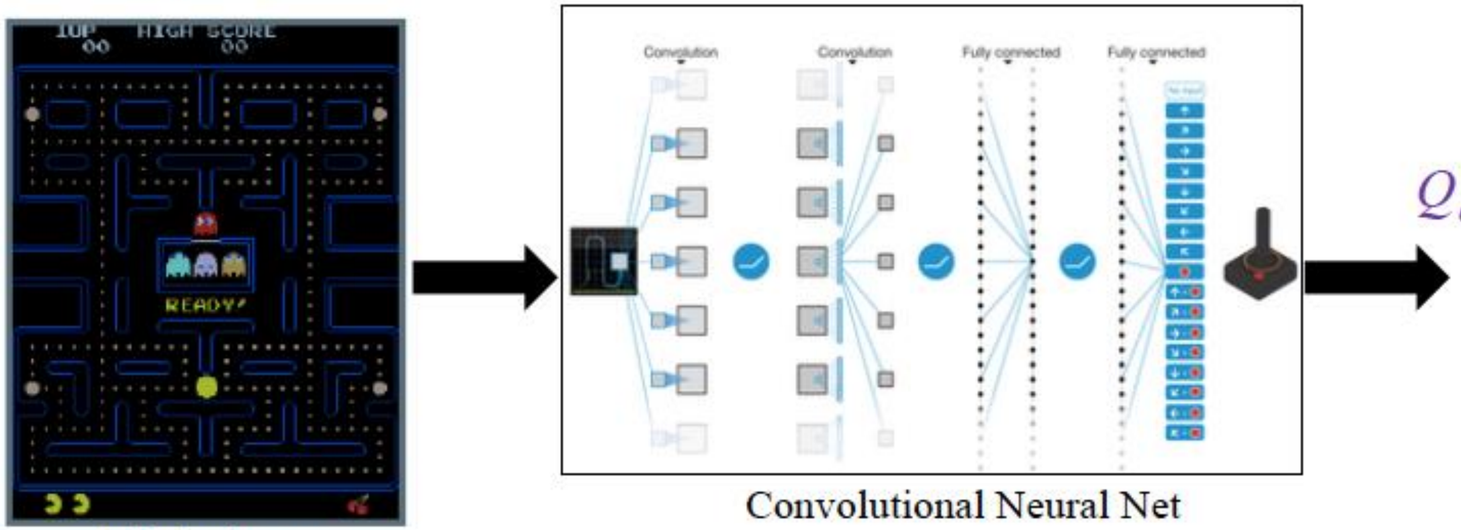
Type of Value Function Approximation

- Differentiable function approximation
    - Linear combination of feature
        - Robots: distance from checking point, target, dead mark, wall
        - Business Intelligence Systems: Trends in stock market
    - Neural Network
        - Deep Reinforcement Learning
- Training strategies



Deep Reinforcement Learning: DeepNN + RL

s



Original state

Convolutional Neural Net

$Q_\theta(s,a)$

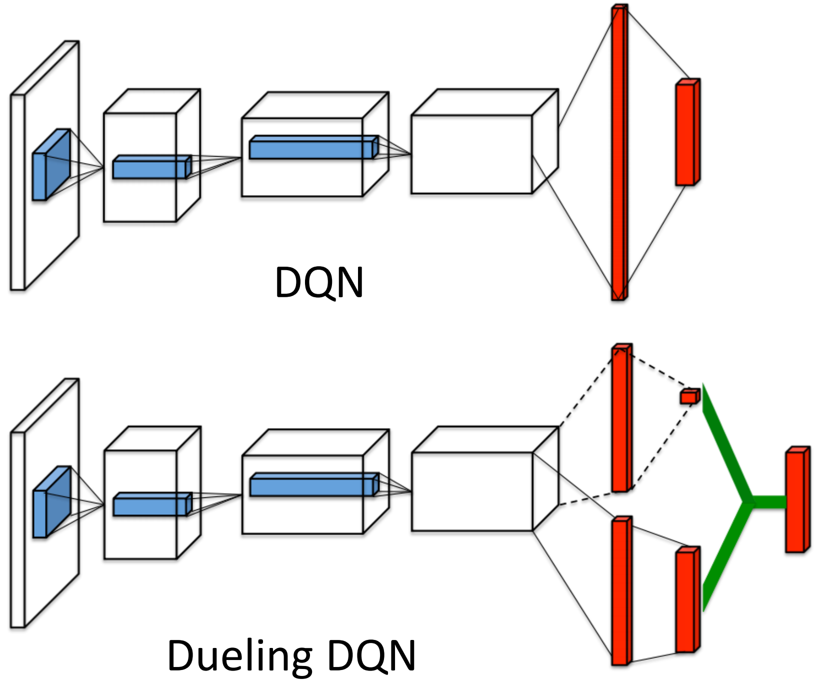
Deep Q-Network trained with stochastic gradient descent.

[DeepMind: Mnih et al., 2015].

Deep Q-Network

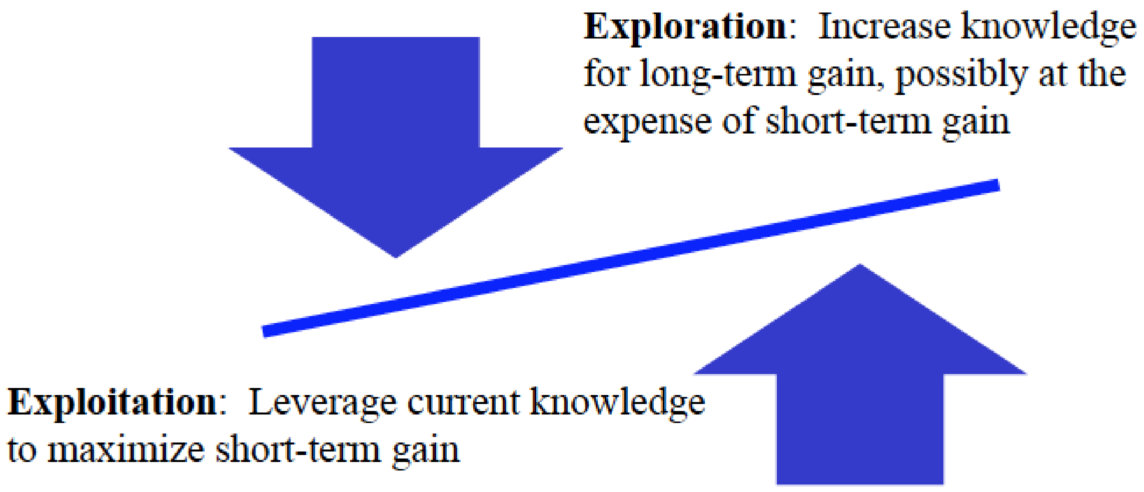
- Deep Q-Network: experience replay and target network
  - Copy network from Q-value function
  - Store experience and sample for training
  - More like supervised learning
- Dueling Q-Learning
  - Separate value and advance functions

    Q(s,a) =V(s) + A(a)
  - Combine them back into a single Q-function at the final layer



Exploration vs Exploitation

- Online decision-making
  - 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



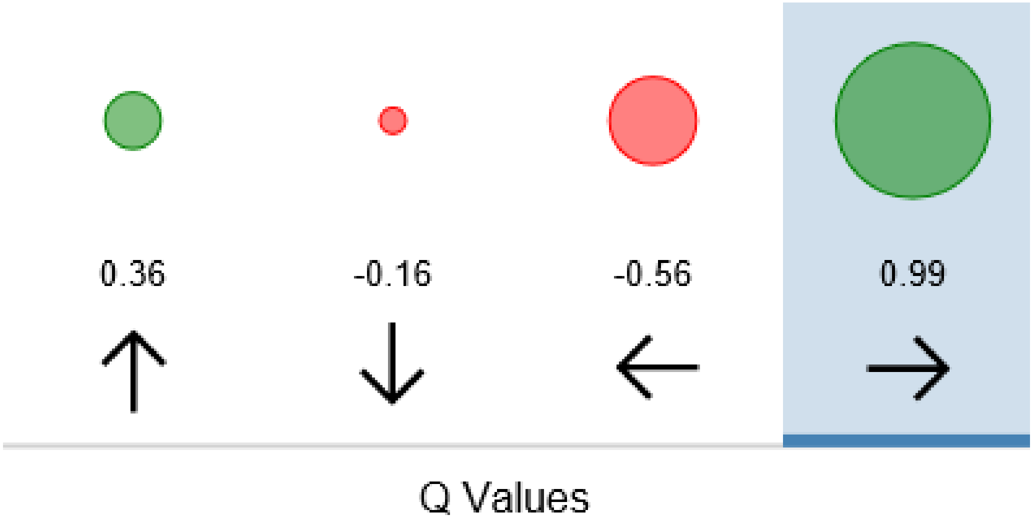
Trade-off between Exploration and Exploitation

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

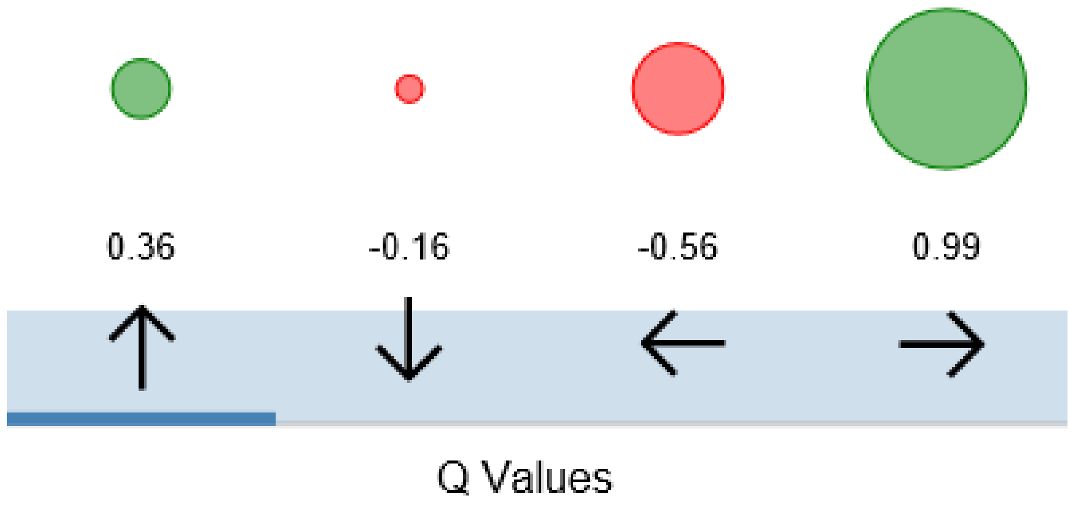
Greedy Approach

- Simply choose the action provide the greatest reward
- Best at the current moment, current knowledge
- No potential exploration



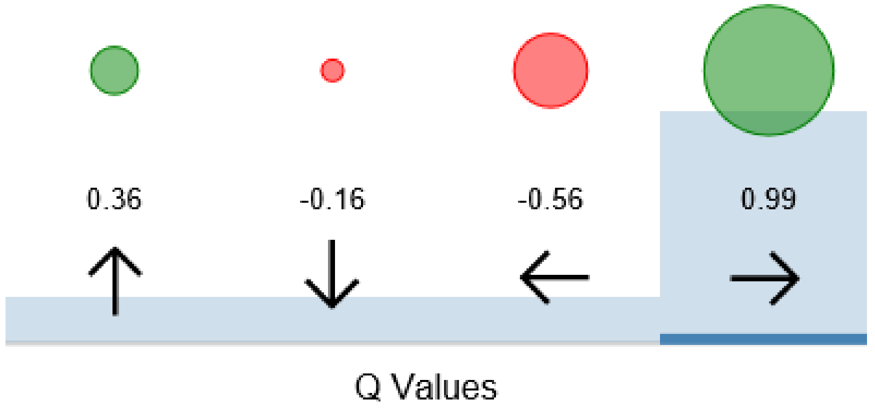
Random Approach

- Opposite to greedy selection: simply choose random action
- More to explore the environment, slow to converge
- Useful in DQN to initial means of sampling to fill an experience buffer



ϵ-greedy

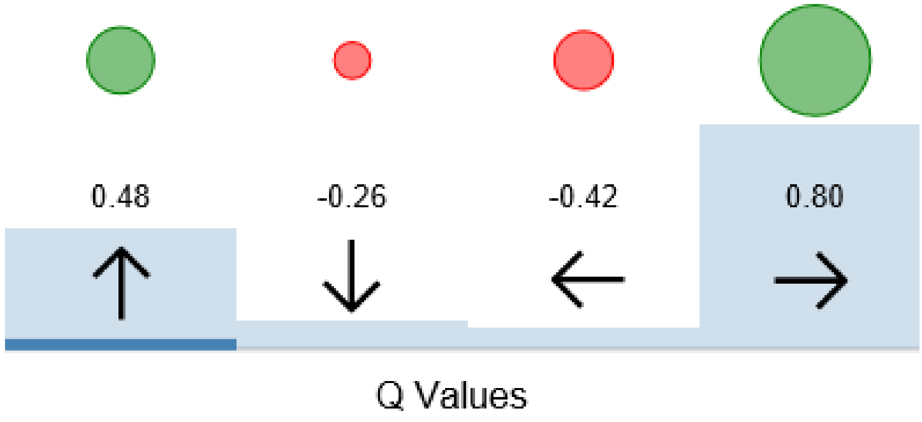
- Combination of greedy and random
- Sometime choose optimal, sometime choose random
- Adjustable parameter ϵ determines the probability of taking a random
- Technique for most recent reinforcement learning algorithms, including DQN and its variants
- Shortcomings: far from optimal



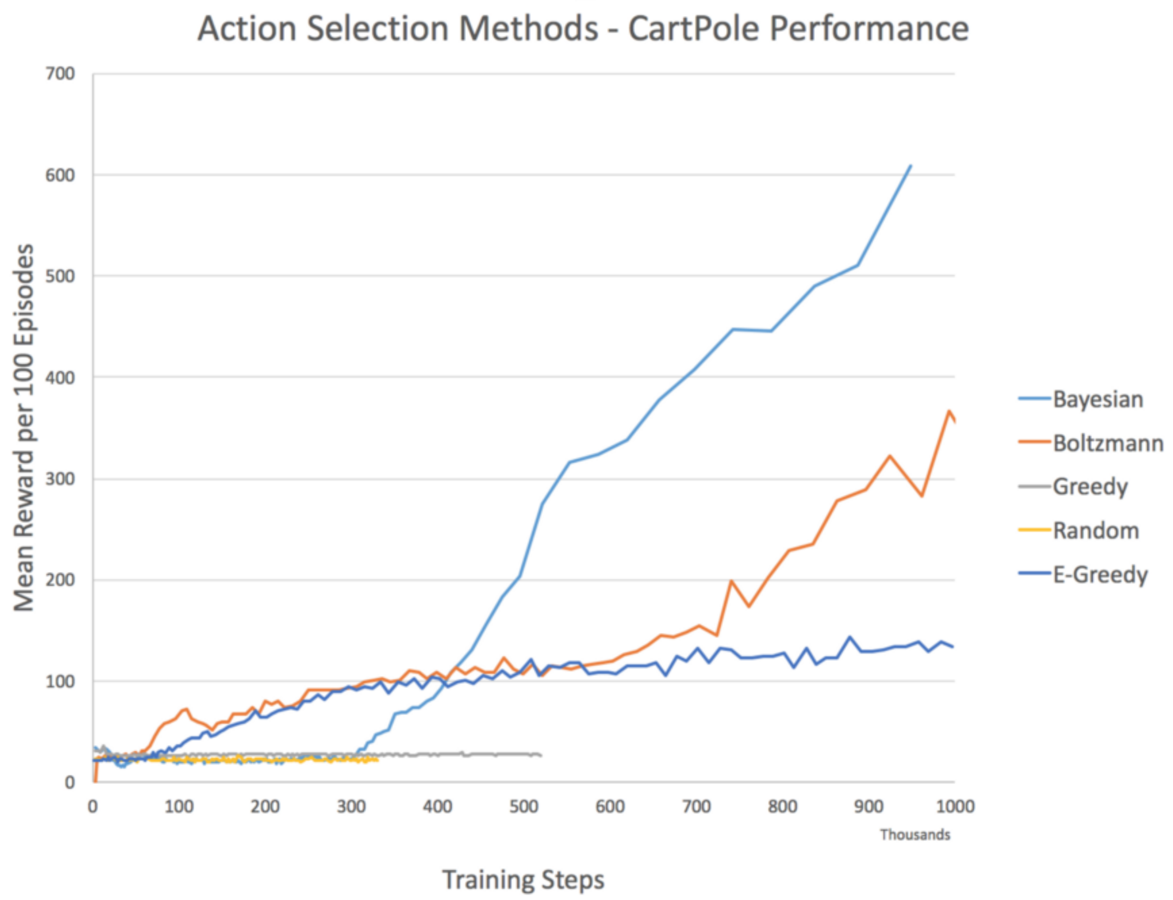
Boltzmann Approach

- Choosing actions with weighted probabilities.
- Use a softmax over networks estimate of each action
- Using parameter to control spread of softmax distribution
- Shortcoming: Based on agent estimating how optimal the action is

$$P_t(a) = \frac{\exp(q_t(a)/\tau)}{\sum_{i=1}^{n} \exp(q_t(i)/\tau)},$$



Comparison of Action Selection Methods

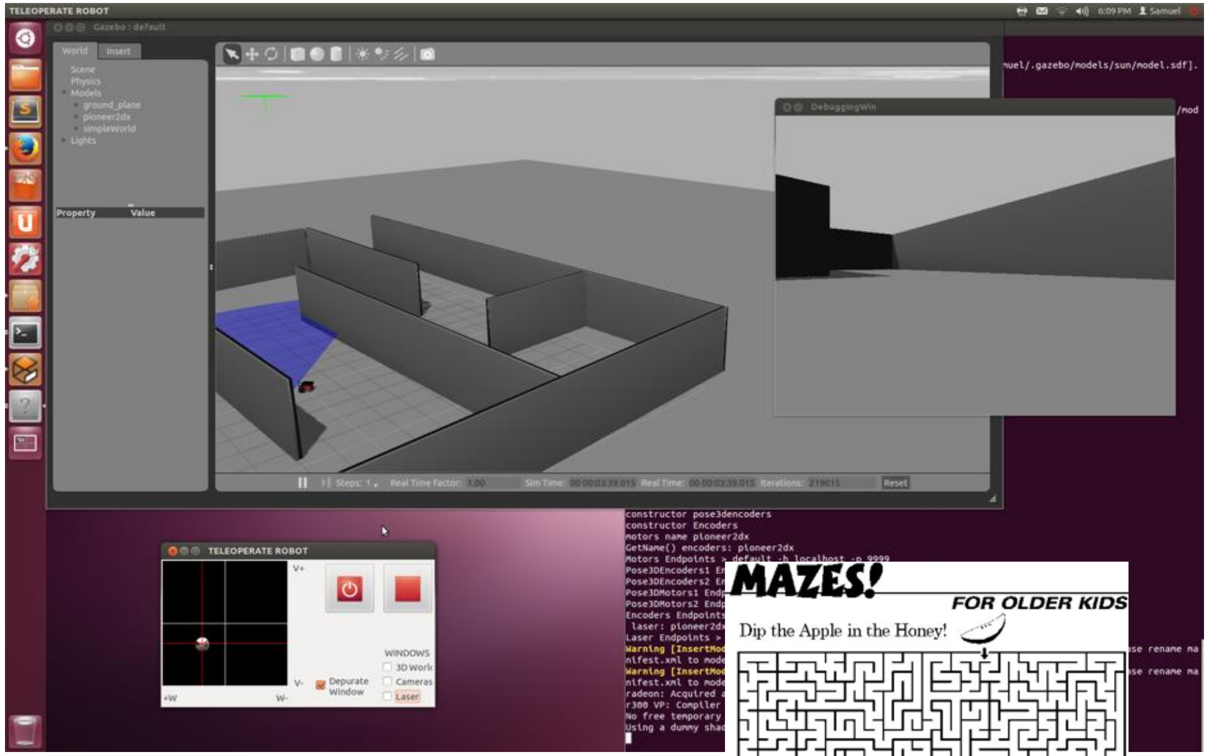


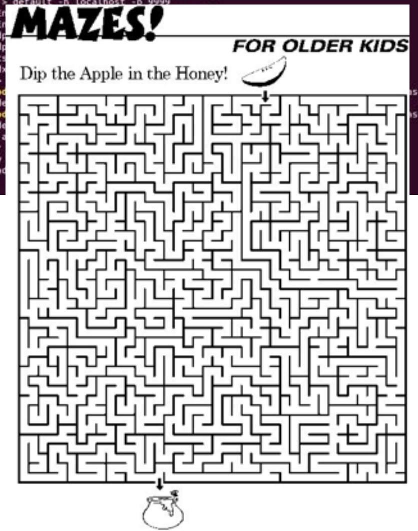
Visualization and Codes

- https://cs.stanford.edu/people/karpathy/reinforcejs/index.html
- https://github.com/awjuliani/DeepRL-Agents/blob/master/Q-Network.ipynb
- http://awjuliani.github.io/exploration/index.html
- https://github.com/awjuliani/DeepRL-Agents/blob/master/Q-Exploration.ipynb

Discussion on challenges

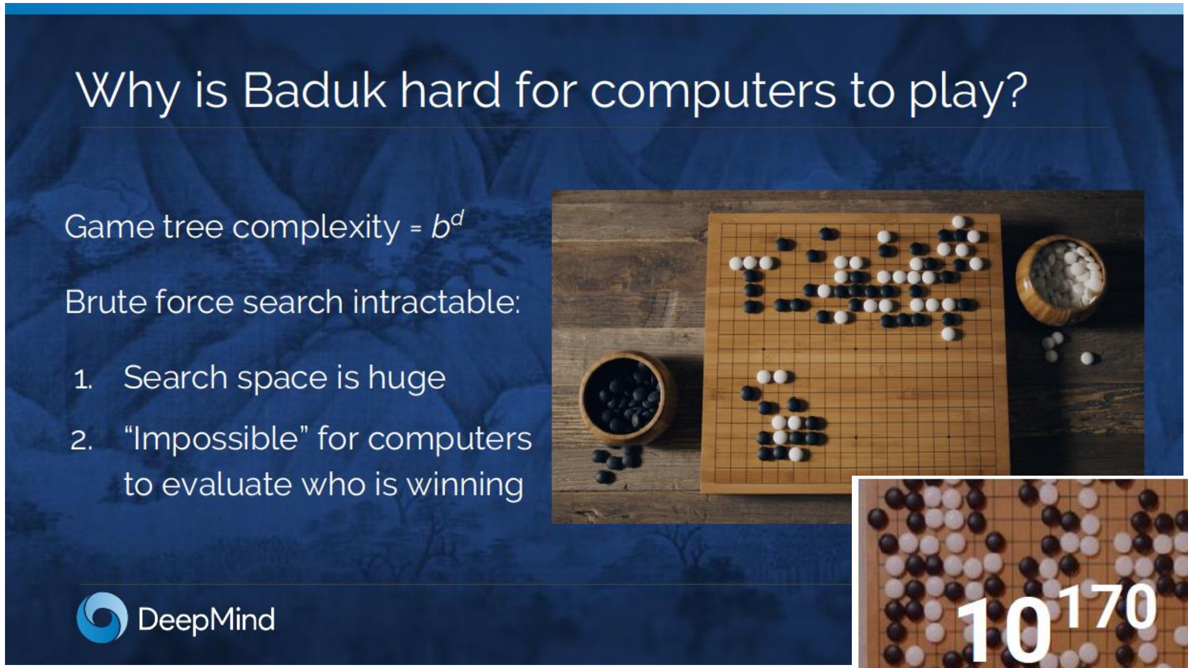
- Designing the problem domain
  - State representation
  - Action choice
  - Cost/reward signal
- Acquiring data for training
  - Exploration / exploitation
  - High cost actions
  - Time-delayed cost/reward signal
- Function approximation
- Validation / confidence measures





Alpha Go

- The game of Go
  - Number of legal moves per position b: 250
  - Game length (depth) d: 150
  - Much more than chess (b=35, d =80)
- Deal with valuated network and MC Tree Search




ARTICLE

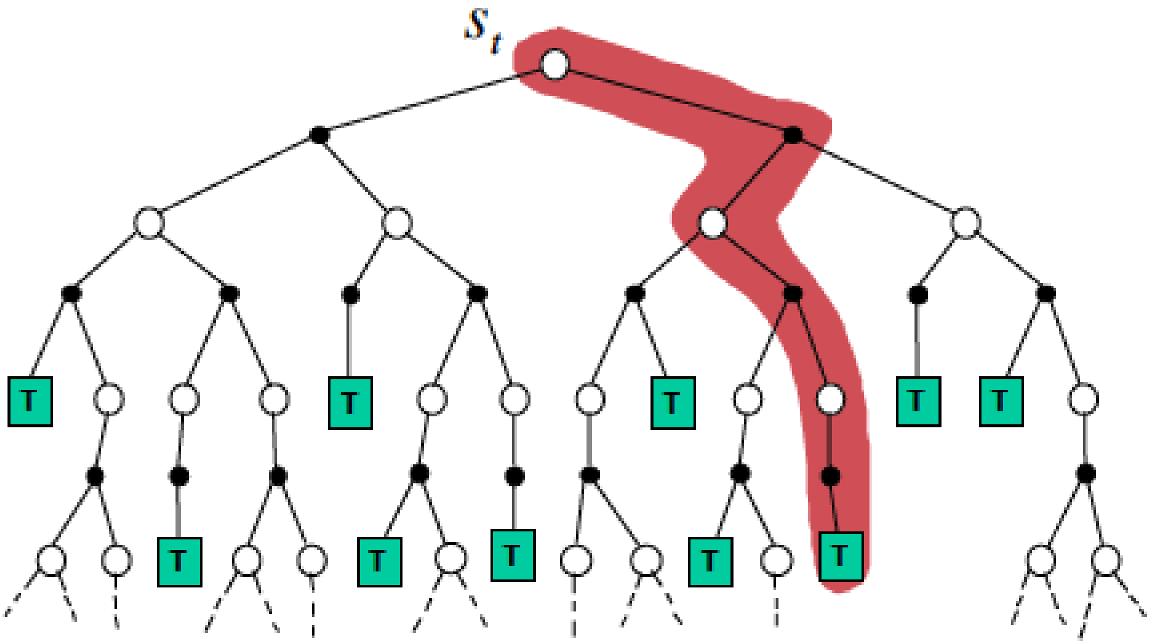
doi:10.1038/nature16961

Mastering the game of Go with deep neural networks and tree search

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

Policy search

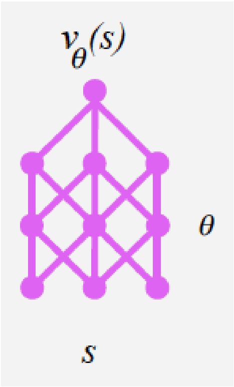
- Forward search algorithms select the best action by look ahead
  - Search tree with current sate s(t) as root
  - Using a model to look ahead
  - No need to solve the whole MDP, just sub of MDP from the current state
- Simulated-Based Search
  - Simulate episodes of experience from current state
  - Apply model-free RL to simulated episodes

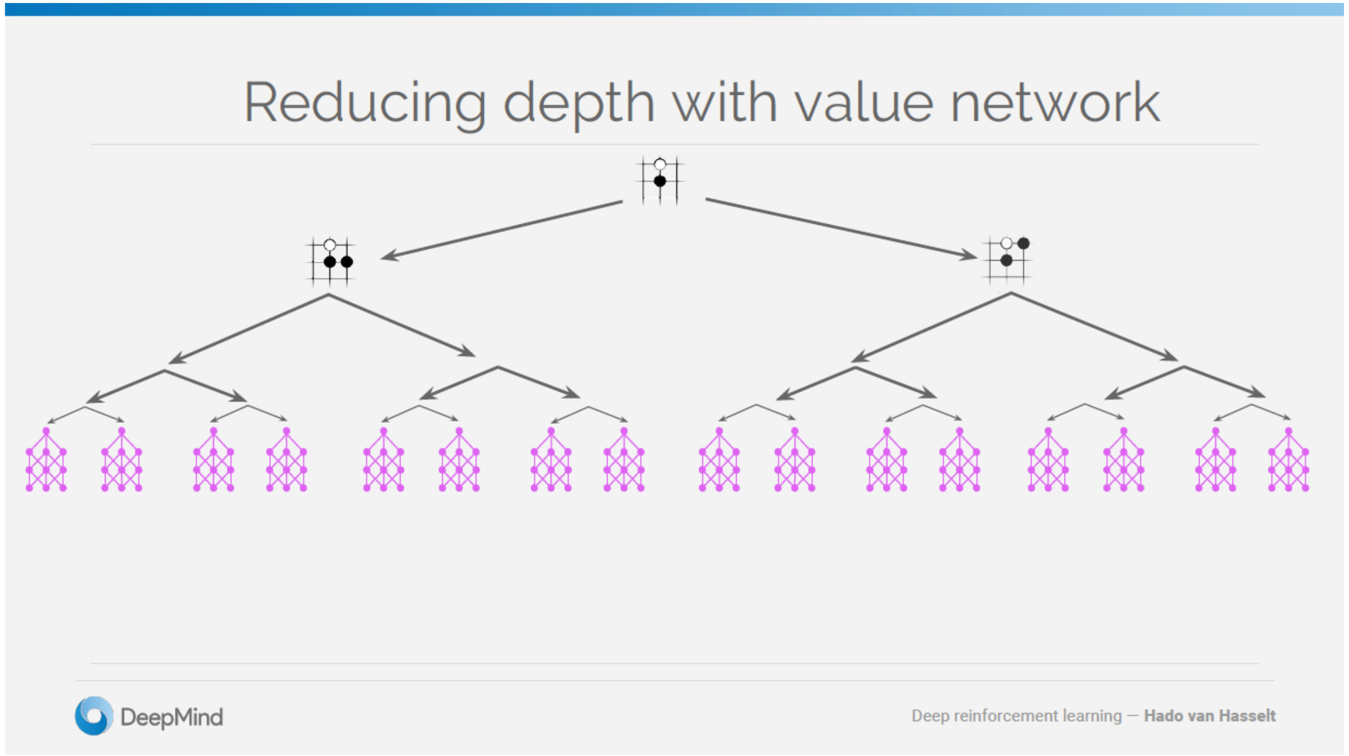


$$\{s_t^k, A_t^k, R_{t+1}^k, \ldots, S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu$$

Policy search motivation

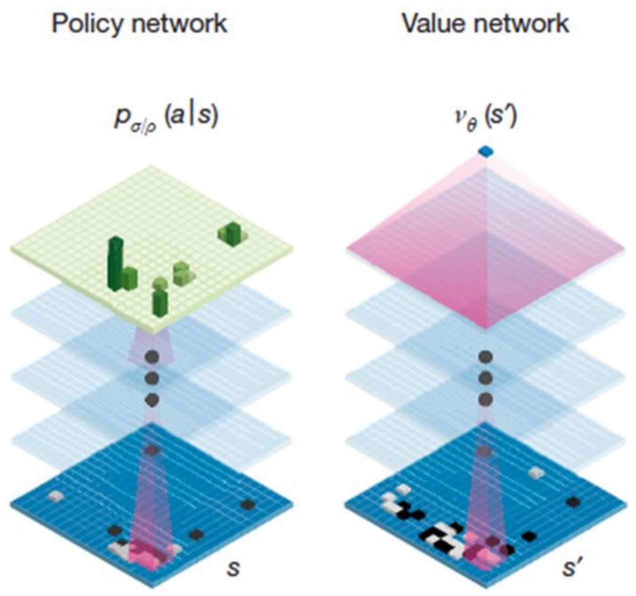
- RL for the game of Go
  - Searching for state value
  - How to reduce search space?
  - State value:

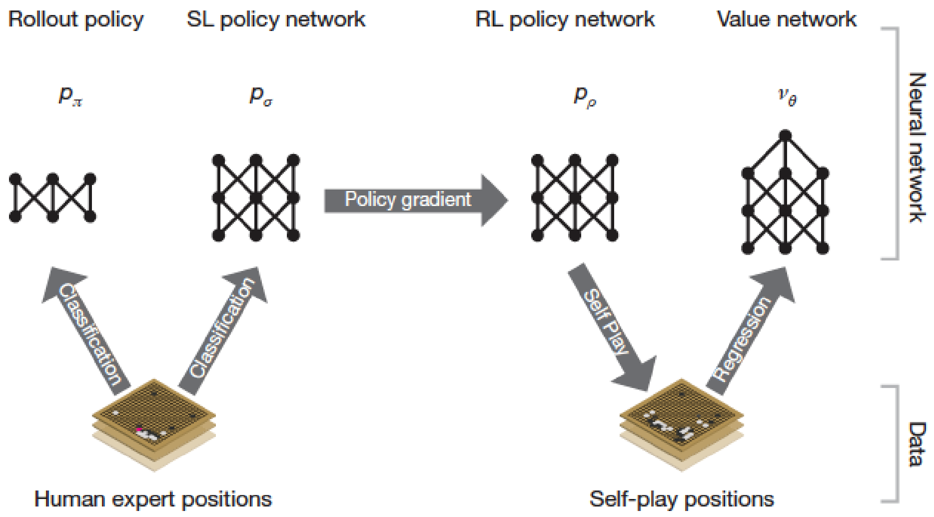




Neural network training pipeline and architecture

- Supervised train two policy network
  - Fast and less accurate p_π
  - More layers network p_σ
- Self train SL policy network to improve policy network p_ρ
- Self train value network v_θ





AlphaGo Zero (Chess)

- Self-play pipeline
  - Build search tree
  - Sample action $a_t \sim \pi_t$ using MCTS with the lates network f_θ
  - Playing at the current state
  - Update weigh vectors
  - At the end, store winner z
- Neural network training
  - Initial to random weight θ_0
  - At time t, execute network $f_{\theta_{t-1}}$ using the last iteration θ_{t-1}
  - When each episode ends, at T, score the reward $r_T = \{\pm 1\}$
  - Store winner score for any step $\{s, \pi, z\}$
  - In parallel, train θ_t using data sample from $\{s, \pi, z\}$ with loss function l