Reinforcement learning

Learning setting

- The learner is provided a set of possible states $\mathcal{S}$, and for each state, a set of possible actions, $\mathcal{A}$ moving it to a next state.
- In performing action a from state s, the learner is provided an immediate reward $r(s, a)$.
- The task is to learn a *policy* allowing to choose for each state s the action a maximizing the overall reward (including future moves).
- The learner has to deal with problems of *delayed reward* coming from future moves, and trade-off between *exploitation* and *exploration*.
- Typical applications include moving policies for robots and sequential scheduling problems in general.

Reinforcement learning: overview

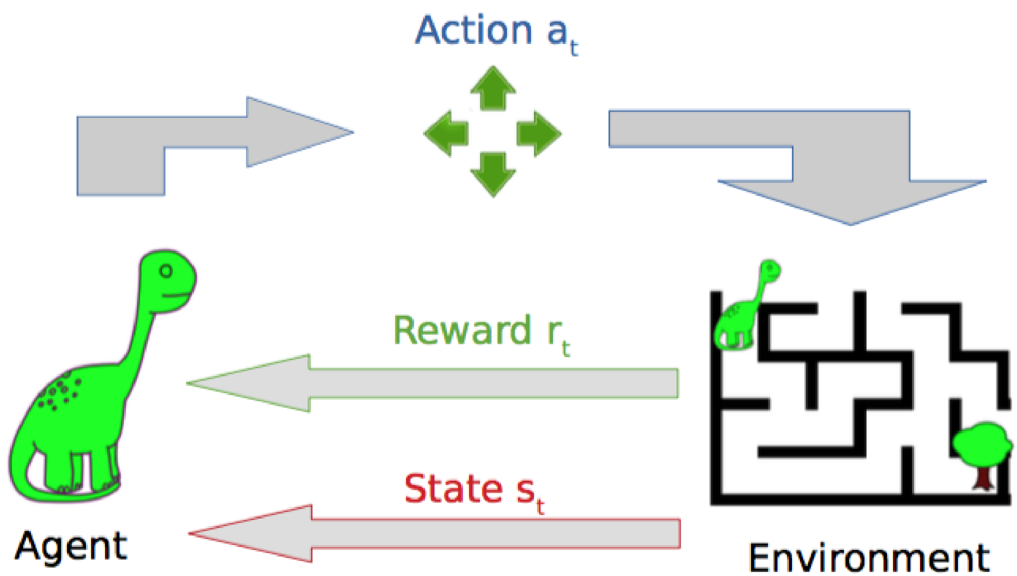


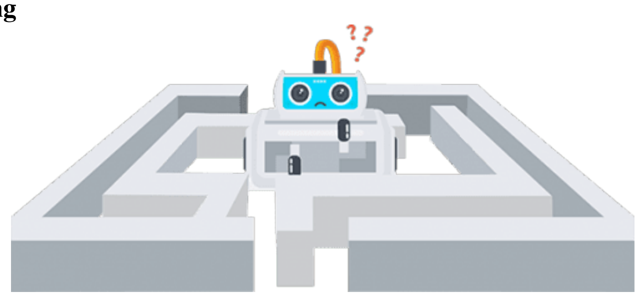
Image from Sean Devlin

Reinforcement learning: applications

Robotics

Game Playing

Sequential Decision Making



Setting

- An agent needs to take a sequence of decisions (e.g. moves in a maze)
- The agent should maximize some utility function (e.g. avoiding holes, exiting the maze)
- There is uncertainty in the result of a decision (e.g. the floor could be slippery)

Formalization

Markov Decision Process (MDP)

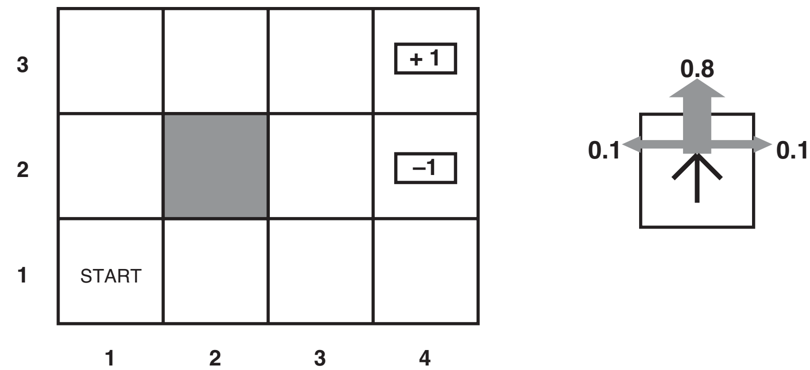
- A set of **states** $\mathcal{S}$ in which the agent can be at each time instant
- A (possibly empty) set of **terminal states** $\mathcal{S}_G \subset \mathcal{S}$
- A set of **actions** $\mathcal{A}$ the agent can make

- A **transition** model providing the probability of going to a state s' with action a from state s

$$P(s'|s,a) \quad s,s' \in \mathcal{S},\ a \in \mathcal{A}$$

- A **reward** $R(s,a,s')$ for making action a in state s and reaching state s'

MDP: Example



Agent moving in room

- **State**: occupied cell
- **Terminal states** (row,column): (4,2), (4,3)
- **Actions**: UP,DOWN,LEFT,RIGHT
- **Transitions probabilities**: 0.8 in direction of action, 0.1 in each orthogonal direction (see figure)
- **Rewards**: R((4,2)) = -1, R((4,3)) = +1, all other rewards = r

Image from Russell & Norvig, 2010

Defining Utilities

Utilities over time

- An **environment history** is a sequence of states
- Utilities are defined over environment histories
- We assume an **infinite horizon** (no constraint on the number of time steps)
- We assume **stationary** preferences (if one history is preferred to another at time t, the same should hold at time t' provided they start from the same state)

Defining Utilities

Utilities over time

Two sensible ways to define utilities under previous conditions

- **Additive rewards**

$$U([s_0, s_1, s_2, \dots]) = R(s_0) + R(s_1) + R(s_2) + \cdots$$

- **Discounted rewards**

$$U([s_0, s_1, s_2, \dots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$$

  for $\gamma \in [0, 1]$

Note

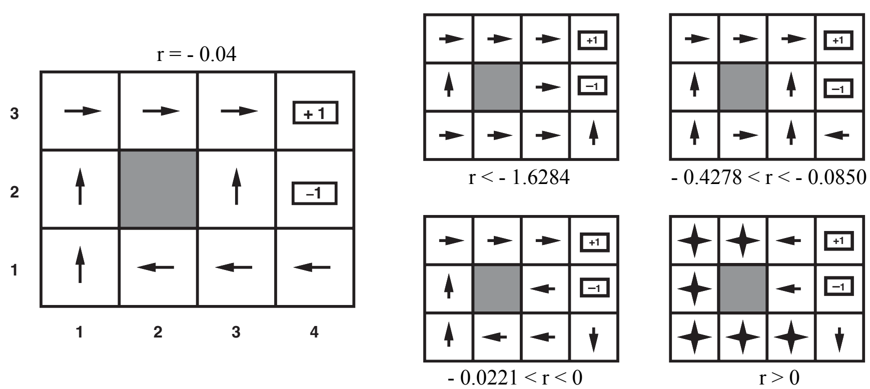
We consider rewards that only depend on the (destination) state. In the more general case each reward should be written as $R(s_t, a_t, s_{t+1})$.

MDP: taking decisions

Optimal Policy

- A **policy** π is a full specification of what action to take at each state.
- The **expected utility** of a policy is the utility of an environment history, taken in expectation over all possible histories generated with that policy
- An **optimal policy** π^* is a policy maximizing expected utility
- For infinite horizons, optimal policies are **stationary**, i.e. they only depend on the current state

Optimal policy: examples

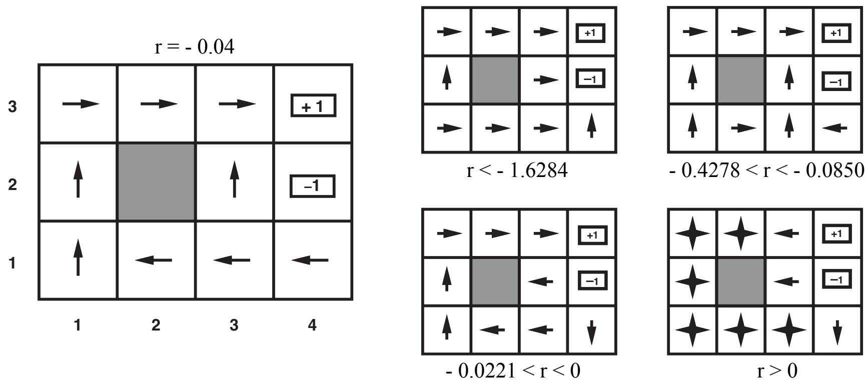


Optimal policies varying r

- utility is made with additive rewards
- r is the reward of non-terminal states
- Arrows indicate the best action to take
- Star indicates all actions are equally optimal

Image from Russell & Norvig, 2010

Optimal policy: examples



Discussion

- If moving is very expensive, optimal policy is to reach any terminal state asap
- If moving is very cheap, optimal policy is avioding the bad terminal state at all costs
- If moving gives positive reward, optimal policy is to stay away of terminal states!! (usefulness of discounted rewards)

Optimal policy: utilities

Utility of states

- The utility of a state given a policy π is:

$$U^{\pi}(s) = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^t R(S_{t+k+1}) \middle| S_t = s\right]$$

  where S_{t+k+1} is the state reached after k steps using policy π starting from $S_t = s$.

- The true utility of a state is its utility under an optimal policy:

$$U(s) = U^{\pi^*}(s)$$

- Given the true utility, an optimal policy is as follows:

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a)U(s')$$

Computing an optimal policy

The utility of a state is its immediate reward plus the expected discounted utility of the next state, assuming that the agent chooses and optimal action

Bellman equation

$$U(s) = R(s) + \gamma * \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a)U(s')$$

- There is a Bellman equation for each state $s \in \mathcal{S}$
- Utilities of states are solutions of the set of Bellman equations
- The solutions to the set of Bellman equations are *unique*
- Directly solving the set of equations is hard (non-linearities because of the $\max$)

Computing an optimal policy
Value iteration

1. Initialize $U_0(s)$ to zero for all s
2. Repeat
   (a) do Bellman update for each state s:
   $$U_{i+1}(s) \leftarrow R(s) + \gamma * \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a) U_i(s')$$
   (b) $i \leftarrow i + 1$
3. Until *max utility difference below a threshold*
4. return U

Optimal policy
The optimal policy can be set as:

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a) U(s')$$

Computing an optimal policy

Policy iteration

1. Initialize π_0 randomly
2. Repeat
   (a) **policy evaluation**, solve set of linear equations:
   $$U_i(s) = R(s) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, \pi_i(s)) U_i(s') \quad \forall s \in \mathcal{S}$$
   where $\pi_i(s)$ is the action that policy π_i prescribes for state s.
   (b) **policy improvement**
   $$\pi_{i+1}(s) \leftarrow \operatorname*{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s,a) U_i(s') \quad \forall s \in \mathcal{S}$$
   (c) $i \leftarrow i + 1$
3. Until *no policy improvement*
4. return π

Reinforcement learning

Dealing with partial knowledge

- Value iteration and policy iteration assume perfect knowledge (environment, transition model,rewards)
- In most cases, some of these aspects are not known
- Reinforcement learning aims at **learning policies** by space exploration
- **policy evaluation**: policy is given, environment is learned (passive agent)
- **policy improvement**: both policy and environment are learned (active agent)

Policy evaluation in unknown environment

Adaptive Dynamic Programming (ADP): algorithm

1. Loop
   (a) Initialize s
   (b) Repeat
      i. Receive reward r, set $R(s) = r$
      ii. Choose next action $a \leftarrow \pi(s)$
      iii. Take action a, reach step s'
      iv. Update counts
      $$N_{sa} \leftarrow N_{sa} + 1; \quad N_{s'|sa} \leftarrow N_{s'|sa} + 1$$
      v. Update transition model
      $$p(s''|s,a) \leftarrow N_{s''|sa}/N_{sa} \quad \forall s'' \in \mathcal{S}$$
      vi. Update utility estimate
      $$U \leftarrow \textsc{PolicyEvaluation}(\pi, U, p, R, \gamma)$$
   (c) Until s is terminal

Policy evaluation in unknown environment

ADP: characteristics

- The algorithm performs **maximum likelihood** estimation of transition probabilities
- Upon updating the transition model, it calls **standard policy evaluation** to update the utility estimate (U is initially empty)
- Each step is **expensive** as it runs policy evaluation

Policy evaluation in unknown environment

Temporal-difference (TD) policy evaluation: rationale

- Avoid running policy evaluation at each iteration
- Locally update utility.
- If transition from s to s' is observed:
  - If s' was *always* the successor of s, the utility of s should be

$$U(s) = R(s) + \gamma U(s')$$

  - The temporal-difference update rule updates the utility to get closer to that situation:

$$U(s) \leftarrow U(s) + \alpha(R(s) + \gamma U(s') - U(s))$$

    where α is a learning rate (possibly decreasing over time)

Policy evaluation in unknown environment

TD policy evaluation: algorithm

1. Loop
   (a) Initialize s
   (b) Repeat
      i. Receive reward r
      ii. Choose next action $a \leftarrow \pi(s)$
      iii. Take action a, reach step s'
      iv. Update local utility estimate

$$U(s) \leftarrow U(s) + \alpha(r + \gamma U(s') - U(s))$$

   (c) Until s is terminal

Policy evaluation in unknown environment

TD policy evaluation: characteristics

- **No need** for a transition model for utility update
- Each **step is much faster** than ADP
- Same as ADP on the long run
- Takes **longer to converge**
- Can be seen as a rough efficient approximation of ADP

Policy learning in unknown environment

Setting

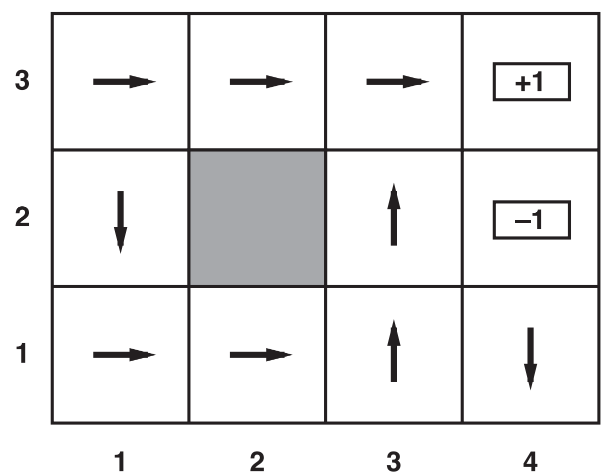
- policy learning requires combining learning the environment and learning the optimal policy for the environment
- A simple option consists of replacing policy evaluation in ADP with optimal policy computation (given current knowledge of the environment, **greedy agent**):

$$U(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p(s'|s, a) U(s')$$

Problem

The knowledge of the environment is **incomplete**. A greedy agent usually learns a suboptimal policy (lack of **exploration**).

Suboptimal policy: example



Discussion

- The algorithm finds a policy reaching the +1 terminal state along the lower route (2,1), (3,1), (3,2), and (3,3)
- It never learns the utilities of the other states
- It fails to discover the optimal route (1,2), (1,3), and (2,3).

Learning optimal policies

Exploration-exploitation trade-off

- **Exploitation** consists in following promising directions given current knowledge
- **Exploration** consists in trying novel directions looking for better (unknown) alternatives
- A reasonable trade-off should be used in defining the search scheme:
  - ϵ-greedy strategy: choose a random move with probability ϵ, be greedy otherwise

– assign higher utility estimates to (relatively) unexplored state-action pairs:

$$U^+(s) = R(s) + \gamma \max_{a\in\mathcal{A}} f\left(\sum_{s'\in\mathcal{S}} p(s'|s,a)U^+(s'), N_{sa}\right)$$

with f increasing over the first argument and decreasing over the second.

Learning optimal policies

TD learning: learning utilities of actions

- TD policy evaluation can also be adapted to learn an optimal policy
- If TD is used to learn a state utility function, it needs to estimate a transition model to derive a policy
- TD can instead be applied to learn an action utility function $Q(s, a)$
- The optimal policy corresponds to:

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

Learning optimal policies

SARSA: on-policy TD learning

1. Loop
   (a) Initialize s
   (b) Repeat
      i. Receive reward r
      ii. Choose next action $a \leftarrow \pi^\epsilon(s)$
      iii. Take action a, reach step s'
      iv. Choose action $a' \leftarrow \pi^\epsilon(s')$
      v. Update local utility estimate

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$$

   (c) Until s is terminal

Note
π^ϵ is an ϵ-greedy (or some other form of non-greedy) policy based on Q.

Learning optimal policies

Q-learning: off-policy TD learning

1. Loop
   (a) Initialize s
   (b) Repeat
      i. Receive reward r
      ii. Choose next action $a \leftarrow \pi^\epsilon(s)$

iii. Take action a, reach step s'

iv. ~~Choose action $a' \leftarrow \pi^{\epsilon}(s')$~~

v. Update local utility estimate

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a))$$

(c) Until s is terminal

Learning optimal policies

SARSA vs Q-learning

- SARSA is **on-policy**: it updates Q using the **current policy**'s action
- Q-learning is **off-policy**: it updates Q using the **greedy policy**'s action (which is NOT the policy it uses to search)
- Off-policy methods are more flexible: they can even learn from traces generated with an unknown policy
- On-policy methods tend to converge faster, and are easier to use for continuous-state spaces and linear function approximators (see following slides)

Scaling to large state spaces

Function approximation

- All techniques seen so far assume a tabular representation of utility functions (of states or actions)
- Tabular representations do not scale to large state spaces (e.g. Backgammon has an order of 10^{20} states)
- The solution is to rely on **function approximation**: approximate $U(s)$ or $Q(s,a)$ with a parameterized function.
- The function takes a state representation as input (e.g. x,y coordinates for the maze)
- The function allows to **generalize** to unseen states

Example: State utility function approximation

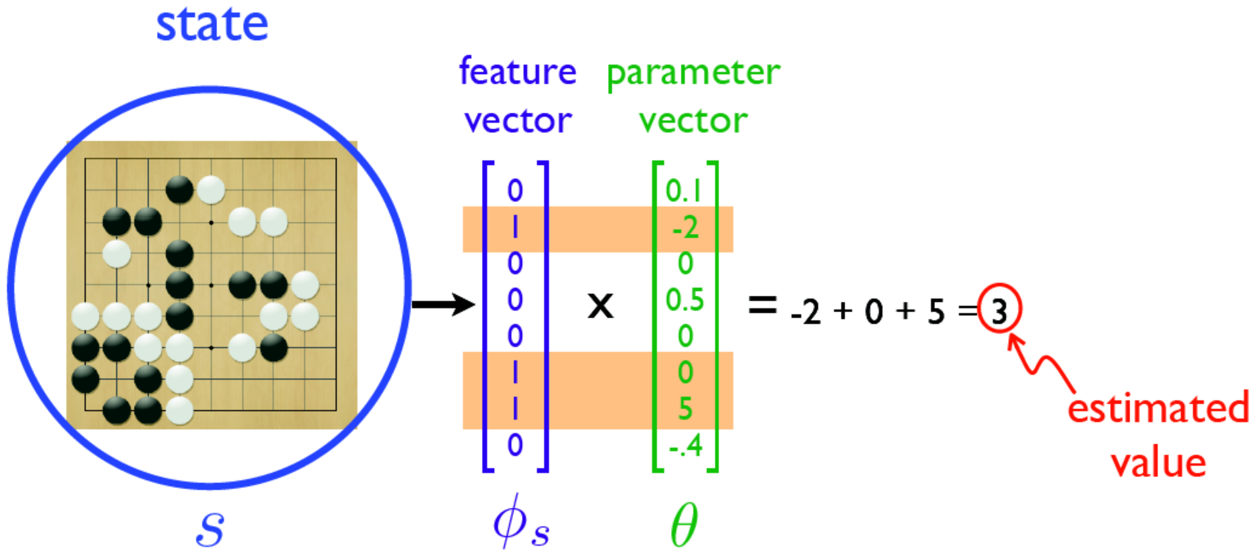


Image from Ngo Anh Vien's lectures

Example: Action utility function approximation

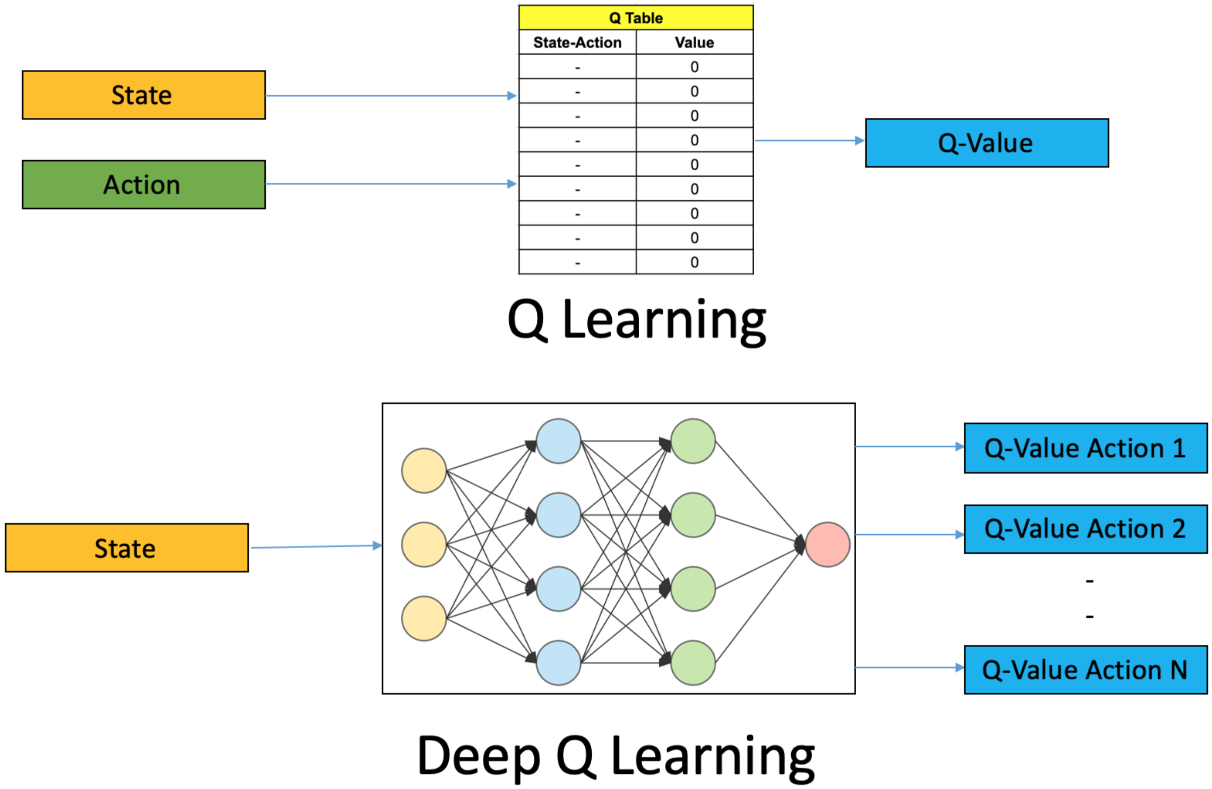


Image from Praphul Sing's blog

Learning the approximation function

TD learning: state utility

- TD error

$$E(s, s') = \frac{1}{2}(R(s) + \gamma U_\theta(s') - U_\theta(s))^2$$

- Error gradient wrt function parameters

$$\nabla_\theta E(s, s') = (R(s) + \gamma U_\theta(s') - U_\theta(s))(-\nabla_\theta U_\theta(s))$$

- Stochastic gradient update rule

$$\begin{aligned} \theta &= \theta - \alpha \nabla_\theta E(s, s') \\ &= \theta + \alpha(R(s) + \gamma U_\theta(s') - U_\theta(s))(\nabla_\theta U_\theta(s)) \end{aligned}$$

Learning the approximation function

TD learning: action utility (Q-learning)

- TD error

$$E((s,a),s') = \frac{1}{2}(R(s) + \gamma \max_{a' \in \mathcal{A}} Q_\theta(s',a') - Q_\theta(s,a))^2$$

- Error gradient wrt function parameters

$$\begin{aligned}\nabla_\theta E((s,a),s') &= (R(s) + \gamma \max_{a' \in \mathcal{A}} Q_\theta(s',a') - Q_\theta(s,a)) \\ &\qquad (-\nabla_\theta Q_\theta(s,a))\end{aligned}$$

- Stochastic gradient update rule

$$\begin{aligned}\theta &= \theta - \alpha \nabla_\theta E((s,a),s') \\ &= \theta + \alpha(R(s) + \gamma \max_{a' \in \mathcal{A}} Q_\theta(s',a') - Q_\theta(s,a))(\nabla_\theta Q_\theta(s,a))\end{aligned}$$

Bibliography

- Russell, S. J., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach (3rd edition)*. Prentice Hall. Chapters 17 and 21.
- Sutton, R. S. & Barto, A. G. (2018). *Reinforcement Learning: an Introduction (2nd edition)*, The MIT PRESS.