REINFORCEMENT LEARNING

Ahmed El-Khouly

CONTENTS

APPLICATION OF RL IN RECOMMENDER SYSTEMS

- Visualization recommender
  - Recommend column to slot mappings
  - Recommend effective visualizations
- Content recommendation
  - Identify primary columns from dataset
  - Recommend column combinations to unearth insights

DEFINITION

”Reinforcement learning is an area of machine learning concerned with how software **agents** ought to take **actions** in an **environment** in order to maximize some notion of **cumulative reward**.”

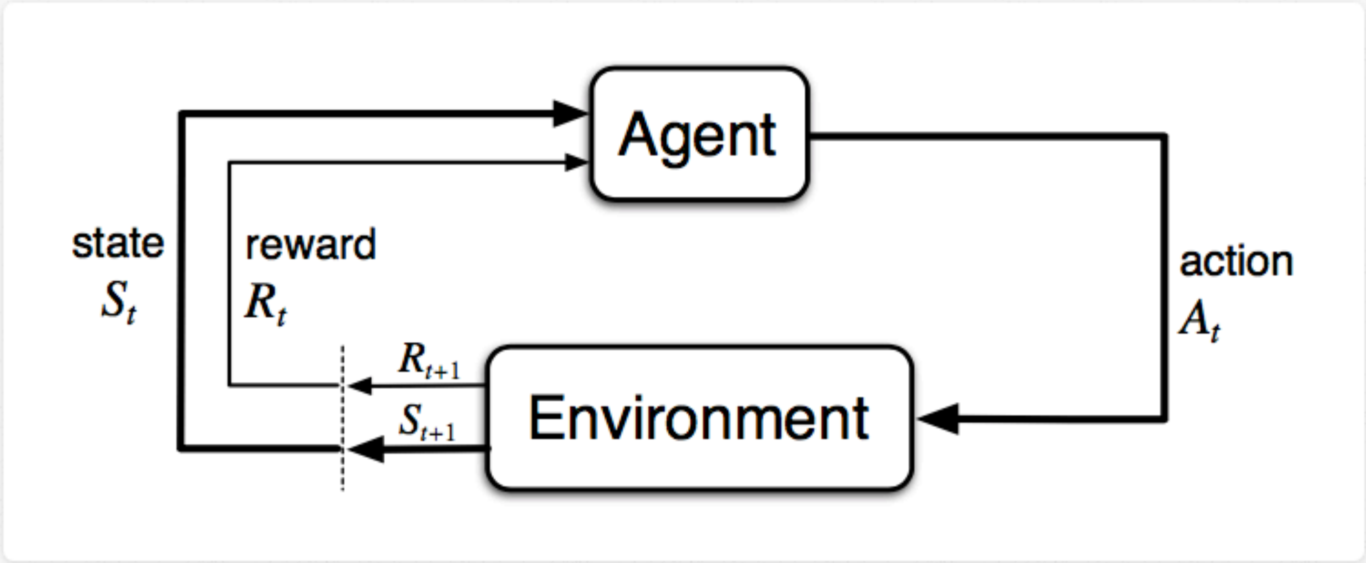
- Wikipedia

Goal of reinforcement: Agent learns to ***maximize cumulative rewards*** *by trial and error*

COMPARISON WITH OTHER ML ALGORITHMS

	Supervised ML	Unsupervised ML	Reinforcement learning
Input	labeled data	unlabeled data	State
Type	classification, regression	clustering, association	policy
Activity	passive	passive	active

FINITE MARKOV DECISION PROCESS



- Components:
  - Environment: embodiment of problem agent interacting with through time
  - Agent: entity learning actions to maximize overall rewards
  - State: representation of environment at specific time step
  - Reward: numerical signal agent receives for taking actions
  - Action: agent's decision making

MARKOV DECISION PROCESS

- Sequence of interaction between the agent and environment Action trajectory: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, \dots$
- *Probability of future states only depends on present:*

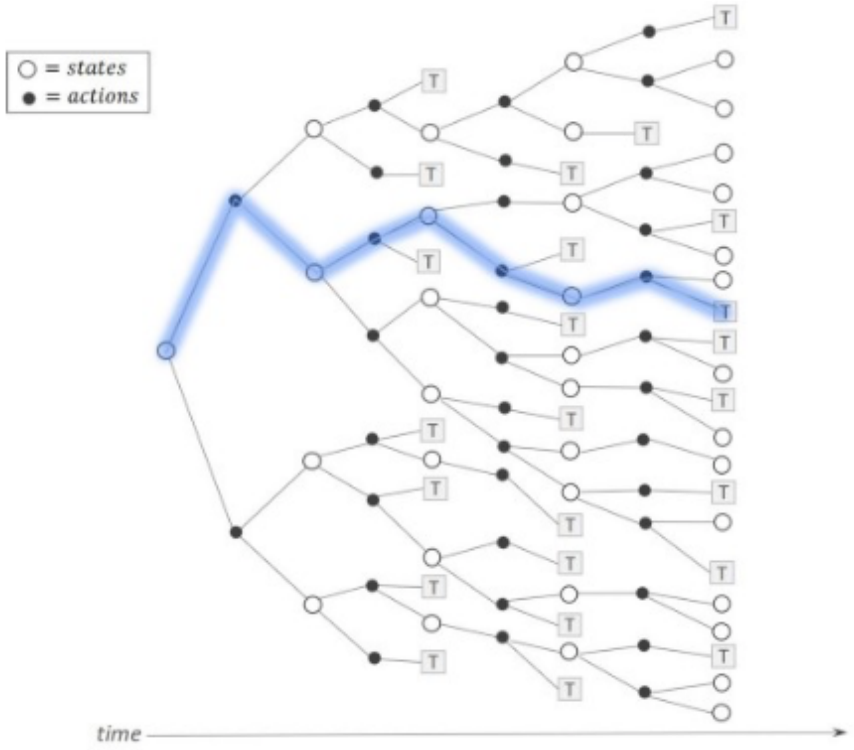
  $P(S_{t+1} | S_t) = P(S_{t+1} | S_1, S_2, S_3, \dots, S_t)$
- Cumulative rewards

  $G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T$
- Expected discounted reward: future is ***uncertain***

  $\mathbf{G_t} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$: $\gamma \in [0, 1]$ discount factor

  $= R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots)$

  $\mathbf{= R_{t+1} + \gamma\, G_{t+1}}$



Goal: **Maximize** the expected discounted reward G_t

REINFORCEMENT LEARNING FUNCTIONS

- **Policy -** how the agent is going to behave given a certain state at any time step

  $\boldsymbol{\pi(a|S_t) = p(A_{t=a}\,|S_t)\ \forall\ a\ \in A}$

- **Value -** of a state in any time step is defined by "how good" the state is in the long run

  $\boldsymbol{V_\pi(s)} = E_\pi[G_t \mid S_t = s]$

  $= \boldsymbol{E_\pi[R_{t+1} + \gamma\, G_{t+1}| \, S_t = s]}$

  $= \sum_a \pi(a|S_t)\ \sum_{s',r} p(s',r|s,a)[r + \gamma\ E_\pi[\ G_{t+1}|\ S_{t+1} = s']]$

  $= \sum_a \pi(a|S_t)\ \sum_{s',r} p(s',r|s,a)[r + \gamma\ V_\pi(s')]$

- $\boldsymbol{Q_\pi(s,a)} = \ E_\pi[G_t \mid S_t = s,\ A_t = a]$

- **Bellman equations**

  $$\boxed{\boldsymbol{V_\pi(s) = \ E_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s]}}$$

  $$\boxed{\boldsymbol{Q_\pi(s,a) = \ E_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s, A_t = a]}}$$

BELLMAN OPTIMALITY EQUATIONS

- Optimal policy is better or equal to any other policy if value function is higher than or equal to any policy for all states

$v_*(s) = max_\pi v_\pi(s)\ \forall\ s \epsilon S$

$= max_a\ q_{\pi*}(s,a)$ ---(eq.1)

$= \boldsymbol{max_a E[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]}$

- Similarly, since q_* gives expected return for taking an action a in state s and follow the optimal policy thereafter:

$q_*(s,a) = E[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$

$= \boldsymbol{E[R_{t+1} + \gamma max_{a'}\ q_*(s',a') \mid S_{t+1} = s', A_t = a]}$ --- by substitution from (eq.1)

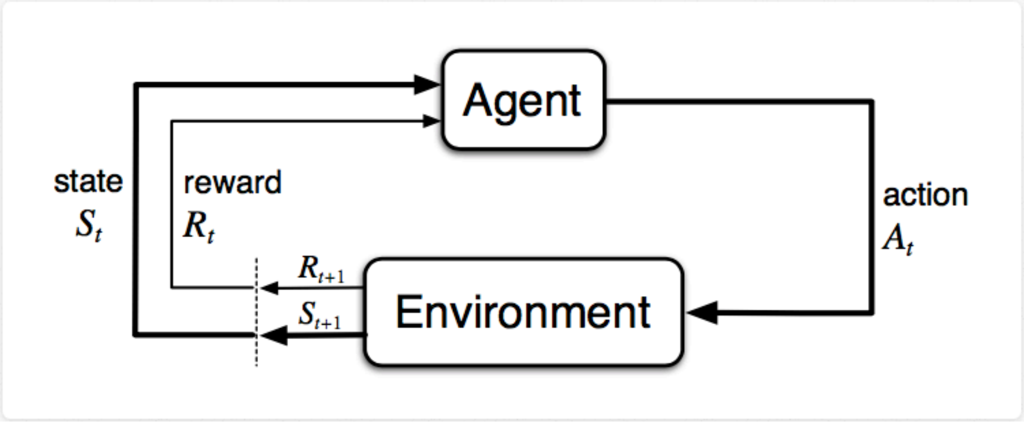
- Optimal functions are greedy!

The goal of reinforcement learning is for the agent to learn optimal functions

RECAP

- *Probability of future states only depends on present*

  $P(S_{t+1}|\, S_t) = P(S_{t+1}|\, S_1, S_2, S_3, \dots, S_t)$



- **Expected discounted reward** $\quad G_t = R_{t+1} + \gamma\, G_{t+1}$

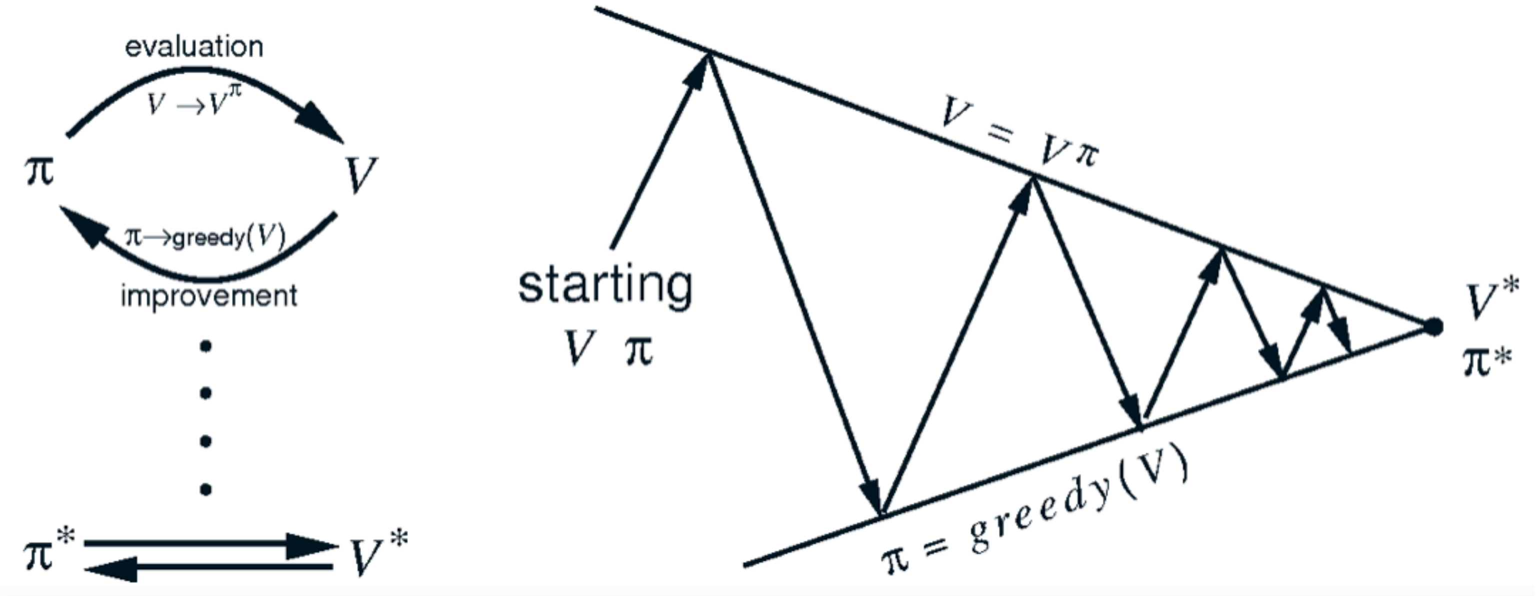
- **Bellman equations**

$$\boxed{V_\pi(s) = E_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s]}$$

$$\boxed{Q_\pi(s,a) = E_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s, A_t = a]}$$

- **Bellman Optimality equation** $\quad v_*(s) = max_a E[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$

$$q_*(s,a) = E[R_{t+1} + \gamma max_{a'} q_*(s',a') \mid S_{t+1} = s', A_t = a]$$

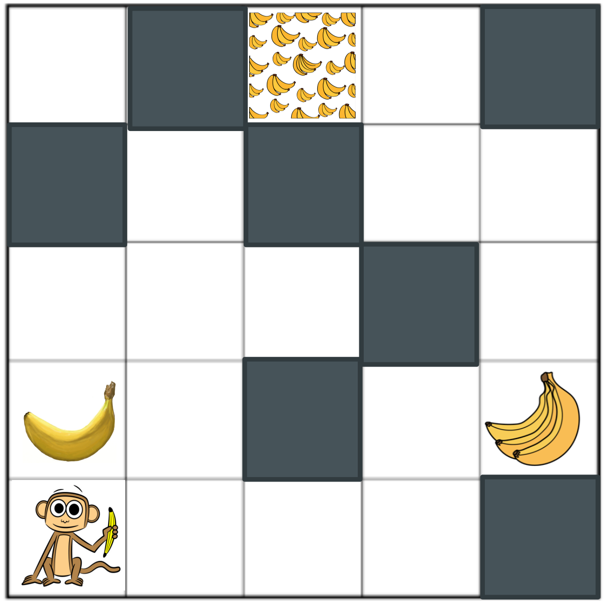


GENERALIZED POLICY ITERATION

Initialization: policy π(s) ← arbitrary policy, v(s) ← 0.

- **Repeat until convergence:**
  - **Policy evaluation:**
    - use policy π to make actions on environment collecting experiences.
    - update value function from returns obtained in experiences, driving value function closer to that corresponding to the policy π.
  - **Policy improvement:**
    - improve the policy π by making it greedy with respect value function.

LEARNING ALGORITHMS – MONTE CARLO (ON POLICY)



- **Episodic Tasks-** learn from complete episode
- **On-policy-** use same policy to generate episodes
- **Exploring starts convergence condition**
  - $\pi(a|s) > 0 \;\; \forall \, a \, \in A, \; s \in S$
  - ϵ-$Greedy$ policy (soft policy)
- **Evaluation-** V_* approximated by averaging discounted returns for every state-action pair
- **Policy improvement-** increase probability of actions yielding max expected discounted returns (greedy)

LEARNING ALGORITHMS – SARSA (TEMPORAL DIFFERENCE)

- **Temporal Difference-** can learn partial time-step subsequence
- **Continuous and episodic** tasks- does not need to wait until episodes end
- **On-policy-** use same policy to generate episodes
- **Exploring starts convergence condition**
  - $\pi(a|s) > 0 \ \ \forall\, a \ \in A, \ \mathrm{s} \in S$
  - ϵ-$Greedy$ policy (soft policy)
- **Evaluation-** learn from quintuple: S,A,R,S’,A’
- **Q-function update rule:** $q(s,a) \mathrel{+}= \ \alpha(\overbrace{\boxed{R + \gamma q(s',a') - q(s,a)}}^{\text{TD Error}})$

Note: *from bellman equation q(s, a) = R+* $\gamma q(s', a')$

LEARNING ALGORITHMS – Q-LEARNING

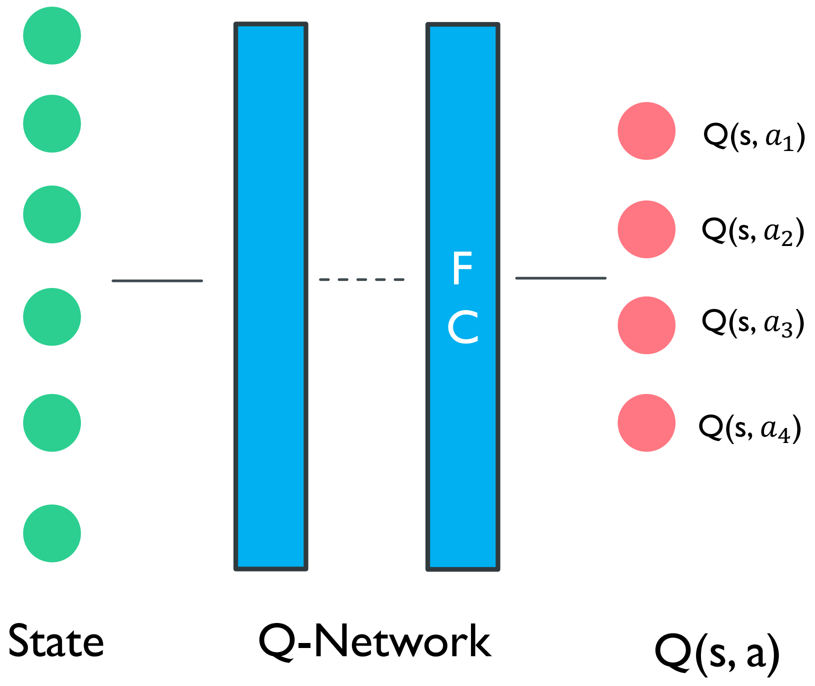
- **Value-based learning**
- **Temporal Difference-** can learn partial time-step subsequence
- **Continuous and episodic** tasks- does not need to wait until episodes end
- **Off-policy-** updates Q-value using greedy action on next state
- **Exploring starts convergence condition**
  - $\pi(a|s) > 0 \ \forall\, a \ \in A,\ s \in S$
  - ϵ-$Greedy$ policy (soft policy)
- **Q-function update rule:** $q(s,a) \mathrel{+}= \alpha(\underbrace{\boxed{R + \ max_{a'} q(s',a') - q(s,a)}}_{\text{TD Error}})$

Note: *from bellman optimality equation* $q_*(s,a) = R + \gamma\ max_{a}{}'\ q(s',a')$

LEARNING ALGORITHMS – DEEP Q-NETWORK (DQN)

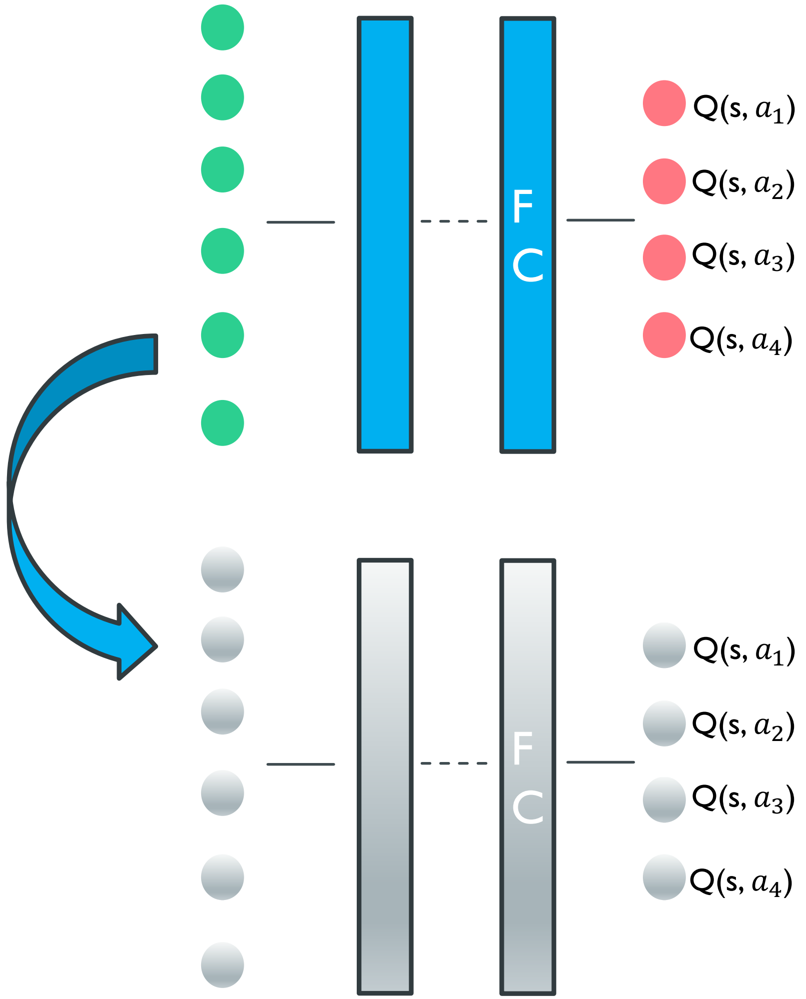
- Q-learning performance drops as the state/actions gets more sophisticated
  - Frozen Lake with 4x4 grid had 4x4x4 = 64 entries that need to be learnt
  - The method does not scale as the number of states increase
- Use Neural networks to approximate optimal Q function
  - **Input layer-** State
  - **Output-** predict the Q-value for every action
  - **Q-Target-** $Q_{target} = R + max_a q(s', a)$
  - **GD Update rule-** $\Delta w = \alpha[(R + \gamma max_a q(s', a) - q(s, a)]\Delta q(s, a)$

  Note: *from bellman optimality equation* $q_*(S,A) = R + \gamma \, max_a \, Q(s', a')$
- **Experience replay**
  - Remember old experiences
  - Reduce correlation between experiences



LEARNING ALGORITHMS – DOUBLE DEEP Q-NETWORK (DDQN)

- **Problem- moving targets**
  - Target =$R + \gamma max_a q(s', a)$ predicted using same weights
  - "network chasing its own tail" phenomenon slow training
- **Solution:** use two neural networks with identical architecture
  - Main DQN - learn approximating Q function
  - Target DQN – fixed weights network used to provide target estimates
  - Update target DQN weights from main DQN every τ time-setps
    - Soft update
    - Hard update



THANK YOU