

【强化学习的算法对比】

DQN：离散化的低维动作空间

DPPG：深度确定性策略梯度算法，可以用来解决连续的动作空间上的深度强化学习问题

Q-learning：离散，低维的动作空间

1、强化学习基本算法

- 马尔科夫决策过程
- 策略迭代
- 价值迭代
- 泛化迭代

2、基于值函数的强化学习方法

- 基于蒙特卡罗方法强化学习方法
- 基于时间差分的强化学习方法
- 基于值函数的强化学习方法（DQN,Q-learning,Double Q_Learning）

3、基于直接策略搜索的强化学习方法

- 基于策略梯度的强化学习方法(Actor-Critic,A3C,)
- 基于置信域策略的强化学习方法(TRPO)
- 基于确定性策略的强化学习方法
- 基于引导策略搜索的强化学习方法(ADMM)

4、强化学习研究及前言

- 逆向强化学习
- 组合策略梯度和值函数方法
- 值函数网络
- 基于模型的强化学习方法：PILCO及其扩展

<https://blog.csdn.net/AMDS123/article/details/70197796>总结的不错。

强化学习算法是机器学习大家族中的一类，使用强化学习能够让机器学着如何在环境中拿到高分表现出优秀的成绩，而这些成绩背后所付出的努力，是不断的试错，不断尝试，累加经验，学习经验。

强化学习是一个大家族，包括许多中算法，比如通过行为的价值来选去特定行为的方法，包括使用表格学习的q learning, sarsa, 使用神经网络学习的deep q network, 还有直接输出行为的policy gradients, 有或者了解所处的环境，想象出一个虚拟的环境病虫虚拟环境中学习等。

强化学习算法包括 Q-learning、sarsa、deep Q Network、policy Gradient、Actor Critic等等

当前机器学习算法有三种：监督学习、无监督学习和强化学习（RL）；

RL是在尝试的过程中学习到在特定的情境下选择哪种行动可以得到最大的回报。在很多场景中，当前的行动不仅会影响当前的rewards，还会影响之后的状态和一系列的rewards。RL最重要的3个特定在于：（1）：基本是一种闭环的形式；（2）不会直接指示选择哪种行动（actions）；一系列的actions和奖励信号（reward signals）都会影响之后较长时间。

RL与有监督学习、无监督学习的比较；

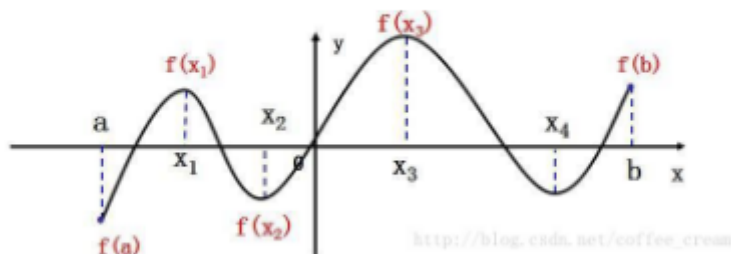
（1）有监督学习是从一个已经有标记的训练集中学习，训练集中每个样本的特征可以视为是对该situation的描述，而其label可以视为是应该执行的正确的action，但是有监督的学习不能学习交互的情景，因为在交互的问题中期望行为的样例是非常不实际的，agent智能从自己的经历(experience) 中进行学习，而experience中采取的行为并不一定是最优的，这是RL就非常合适的，因为RL不是利用正确的行为进行指导，而是利用已有的训练信息来对行为进行评价。

（2）因为RL利用的并不是采取正确行动的experience，从这点看和无监督学习的确有些相像，但是还是不一样，无监督学习的目的可以说是从一堆未标记样本中发现隐藏的结构，而RL的目的是最大化reward signal。

（3）总的来说，RL与其它机器学习算法 不同地方在于：其中没有监督者，只有一个reward信号，反馈是延迟的，不是立即生成的；时间在RL中具有重要意义；agent的行为会影响之后的一系列data。

RL采用的是边获得样例边学习的方式，在获得样例之后更新自己的模型，利用当前的模型来指导下一步的行动，下一步的行动获得reward之后再更新模型，不断迭代重复直到模型收敛。在这个过程中，非常重要的一点在于“在已有当前模型的情况下，如果选择下一步的行动才对完善当前的模型最有利”，这就涉及到了RL中的两个非常重要的概念：探索（exploration）和开发（exploitation），exploration是指选择之前未执行过的actions，从而探索更多的可能性；exploitation是指选择已执行过的actions，从而对已知的actions的模型进行完善。

举一个简单的例子， $f(x)$ 在一个未知的 $[a,b]$ 的连续函数，现在让你选择一个 x 使得 $f(x)$ 取最大值，规则是你可以通过自己给定的 x 来查看其所对应的 $f(x)$ ，假如通过 $[a,0]$ 之间的几次尝试，你会发现 x_1 附近的时候值较大，于是你想通过在 x_1 附近不断的尝试和逼近来寻找这个可能的最大值，这个就成为开发（exploitation），但是在 $[0,b]$ 之间是个未探索过的领域，这是选择这一部分就称为是exploration，如果不尽兴exploration也许只会找到局部最优的极值。exploration和exploitation在RL中同等重要，如何在探索（exploration）和开发（exploitation）之间权衡是RL中的一个重要问题和挑战。



在RL中，agents是具有明确的目标的，所有的agents都能感知自己的环境，并根据目标来指导自己的行为，因此RL的另一个特点是他将agents和与其交互的不确定的环境为是一个完整的问题，在RL中有四个非常重要的概念：

（1）规则（policy）

policy定义了agents在特定的时间特定的环境下的行为方式，可以视为是从环境状态到行为的映射；

(1) 规则 (policy)

Policy定义了agents在特定的时间特定的环境下的行为方式，可以视为是从环境状态到行为的映射，常用 π 来表示。policy可以分为两类：

确定性的policy (Deterministic policy) : $a = \pi(s)$

随机性的policy (Stochastic policy) : $\pi(a|s) = P[A_t = a|S_t = s]$

其中， t 是时间点， $t = 0, 1, 2, 3, \dots$

$S_t \in S$, S 是环境状态的集合， S_t 代表时刻 t 的状态， s 代表其中某个特定的状态；

$A_t \in A(S_t)$, $A(S_t)$ 是在状态 S_t 下的actions的集合， A_t 代表时刻 t 的行为， a 代表其中某个特定的行为。

(2) 奖励信号 (a reward signal)

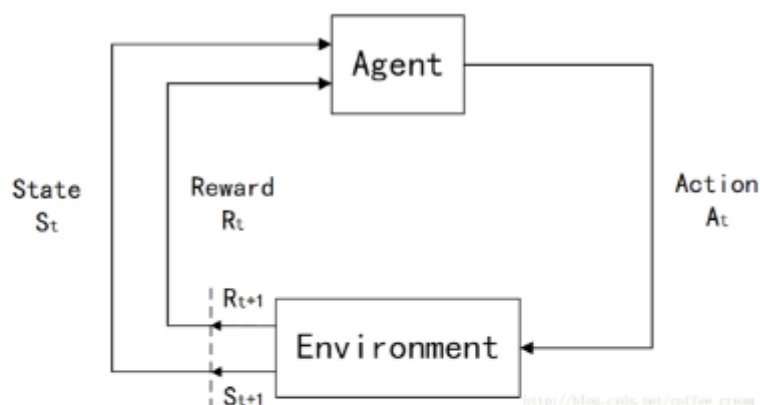
Reward就是一个标量值，是每个time step中环境根据agent的行为返回给agent的信号，reward定义了在该情景下执行该行为的好坏，agent可以根据reward来调整自己的policy。常用 R 来表示。

(3) 值函数 (value function)

Reward定义的是立即的收益，而value function定义的是长期的收益，它可以看作是累计的reward，常用 V 来表示。

(4) 环境模型 (a model of the environment)

整个Agent和Environment交互的过程可以用下图来表示：



其中， t 是时间点， $t = 0, 1, 2, 3, \dots$

$S_t \in S$, S 是环境状态的集合；

$A_t \in A(S_t)$, $A(S_t)$ 是在状态 S_t 下的actions的集合；

$R_t \in R \in \mathbb{R}$ 是数值型的reward。

一、强化学习解决什么问题？

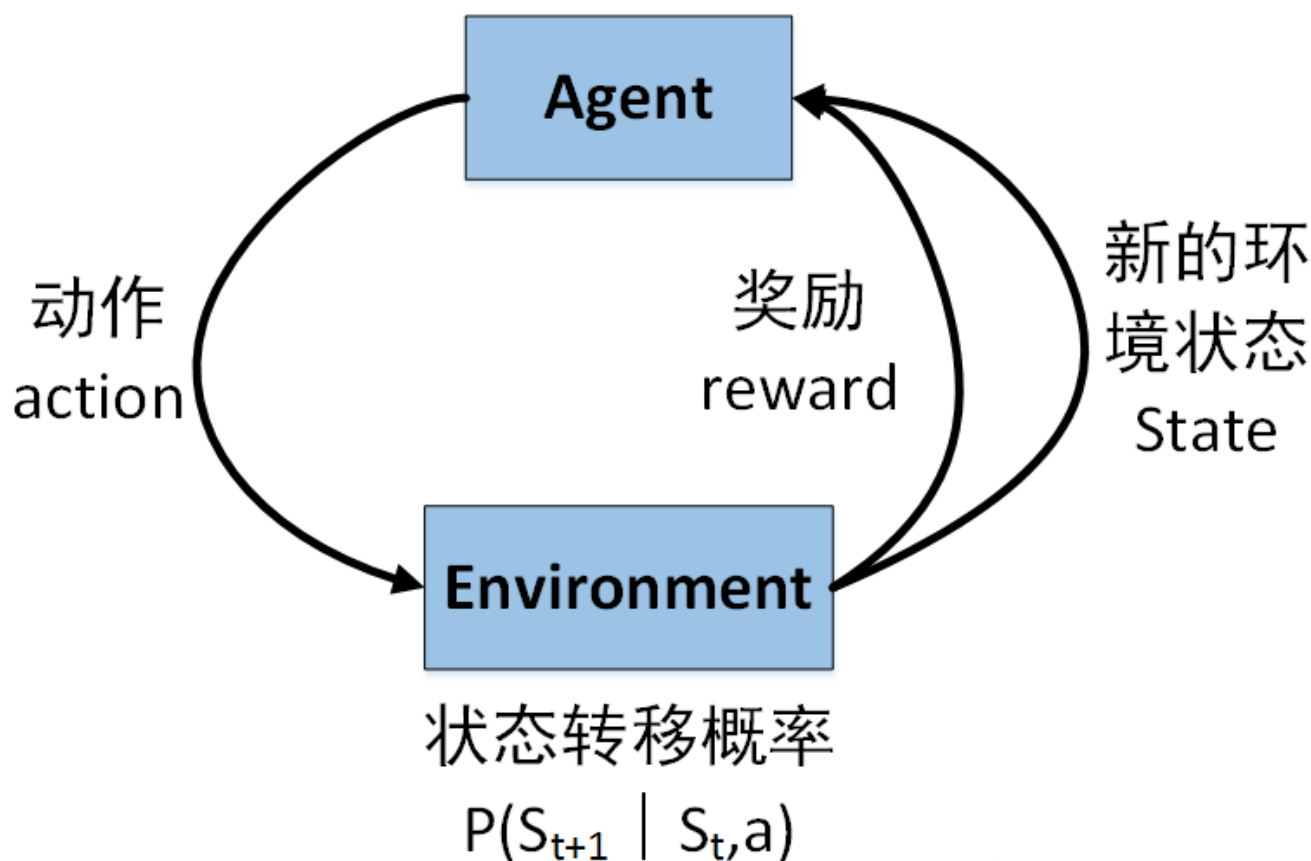
强化学习的经典应用案例有：非线性二级摆系统（非线性控制问题）、棋类游戏、机器人学习站立和走路、无人驾驶、机器翻译、人机对话 等。概括来说，强化学习所能解决的问题为**序贯决策问题**，就是需要连续不断做出决策，才能实现最终目标的问题。强化学习与其它的机器学习方法相比，专注于从交互中进行以目标为导向的学习。

二、强化学习如何解决问题？

强化学习主要是学习What to do（在当前情况下该如何选择动作），就是如何将情况（situations）与动作（actions）对应起来，以达到最大化的数值奖励信号。

2.1、强化学习的基本框架

强化学习的基本框架如下图所示，强化学习的主体一般被称为agent，其与environment互动以达到某种目的。所有强化学习的agent有明确的目标，能够感知全部或者部分environment，并且选择action执行来影响environment。除此之外，还要假设在最开始，agent需要在对于所面对的environment一无所知的情况下采取行动。



https://blog.csdn.net/Luqiang_Shi

强化学习的三个特征：（1）强化学习是一个闭环问题。（2）没有直接对该如何选择action的指示，需要试探搜索去发现哪个动作会产生最大的数值奖励。（3）动作不光会影响直接的奖励，还会影响接下来的环境状态。

2.2、强化学习系统的要素

一个强化学习系统一般包括如下的几个要素：

（1）policy（策略）：

policy是从感知到的environment的state到在这些state下要执行的action的映射。这对应于心理学中stimulus-response rules or associations（应激-反应规则或关联）。在一些情况下,policy可以是简单的函数或者是查表。而在别的情况下，可能会需要更多的计算，例如暴力搜索。policy是强化学习agent的核心，因为一个policy便可以单独决定agent的行为。另外，policy也可能是随机的。

（2）reward signal（奖励信号）：

reward signal定义了强化学习问题的目标（goal）。在每个时间步内，environment给agent发送一个奖励值。agent唯一的目标便是在长时间运行中最大化接收到的总的reward。reward signal定义了一件事对于

agent来说是好是坏。任一时刻发送给agent的reward取决于agent当前的action以及当前environment的state。agent影响reward signal的唯一方式就是通过action直接影响reward，或者通过改变environment的state间接影响。reward signal是改变policy的首要基础。如果一个policy选择了低reward的action，则之后在同样情况下，policy可能会改为选其他的action。总的来说，reward signal可能是environment state和采取的action的函数（也可以带有随机性）。

(3) value function (值函数)：

reward signal表示的是在直接感受下哪个是好的。而value function则是表示从长期来看，什么是好的。也就是state的value（价值）是一个agent从当前state开始，在未来期望累积的总的reward。reward定义的是环境state直接的好处，而value是state长期的好处，考虑了之后可能会出现的状态以及这些state下的reward。例如一个state可能会产生一个低的reward，但是有着高的value，因为跟着它的通常是一些会产生高reward的state。反之亦然。

reward是首要的，而value是其次的，因为没有reward就没有value。但是当我们做决策并对决策进行评估时，更关注的是value。对于action的选择是基于value来判断的。我们追寻的是带来最高value的state而不是最高的reward，因为这些action从长远上来讲会获得最多的reward。但不幸的是，决定value要比决定reward难得多。reward是由environment直接给出的，但是value是需要对agent的整个执行时间内的状况进行观察，以此对value进行估计和重估计。事实上，强化学习算法中最重要的部分就是如何有效地估计value。

(4) model of the environment (环境模型)：

model of the environment是用来模拟真实environment的行径的，或者说是environment会如何表现的推断。例如，给定一个state和action，这个model会预测因此产生的下一state和下一reward。model一般用在planning（规划）里，可以在真正执行之前通过考虑未来的可能情况来决定一系列动作。利用model和planning用来解决强化学习问题的方法被称为model-based方法，作为更简单的model-free方法的对照。model-free方法主要是指通过trial-and-error（试错）学习，是planning的对立。（就是planning不需要直接执行action与真实的environment互动，可以对environment建模，然后自己跟这个environment模型互动。这样就叫planning）

2.3、强化学习与监督学习的区别

监督学习需要的是标签数据，通过训练获得模型来推断那些不在训练集中的样本的标签应该是什么，监督学习的主要目的是总结概括。强化学习需要的是带有回报的交互数据，agent需要通过从自己的经验中学习如何使自己的利益最大化。

2.4、强化学习与非监督学习的区别

非监督学习一般是寻找未标注数据中同一类的结构信息，而强化学习专注于为agent实现reward最大化。

三、强化学习实例

3.1、训练Tic-Tac-Toe游戏玩家的强化学习理解

Tic-Tac-Toe（井字棋）游戏：两名玩家轮流在一个三乘三的棋盘上比赛。一个玩家画X和另一个玩家画O，直到一个玩家通过在水平，垂直或对角线上连续放置三个标记来获胜（下图表示画X的玩家获胜）。如果整个表都被填满了而没有人获胜，那么比赛就是平局。由于一个比较熟练的玩家可以保证不败，我们假设我们的对手并不是一个完美的玩家，有时他会下错地方。我们认为输和平局对我们来说是一样坏的，我们该如何构建一个玩家，其能够发现对手的不完美并且学习如何最大化自己获胜的可能性？

X	O	O
O	X	X
		X

本博文介绍基于value function的强化学习来训练tic-tac-toe游戏玩家的方法。首先对这个游戏每个可能的state建立一个数表，每个数都是从这个state开始获胜概率的最新估计。我们把这个估计看做这个state的value。若State A有比B更高的value，说明A比B好，也就是当前估计下，从state A开始获胜的概率比从B开始大。

假设我们是下X的，那么所有有三个X在一行（一列，斜对角）的state的获胜概率是1，所有三个O在一行（一列，斜对角）或者填满（平局）的state的获胜概率是0，其余所有state的初始概率是0.5，代表我们猜测我们有一半几率获胜。

强化学习训练过程中，需要权衡Exploration 和 Exploitation的平衡关系（参考资料4），下面的Tic-Tac-Toe游戏的python代码利用的是greedgreed and ϵ -greed的方法。与对手玩多轮游戏。为了选择下一步该下在哪，我们检测了我们每一个可能的落子之后产生的state，在表中查询他们当前的value。一般情况下，我们是greedy的，也就是选择那些会导致有最高value的state的地方落子，也就是有最高获胜可能性的地方。偶尔，我们会在其他位置里随机选一个落子。这种叫exploratory（试探性的）的落子，因为这能让我们去经历之前没见过的state。

在玩这个游戏的过程中，不停改变这些state的value，尽可能让它们更准确地估计获胜概率。每一次落子之后，将之前state的value向之后state的value靠近。ss表示落子之前的state， s' 表示落子之后的state， $V(S)$ 表示ss对应的value，对于 $V(S)$ 的更新可以写为：

$$V(S) \leftarrow V(S) + \alpha [V(s') - V(S)]$$

α 为步长，是一个小的分数，影响学习率。这种更新规则被称作temporal-difference（时间差分）学习方法。因为其更新是基于两个不同时间对value的估计之差 $V(s') - V(S)$ 。

如果步长参数随着时间适当减小，对于任意固定的对手，这个方法是可以收敛到每个state下我们真实的获胜概率的。更进一步地，我们所选择的落子的地方（除了exploratory的选择）事实上都是对抗对手最优的选择。也就是说，这种方法能够收敛到最优的policy。

强化学习可以应用于系统的高阶和低阶。虽然井字游戏的玩家只学习了游戏的基本动作，但没有什么能阻止强化学习在更高的层次上进行，在这个层次上，每个动作“可能本身就是一种精心设计的解决问题方法的应用”。在分层学习系统中，强化学习可以在多个层次上同时进行。

3.2、训练Tic-Tac-Toe游戏玩家的python代码

```
import numpy as np
import pickle

BOARD_ROWS = 3    ##棋盘行数
BOARD_COLS = 3    ##棋盘列数
BOARD_SIZE = BOARD_ROWS * BOARD_COLS

class State:
    def __init__(self):
        # the board is represented by an n * n array,
        # 1 represents a chessman of the player who moves first,
        # -1 represents a chessman of another player
        # 0 represents an empty position
        self.data = np.zeros((BOARD_ROWS, BOARD_COLS))
        self.winner = None
        self.hash_val = None
        self.end = None

    # compute the hash value for one state, it's unique
    def hash(self):
        ''' 计算棋盘不同state对应的hash值
        ...

        if self.hash_val is None:
            self.hash_val = 0
            for i in self.data.reshape(BOARD_ROWS * BOARD_COLS):
                if i == -1:
                    i = 2
                self.hash_val = self.hash_val * 3 + i
            return int(self.hash_val)

    # check whether a player has won the game, or it's a tie (监测是否有玩家获胜或平局)
    def is_end(self):
        if self.end is not None:
            return self.end
        results = []
        # check row
        for i in range(0, BOARD_ROWS):
            results.append(np.sum(self.data[i, :]))
        # check columns
        for i in range(0, BOARD_COLS):
            results.append(np.sum(self.data[:, i]))

        # check diagonals
        results.append(0)
        for i in range(0, BOARD_ROWS):
            results[-1] += self.data[i, i]
        results.append(0)
```

```

for i in range(0, BOARD_ROWS):
    results[-1] += self.data[i, BOARD_ROWS - 1 - i]

for result in results:
    if result == 3:
        self.winner = 1
        self.end = True
        return self.end
    if result == -3:
        self.winner = -1
        self.end = True
        return self.end

# whether it's a tie (当棋盘下满还没有胜负, 则游戏为平局)
sum = np.sum(np.abs(self.data))
if sum == BOARD_ROWS * BOARD_COLS:
    self.winner = 0
    self.end = True
    return self.end

# game is still going on
self.end = False
return self.end

# @symbol: 1 or -1
# put chessman symbol in position (i, j)
def next_state(self, i, j, symbol):
    new_state = State()
    new_state.data = np.copy(self.data)
    new_state.data[i, j] = symbol
    return new_state

# print the board
def print(self):
    for i in range(0, BOARD_ROWS):
        print('-----')
        out = ' | '
        for j in range(0, BOARD_COLS):
            if self.data[i, j] == 1:
                token = '*'
            if self.data[i, j] == 0:
                token = '0'
            if self.data[i, j] == -1:
                token = 'x'
            out += token + ' | '
        print(out)
    print('-----')

```



```
def get_all_states_impl(current_state, current_symbol, all_states):
    for i in range(0, BOARD_ROWS):
        for j in range(0, BOARD_COLS):
            if current_state.data[i][j] == 0:
                newState = current_state.next_state(i, j, current_symbol)
                newHash = newState.hash()
                if newHash not in all_states.keys():
                    isEnd = newState.is_end()
                    all_states[newHash] = (newState, isEnd)
                    if not isEnd:
                        get_all_states_impl(newState, -current_symbol, all_states)

def get_all_states():
    current_symbol = 1
    current_state = State()
    all_states = dict()
    all_states[current_state.hash()] = (current_state, current_state.is_end())
    get_all_states_impl(current_state, current_symbol, all_states)
    return all_states

# all possible board configurations
all_states = get_all_states()

class Judge:
    # @player1: the player who will move first, its chessman will be 1
    # @player2: another player with a chessman -1
    # @feedback: if True, both players will receive rewards when game is end
    def __init__(self, player1, player2):
        self.p1 = player1
        self.p2 = player2
        self.current_player = None
        self.p1_symbol = 1
        self.p2_symbol = -1
        self.p1.set_symbol(self.p1_symbol)
        self.p2.set_symbol(self.p2_symbol)
        self.current_state = State()

    def reset(self):
        self.p1.reset()
        self.p2.reset()

    def alternate(self):
        while True:
            yield self.p1
            yield self.p2
```

```
# @print: if True, print each board during the game
def play(self, print=False):
    alternator = self.alternate()
    self.reset()
    current_state = State()
    self.p1.set_state(current_state)
    self.p2.set_state(current_state)
    while True:
        player = next(alternator)
        if print:
            current_state.print()
        [i, j, symbol] = player.act()
        next_state_hash = current_state.next_state(i, j, symbol).hash()
        current_state, is_end = all_states[next_state_hash]
        self.p1.set_state(current_state)
        self.p2.set_state(current_state)
        if is_end:
            if print:
                current_state.print()
            return current_state.winner

# AI player
class Player:
    # @step_size: the step size to update estimations
    # @epsilon: the probability to explore
    def __init__(self, step_size=0.1, epsilon=0.1):
        self.estimations = dict()
        self.step_size = step_size
        self.epsilon = epsilon
        self.states = []
        self.greedy = []

    def reset(self):
        self.states = []
        self.greedy = []

    def set_state(self, state):
        self.states.append(state)
        self.greedy.append(True)

    def set_symbol(self, symbol):
        self.symbol = symbol
        for hash_val in all_states.keys():
            (state, is_end) = all_states[hash_val]
            if is_end:
                if state.winner == self.symbol:
                    self.estimations[hash_val] = 1.0
```

```

        elif state.winner == 0:
            # we need to distinguish between a tie and a lose
            self.estimated[hash_val] = 0.5
        else:
            self.estimated[hash_val] = 0
    else:
        self.estimated[hash_val] = 0.5

# update value estimation
def backup(self):
    # for debug
    # print('player trajectory')
    # for state in self.states:
    #     state.print()

    self.states = [state.hash() for state in self.states]

    for i in reversed(range(len(self.states) - 1)):
        state = self.states[i]
        td_error = self.greedy[i] * (self.estimated[self.states[i + 1]] -
self.estimated[state])
        self.estimated[state] += self.step_size * td_error

# choose an action based on the state
def act(self):
    state = self.states[-1]
    next_states = []
    next_positions = []
    for i in range(BOARD_ROWS):
        for j in range(BOARD_COLS):
            if state.data[i, j] == 0:
                next_positions.append([i, j])
                next_states.append(state.next_state(i, j, self.symbol).hash())

    if np.random.rand() < self.epsilon:
        action = next_positions[np.random.randint(len(next_positions))]
        action.append(self.symbol)
        self.greedy[-1] = False
        return action

    values = []
    for hash, pos in zip(next_states, next_positions):
        values.append((self.estimated[hash], pos))
    np.random.shuffle(values)
    values.sort(key=lambda x: x[0], reverse=True)
    action = values[0][1]
    action.append(self.symbol)

```

```
        return action

def save_policy(self):
    with open('policy_%s.bin' % ('first' if self.symbol == 1 else 'second'), 'wb') as f:
        pickle.dump(self.estimateds, f)

def load_policy(self):
    with open('policy_%s.bin' % ('first' if self.symbol == 1 else 'second'), 'rb') as f:
        self.estimateds = pickle.load(f)

# human interface
# input a number to put a chessman
# | q | w | e |
# | a | s | d |
# | z | x | c |
class HumanPlayer:
    def __init__(self, **kwargs):
        self.symbol = None
        self.keys = ['q', 'w', 'e', 'a', 's', 'd', 'z', 'x', 'c']
        self.state = None
        return

    def reset(self):
        return

    def set_state(self, state):
        self.state = state

    def set_symbol(self, symbol):
        self.symbol = symbol
        return

    def backup(self, _):
        return

    def act(self):
        self.state.print()
        key = input("Input your position:")
        data = self.keys.index(key)
        i = data // int(BOARD_COLS)
        j = data % BOARD_COLS
        return (i, j, self.symbol)

def train(epochs):
    player1 = Player(epsilon=0.01)
    player2 = Player(epsilon=0.01)
    judger = Judger(player1, player2)
```

```
player1_win = 0.0
player2_win = 0.0
for i in range(1, epochs + 1):
    winner = judger.play(print=False)
    if winner == 1:
        player1_win += 1
    if winner == -1:
        player2_win += 1
    print('Epoch %d, player 1 win %.02f, player 2 win %.02f' % (i, player1_win / i,
player2_win / i))
    player1.backup()
    player2.backup()
    judger.reset()
player1.save_policy()
player2.save_policy()

def compete(turns):
    player1 = Player(epsilon=0)
    player2 = Player(epsilon=0)
    judger = Judger(player1, player2)
    player1.load_policy()
    player2.load_policy()
    player1_win = 0.0
    player2_win = 0.0
    for i in range(0, turns):
        winner = judger.play()
        if winner == 1:
            player1_win += 1
        if winner == -1:
            player2_win += 1
        judger.reset()
    print('%d turns, player 1 win %.02f, player 2 win %.02f' % (turns, player1_win / turns,
player2_win / turns))

# The game is a zero sum game. If both players are playing with an optimal strategy, every game
will end in a tie.
# So we test whether the AI can guarantee at least a tie if it goes second.
def play():
    while True:
        player1 = HumanPlayer()
        player2 = Player(epsilon=0)
        judger = Judger(player1, player2)
        player2.load_policy()
        winner = judger.play()
        if winner == player2.symbol:
            print("You lose!")
        elif winner == player1.symbol:
```

```
        print("You win!")
    else:
        print("It is a tie!")

if __name__ == '__main__':
    ##两个AI玩家进行强化学习训练
    train(int(1e5))
    ##两个AI玩家进行强化学习测试
    compete(int(1e3))
    ##human玩家与AI玩家比赛
    play()
```

参考资料

- 1、《深入浅出强化学习原理入门》
- 2、《Reinforcement Learning: An Introduction》
- 3、<https://zhuanlan.zhihu.com/p/50347818>
- 4、<https://blog.csdn.net/LagrangeSK/article/details/81010195>

Q-learning(on-policy)

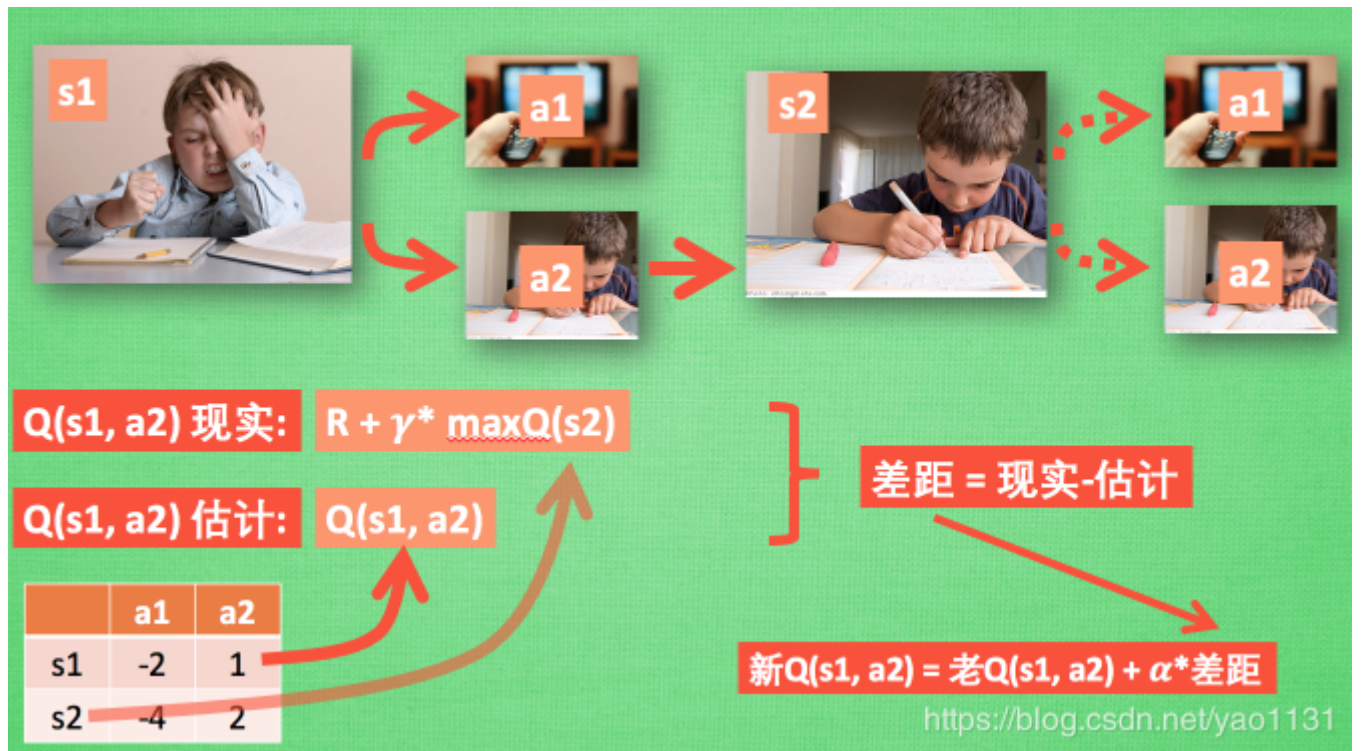
参考: <http://www.mnemstudio.org/path-finding-q-learning-tutorial.htm>
<http://www.mnemstudio.org/path-finding-q-learning-tutorial.htm>

理论知识:

假设一个小学生放学后有两种选择——看电视或者是做作业，这里小学生就是agent，看电视和做作业就是两种action，分别记为a1和a2。然后建立奖惩机制，看电视会扣2分，做作业会加1分，初始化Q表在初始化状态下为0.然后根据我们的奖惩机制可以，更新在S1状态的Q表值。



由于做作业的Q值高，所以我们会选择 $a2$ 动作，也即 $Q(s1, a2)$ ，这样下一个状态 $s2$ 也同样会对应两个动作。Q-learning就是在考虑本步状态的同时也会涉及到下一个状态。对于这里 $s2$ 中 $a2$ 动作的Q值比 $a1$ 大，所以在更新 $Q(s1, a2)$ 时，把大的 $Q(s2, a2)$ 乘上一个衰减值 γ (比如是0.9) 并加上到达 $s2$ 时所获取的奖励 R (这里还没有获取到我们的棒棒糖，所以奖励为 0)，因为会获取实实在在的奖励 R ，我们将这个作为我现实中 $Q(s1, a2)$ 的值，这个现实值会与老值做差，然后以一个学习效率 α 累加上老的 $Q(s1, a2)$ 的值 变成新的值，如下图。



算法的思路如下：


```


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.,  $\epsilon$ -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
  
```


$Q(s2)$ 最大估计
 $Q(s1, a2)$ 现实
 $Q(s1, a2)$ 估计


问题1：更新到什么时候是个头啊？

式子中的Gamma,取不同值对应的效果，如下图：

$Q(s1) = r2 + \gamma Q(s2) = r2 + \gamma [r3 + \gamma Q(s3)] = r2 + \gamma [r3 + \gamma [r4 + \gamma Q(s4)]] = \dots$
 $Q(s1) = r2 + \gamma r3 + \gamma^2 r4 + \gamma^3 r5 + \gamma^4 r6 + \dots$

$\gamma = 1$  $Q(s1) = r2 + 1 * r3 + 1 * r4 + 1 * r5 + 1 * r6 + \dots$

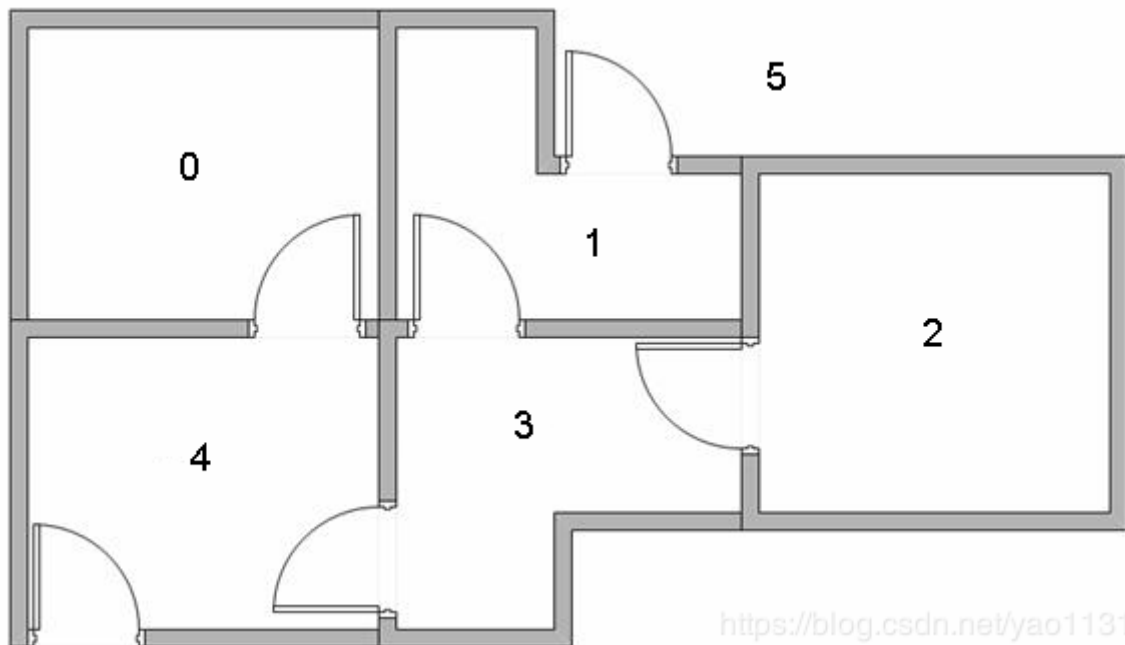
$\gamma = (0 \sim 1)$  $Q(s1) = r2 + \gamma r3 + \gamma^2 r4 + \gamma^3 r5 + \gamma^4 r6 + \dots$

$\gamma = 0$  $Q(s1) = r2$

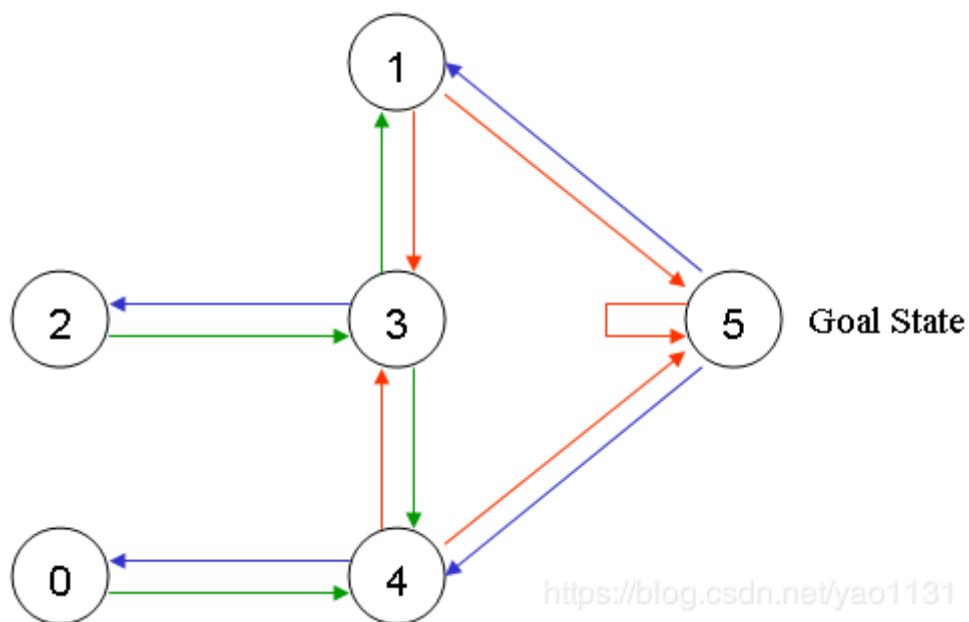
<https://blog.csdn.net/yao1131>

趣味例子：

结合一个例子介绍Q-learning算法，首先，假设我们有五个房间0-4，其中5代表的是外面，这个小迷宫如下图。部分房间与房间是相通的，例如：房间4和房间0，3以及5相通。

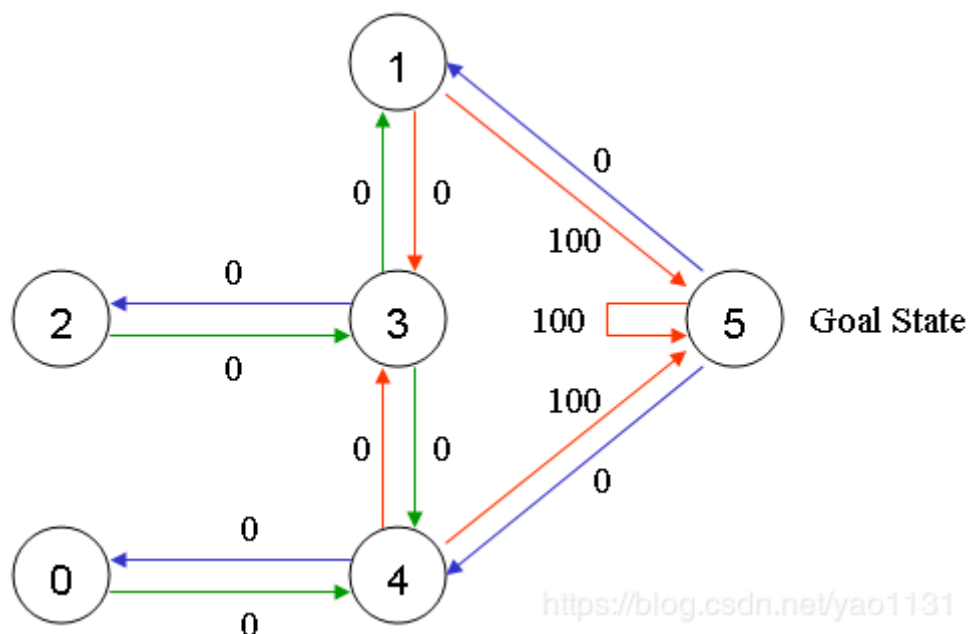


简化上述过程如下：相互连接起来的两个房间代表相通。



现在我们将一个agent随机放进这个小迷宫中，我们的目标是让agent走出去，也就是到达5号房。基于此，我们建立了如下的奖惩机制，能够直接引导agent到达目的地的路径给予100奖励，其他的没有和5号房直接连接的路径给

予0奖励，表示如下：



在此基础上，我们初始化Q值为0，如下：

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

路径对应的回报如下：

$$R = \begin{matrix} & \begin{matrix} \text{Action} \\ 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{matrix}$$

现在假设我们的agent位于房间1，然后随机选择下一个状态为房间5，则对于房间5会有1, 4, 5三种选择。

$$Q(\text{state}, \text{action}) = R(\text{state}, \text{action}) + \text{Gamma} * \text{Max}[Q(\text{next state}, \text{all actions})]$$

$$Q(1, 5) = R(1, 5) + 0.8 * \text{Max}[Q(5, 1), Q(5, 4), Q(5, 5)] = 100 + 0.8 * 0 = 100$$

Q值表更新为：

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

(100写错位了，对应 (1, 5) 处为100才对)

此时，到达终点房间5，然后结束本回合。开始下一回合，假设在房间3，我们有1, 2和4三种选择，随机选择1为下一个状态，则：

$$Q(\text{state}, \text{action}) = R(\text{state}, \text{action}) + \text{Gamma} * \text{Max}[Q(\text{next state}, \text{all actions})]$$

$$Q(3,1) = R(3, 1) + 0.8 * \text{Max}[Q(1, 3), Q(1, 5)] = 0 + 0.8 * \text{Max}(0, 100) = 80$$

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

以此类推不断更新最后得到：

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{matrix}$$

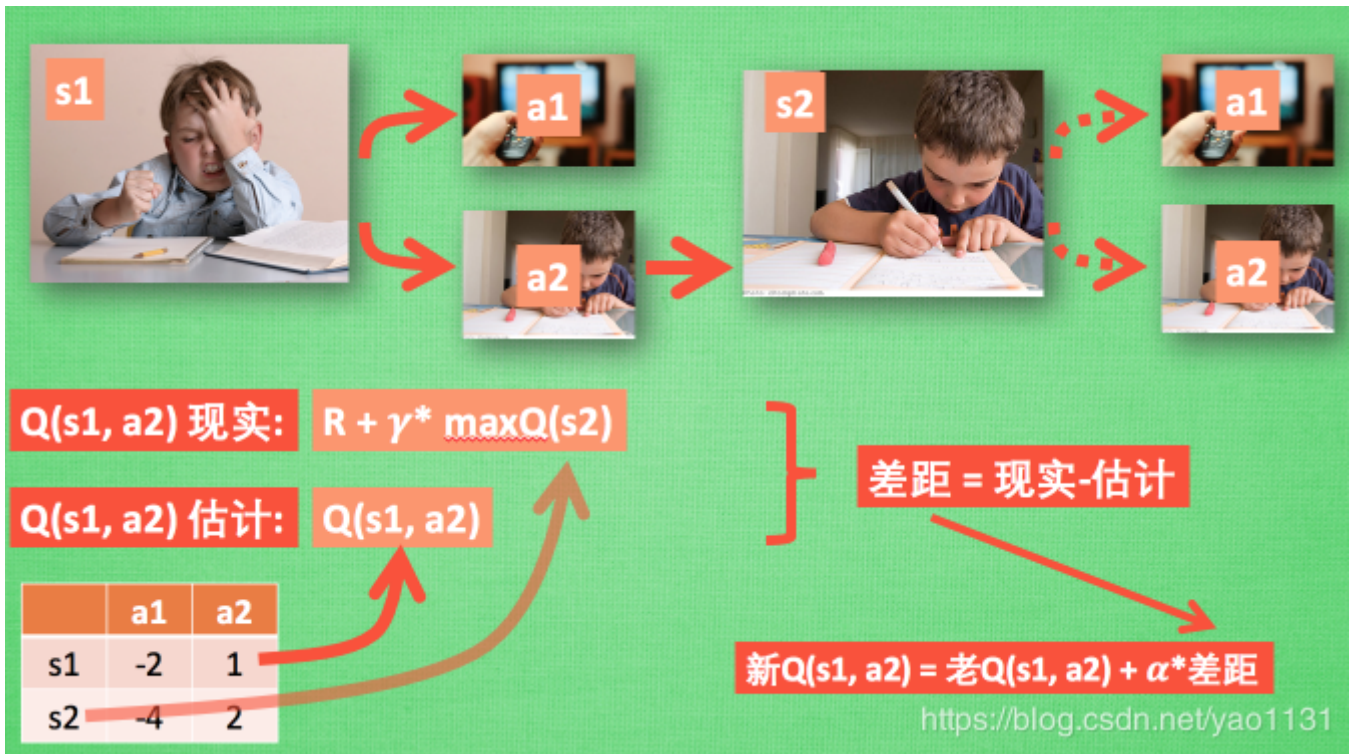
(其中对于Q值更新部分说明：对于这个例子来说，即便是按Q值更新，一旦所有位置的Q值确定后，除了第五列的三个值以外，其他值无论再算，都是固定的了，因为对应的路径回报值R都是0，所以这些Q值也就取决于其对应下一个状态的最大Q值了，至于第五列的三个Q值更新问题倒是值得进一步细究)

问题2：第五列的三个Q值更新问题？

SARSA(off-policy)

理论知识

类比Q-learning, SARSA就相对好理解很多, 差别仅仅在于现实值求法的不同, 其中Q-learning中取的是下一个状态中所有动作对应的最大的Q值, 而此处取的是下一个状态确定动作的Q值。对比下图和上面Q-learning对应的算法图。



细节算法流程对比:

