Обучение с подкреплением (Reinforcement Learning, RL)

RL - набор методов, позволяющих агенту (интеллектуальной системе) вырабатывать оптимальную стратегию при его взаимодействии со средой (внешним миром).

В каждый (обычно дискретный) момент времени t агент может оказать на среду некоторое воздействие (action) a_t. В ответ он получает информацию – наблюдение (observation) o_{t+1}, полностью или частично характеризующее новое состояние среды (state) s_{t+1} и числовую характеристику успешности действий, называемую наградой (reward) r_{t+1}

$$o_0 \xrightarrow{a_0} \{o_1, r_1\} \xrightarrow{a_1} \{o_2, r_2\} \xrightarrow{a_2} \dots$$

Обучение с подкреплением (Reinforcement Learning, RL)

Важное отличие обучения с подкреплением от стандартного обучения с учителем - это отсутствие обучающих данных вида ,

$$\{(s, a), (s', a'), \ldots\}$$

где *a* - "известные учителю" оптимальные действия, которые необходимо выполнить, когда среда находится в данном состоянии.

Агент должен решить задачу, даже если человек не знает как её решать (т.е. не знает какие действия оптимальны).

Обучение с подкреплением (Reinforcement Learning, RL)

На основе полученной к моменту времени ***t*** информации

$\{\ldots; o_{t-2}, a_{t-2}; o_{t-1}, a_{t-1}; o_t \_\}$ *(траектории)*

агент должен выбрать очередное действие $\boldsymbol{a_t}$ так, чтобы максимизировать накопленное вознаграждение ***R = r1+r2+…***
за достаточно большой промежуток времени.

Среда (environment) может быть -

- детерминированной или стохастической;
- с полным или неполным описанием;
- стационарной или нестационарной.

В среде могут быть терминальные состояния, при достижении которых "игровой" эпизод оканчивается.

Обучение с подкреплением (Reinforcement Learning, RL)

Возможные действия $\boldsymbol{a_t}$ могут принадлежать конечному множеству целых чисел (0 - шаг влево или 1 - шаг вправо) или быть вектором вещественных чисел: {скорость шага, угол поворота}.

Во многих вариантах реализации методы способны работать или только с дискретными действиями, или только с непрерывными действиями.

В типичных задачах RL вознаграждение r_{t+1} часто не связано непосредственно с действием $\boldsymbol{a_t}$. Отличное от нуля вознаграждение может поступать редко, запаздывать и возникать только после серии негативных значений (долго шел, но затем много получил). Иногда вознаграждение приходит только в конце (например, при игре в *го*). Вознаграждение также может быть всегда отрицательным.

Обучение с подкреплением (Reinforcement Learning, RL)

Обычно вместо наблюдения за средой o_t, говорят о её состоянии s_t. Следует помнить, что в общем случае эти величины могут не совпадать. Для среды с полным описанием $s_t = o_t$.

При неполном описании нужно учитывать историю наблюдений и действий. Но даже при учёте всей траектории *s0, a0 s1, a1, …*
состояние ***среды*** может быть *вероятностно* или оказаться "вещью в себе« (т.е. неопределённо).

Обучение с подкреплением (Reinforcement Learning, RL)

В процессе обучения агент может формировать модель среды, которая предсказывает вероятность следующего наблюдения на основании траектории:

$$P(s_{t+1}, r_{t+1} \mid \ldots, s_t, a_t)$$

Если награда определяется только состоянием, то модель это

$$P(s_{t+1}, \mid \ldots, s_t, a_t) \qquad r_t = r(s_t)$$

Если текущее состояние среды S_t содержит всю информацию, необходимую для принятия решения, то такая среда называется марковской.
Формально, это означает, что условная вероятность перехода в новое состояние зависит только от текущего состояния и действия:

$$P(s_{t+1}, r_{t+1} \mid \ldots, s_{t-1}, a_{t-1}, s_t, a_t) = P(s_{t+1}, r_{t+1} \mid s_t, a_t)$$

Описания процесса

Пространство состояний СМО

M / M / 1

Число означает количество заявок в системе
λ - интенсивность поступления заявок
μ – параметр показательно распределенного времени обслуживания в ОА

марковский процесс

Обучение с подкреплением (Reinforcement Learning, RL)

Выбор оптимальной стратегии в такой среде называется **марковским процессом принятия решения** (Markov decision process, MDP).

Если наблюдение не даёт всей информации о состоянии (неполное описание, частичная наблюдаемость), то решения в такой среде называется *частично наблюдаемым* марковским процессом принятия решения (partially observable Markov decision process, POMDP).

Обучение с подкреплением (Reinforcement Learning, RL)

Алгоритм, по которому агент выбирает действие, принято называть его **политикой**.

Это может быть обычная (детерминированная) функция

$$a_t = \pi(s_t) \qquad \textit{(policy function)}$$

где в качестве аргумента выступает текущее состояние s_t.

В s_t могут также включаться предшествующие действия и награды.

Для изучения среды агент должен постоянно *экспериментировать.* Даже в стационарной среде с полным описанием, эксперименты нужны, чтобы *не застрять* в некоторой области пространства состояний, т.к. возможно, есть более выгодные состояния среды.

Поэтому в общем случае **политика** - это условная вероятность $p(a|s)$ выбора агентом действия a в состоянии s

Обучение с подкреплением (Reinforcement Learning, RL)

Детерминированную функцию политики можно превратить в вероятностную. Например, с вероятностью *p* выбирается оптимальное (к данному моменту обучения) действие $a = \pi(s)$, а с вероятностью *1-p* - равновероятно любое другое состояние.

Если множество действий дискретно, можно, например, аппроксимировать $p(a|s)$ нейронной сетью с числом выходов равных числу возможных действий на которых будет (после softmax) распределение вероятностей соответствующего решения.

Граф состояний системы

Многие задачи обучения с подкреплением сводятся к марковским процессам. Формально, это означает, что условная вероятность перехода в новое состояние зависит только от текущего состояния, выполненного в нём действия и не зависит от предыдущих состояний и действий.

$$P(s_{t+1}\, r_{t+1} \,|\, \dots\ s_{t-1},\, a_{t-1},\ s_t,\, a_t) = P(s_{t+1}\, r_{t+1} \,|\, s_t,\, a_t)$$

В этом случае будущее определяется настоящим и не требует знания прошлого.
В общем случае состояние может зависеть от нескольких предыдущих наблюдений.
Выбор оптимальной стратегии (policy function) в такой марковской модели называется марковским процессом принятия решения
(Markov decision process, MDP).

Модель Марковских переходов

Рассмотрим модель, в которой нет действий.
Среда в дискретные моменты времени из состояния s случайно переходит в очередное состояние s’ с вероятностями $P_s = P(s' \mid s)$
При попадании в состояние с номером s’ начисляется фиксированное вознаграждение r’, зависящее от s’.
В результате получается временная последовательность *(нижний индекс - момент времени)*:

$$s_0 \mapsto s'_1, r'_1 \mapsto s''_2, r''_2 \mapsto \dots, \qquad s_t \in \mathbb{S},\ r_t \in \mathbb{R},$$

где ***S*** - множество целых чисел (номера состояний), а ***R*** - множество вещественных чисел (награды).
Нас интересует суммарное будущее дисконтированное вознаграждение, полученное после момента времени *t*

$$R_{t+1} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = r_{t+1} + \gamma R_{t+2}$$

Ценностью состояния ***V***(s) называется суммарное дисконтированное вознаграждение, при старте из состояния ***s*** в последующих состояниях, усреднённое по всем возможным траекториям:

$$V(s) = \langle R_{t+1} \mid s_t = s \rangle$$

Запишем будущее накопленное дисконтированное вознаграждение в виде

$R_{t'} = r_{t'} + gR_{t''}$ *где* $t'=t+1$ $t''=t+2$

Усредним его выражение по траекториям, стартующим из состояния $s_t = s$

$$V(s) = \langle r_{t+1} + \gamma R_{t+2} \,|\, s_t = s \rangle$$

Пусть среда из состояния с номером *i* может перейти в состояния 1, 2, ..n .
При переходе в *j*-е состояние агент получит вознаграждение r_j (за попадание в него) плюс дисконтированное будущее вознаграждение равное V*j* за всю последующую историю (ценность *j*-го состояния).
Для получения V*i* все эти награды следует усреднить.

Для этого умножим их на вероятности переходов $P_{ij} = P(s_{t'} = j \mid s_t = i)$
и просуммируем по *j*:

$$V_i = \sum_j P_{ij} \cdot (r_j + \gamma V_j)$$

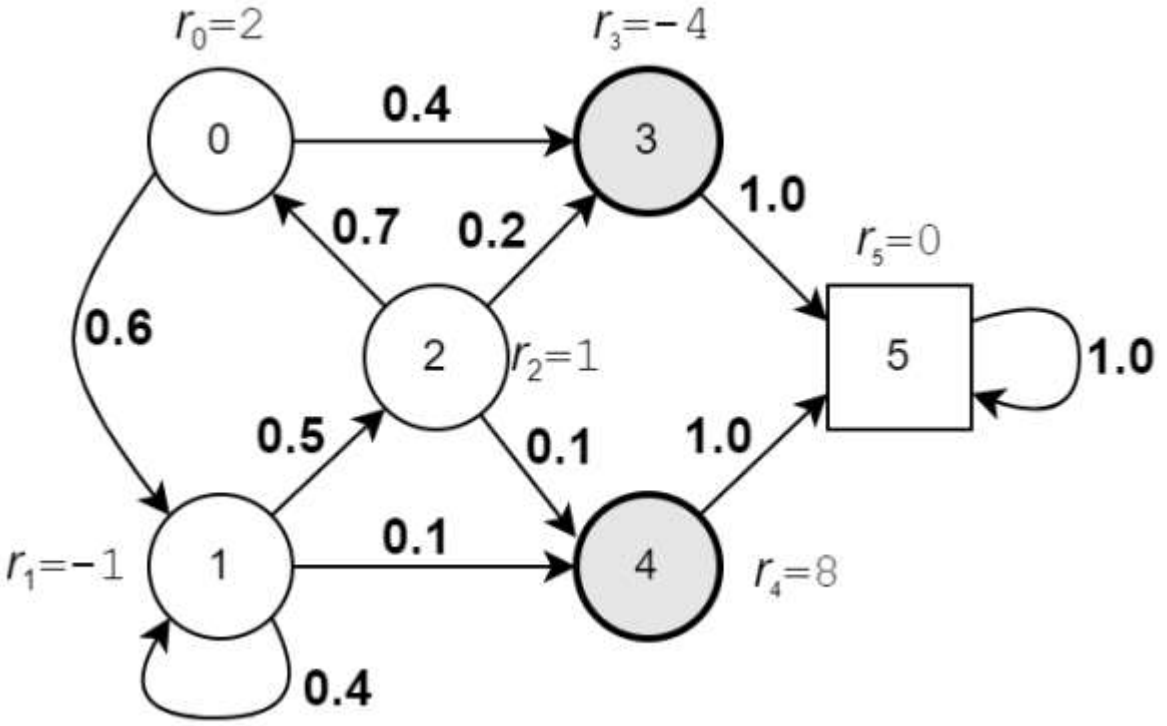
Эта формула называется уравнением Беллмана

Пример модельной среды

Пусть среда может находится в пяти состояниях, два из которых (3 и 4) терминальные (после них "блуждание" заканчивается).

Введем ещё одно состояние (на рисунке квадрат) при попадании в которое вознаграждения нет и там среда остаётся "навсегда" и награды перестают начисляться (конец блужданиям). Вероятности переходов указаны рядом со стрелками, а вознаграждения - возле состояний. Запишем вектор вознаграждений и матрицу вероятностей переходов для такой среды:

```
import numpy as np
r = np.array(  [2.,  -1.,   1.,  -4.,   8.,   0. ] )
P = np.array([ [0.,  0.6,  0.,  0.4,  0.,  0., ],
               [0.,  0.4,  0.5,  0.,  0.1,  0.,  ],
               [0.7,  0.,  0.,   0.2, 0.1,  0.,  ],
               [0.,   0.,   0.,   0.,   0.,  1.0,  ],
               [0.,   0.,   0.,   0.,   0.,  1.0,  ],
               [0.,   0.,   0.,   0.,   0.,  1.0,  ]  ])
```



Пример модельной среды

Для получения функции полезности состояний, воспользуемся матричным решением уравнения Беллмана с $\gamma = 0.9$:

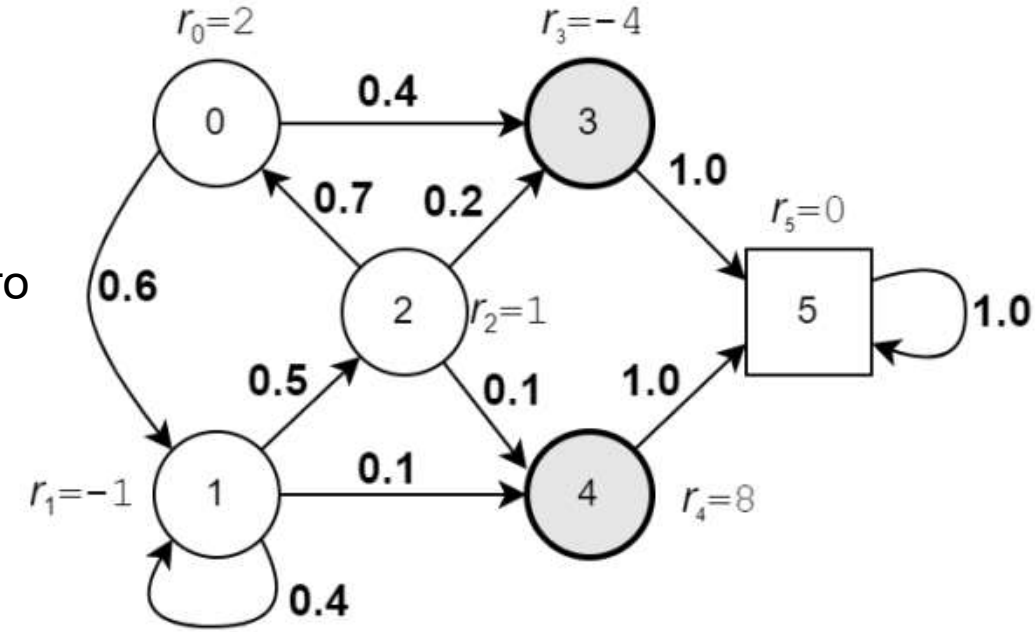
```
from numpy.linalg import inv  #получение обратной матрицы
gamma = 0.9
np.dot(inv( np.eye(len(r)) - gamma*P), np.dot(P, r))
```

В библиотеке *numpy* функция *dot* вычисляет произведение со свёрткой (в данном случае сумма по *j* от $P_{ij} \cdot r_j$), *eye* - это единичная матрица. В результате получаем

[-1.195, 1.861, 0.647, 0., 0., 0.]

Ценности терминальных состояний 3,4,5 нулевые (начиная с них агент дальше ничего не получает).

Наименее ценное состояние 0, а наиболее ценное - состояние 1.



Пример модельной среды

Заметим, что можно не вводить терминального состояния. Однако тогда награды должны быть матрицами r_{ij} . Например, при переходе из терминального состояния 4 в него же награда будет нулевая $r_{44}=0$.

В тоже время $r_{14}= 8$.

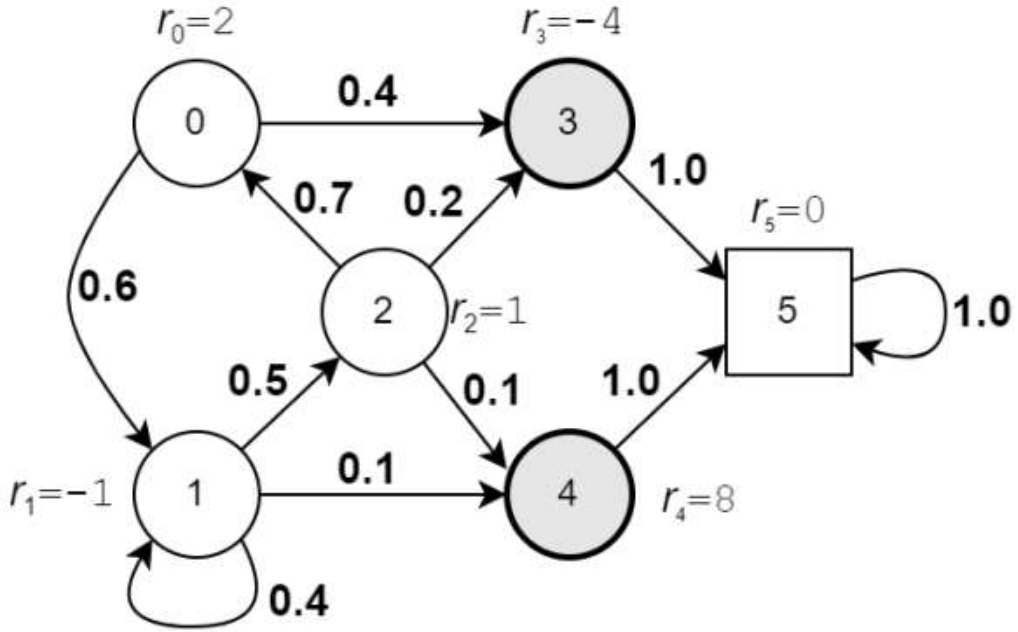
Уравнение Беллмана в этом случае принимает следующий вид:

$$V_i = \sum_j P_{ij} \cdot \left(r_{ij} + \gamma V_j\right)$$

В общем случае, награды могут быть вероятностными и тогда:

$$V_i = \sum_{j,\,r} P_{i\,j\,r} \cdot \left(r + \gamma V_j\right)$$

где *Pijr* – вероятность перехода из состояния *i* в состояние *j* с получением при этом награды *r* [*P(j, r|i)*]



Пример моделирования Монте-Карло

Найдём ценности состояний этой же среды при помощи статистического моделирования. Для этого будем много раз запускать блуждание среды начиная с одного и того же состояния *s0*, вычисляя по каждому эпизоду полученное вознаграждение:

```
states       = np.arange(len(r))              # [0,1,2,3,4,5]
terminals    = [3,4]                          # терминальные состояния
rews, steps = [], []                          # вознаграждения и шагов
s0           = 0                              # стартовое состояние

for _ in range(10000):                        # делаем 10000 экспериментов
    rew, s, discont = 0, s0, 1
    for i in range(100):                      # на всякий случай ограничиваем
        s = np.random.choice(states,p=P[s]) # состояние по распределению P[s]

        rew += r[s] * discont                 # накапливаем вознаграждение
        discont *= gamma                      # сильнее дисконтируем

        if s in terminals:                    # попали в терминальное состояние
            break                             # дальше вознаграждение не меняется

    rews.append(rew)                          # сохраняем вознаграждение
    steps.append(i+1)                         # сохраняем число сделанных шагов
```

Ошибка убывает как $\sigma / \sqrt{N}$, где σ - стандартное отклонение std, а N=10000 - число экспериментов. В рамках этой ошибки результаты моделирования совпадают с точными значениями, полученными при помощи уравнения Беллмана.

Функция политики

Рассмотрим типичную (для обучения с подкреплением) постановку задачи.

Пусть в состоянии с номером i агент с вероятностью $\pi_{i\alpha}=\pi(\alpha|i)$ совершает действие с номером α (латинские индексы используются для номеров состояний, а греческие - для действий).

В результате действия агента среда переходит в состояние под номером j с вероятностью $P_{i\alpha j} = P(j|i,\alpha)$ получая в нём награду $r_j = r(s_j)$.

$$V_i = \sum_j \tilde{P}_{ij}\left(r_j + \gamma V_j\right), \qquad \tilde{P}_{ij} = \sum_\alpha \pi_{i\alpha} P_{i\alpha j}$$

Ценности состояний по-прежнему удовлетворяют уравнению Беллмана, где P_{ij} вероятности перехода из i в j равны произведению вероятности сделать в состоянии i действие α на вероятность перейти после этого в состояние j с суммой по всем возможным действиям.

Функция политики

Найдём уравнение для Q-функции полезности действия α , совершённого в состоянии s:

$$Q(s,a) \;=\; \langle R_{t+1} \,|\, s_t = a,\; a_t = a \rangle$$

Для дискретного случая это будет матрица $Q_{i\alpha}$.
Зная её, можно вычислить полезность состояния V (при совершении любого действия с вероятностями $\pi_{i\alpha}$):

$$V_i = \sum_{\alpha} \pi_{i\alpha} Q_{i\alpha}$$

Можно также записать обратную связь. Ценность действий $Q_{i\alpha}$ в состоянии i равна среднему вознаграждению при непосредственном переходе в состояние j плюс дисконтированные будущие вознаграждения состояний, в которые среда попадает при таком переходе:

$$Q_{i\alpha} = \sum_{j} P_{i\alpha j} \left(r_j + \gamma V_j \right)$$

полезности состояний V и действий Q зависят от выбранной агентом политики (матрицы вероятностей $\pi_{i\alpha}$). Изменение политики меняет и эти функции.

Функция политики

Разные политики $\pi(a|s)$ приводят к различным Q и V функциям. Из всех возможных политик оптимальной будет та, для которой значения ценности состояний и действий максимальны:

$$V^*(s) = \arg\max_\pi V^\pi(s), \qquad Q^*(s,a) = \arg\max_\pi Q^\pi(s,a)$$

В свою очередь, если известна оптимальная Q*-функция несложно записать оптимальную детерминированную политику:

$$a = \pi^*(s) = \arg\max_a Q^*(s,a)$$

В процессе обучения (пока Q известна плохо) такую детерминированную политику можно рандомизировать, выбрав параметр ε и равномерно распределённое случайное число p

$$a = \begin{cases} \arg\max_a Q(s,a) & p > \epsilon \\ \text{any} & \text{else} \end{cases}$$

Функция политики

Если *ε=0* то всегда выбирается оптимальное действие. При *ε=0.5* в половине случаев будет выбираться случайное действие. По мере обучения *ε* уменьшается, что позволяет на начальных этапах агенту больше "экспериментировать".

Вместо *ε* -стратегии действие можно также выбирать с вероятностями пропорциональными их полезности.
Для этого используется функция softmax по индексу α:

$$\pi_{i\alpha} = \mathrm{soft}(Q_{i\alpha} - \bar{Q}_{i\alpha}), \qquad \mathrm{soft}(x_\alpha) = \frac{e^{x_\alpha}}{\sum_\beta e^{x_\beta}}$$

где среднее Qiα (также по индексу α) вычитается, чтобы не произошло переполнения при вычислении экспонент. Решение на Python:

```
from scipy.special import softmax  # импортируем функцию softmax

Q  = np.random.uniform( size=(num_states, num_actions) )
aQ = Q.mean(axis=1, keepdims=True) # средние значения (num_states, 1)
pi = softmax(Q - aQ, axis=1)       # условные вероятности pi(a|s)
s  = 1                             # номер состояния
a  = np.random.choice(np.arange(num_actions), p=pi[s])
```

Максимизация Q-функции

Пусть агент выбирает действие, соответствующее его максимальной полезности a = argmax$_\alpha$ Q(s, a) и всегда ей следует. Тогда:

$$V_i = \sum_\alpha \pi_{i\alpha} Q_{i\alpha} = \max_\alpha Q_{i\alpha}$$

так как все вероятности политики равны нулю, кроме того действия, для которого $Qi\alpha$ максимальна. Поэтому, уравнение Беллмана $Qi\alpha = \sum_j Pi\alpha j\ (rj + \gamma\ Vj)$ для полезности действий в оптимальной стратегии приобретает вид:

$$Q_{i\alpha} = \sum_j P_{i\alpha j} \left(r_j + \gamma \max_\beta Q_{j\beta}\right)$$

Даже, если модель среды ($Pi\alpha j$, rj) известна, непосредственно это уравнение решить нельзя. Однако можно воспользоваться итерационным методом:

$$Q_{i\alpha}^{\text{new}} = Q_{i\alpha} + \lambda \Big[\sum_j P_{i\alpha j} \left(r_j + \gamma \max_\beta Q_{j\beta}\right) - Q_{i\alpha} \Big]$$

где λ - гиперпараметр (скорость обучения). Смысл этого соотношения такой: если Qiα меньше, чем требуемое по уравнению Беллмана значение, то её необходимо немного увеличить, а если больше - то уменьшить. Параметр λ регулирует степень изменения Qiα на каждой итерации.

Метод Q-обучения (Q-learning)

Если модель среды неизвестна, используют статистическое моделирование, в процессе которого подправляют значения *Qiα*.
При этом, для исследования пространства состояний применяется ε-жадная стратегия с убывающим ε :

$$a = \pi_\epsilon(s) = \text{any } \textbf{if } p < \epsilon \textbf{ else } \arg\max_a Q(s_t, a)$$

Тогда на каждом шаге моделирования производится вычисление:

$$a_t = \pi_\epsilon(s_t) \quad \mapsto \quad s_{t+1},\ r_{t+1}$$

$$Q^{\text{new}}(s_t, a_t) = Q(s_t, a_t) + \lambda \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Выражение в квадратных скобках называется temporal difference (TD) и обычно обозначается символом δ (дельта).
Если δ > 0, то переход $s, a \mapsto s', r'$ является позитивно неожиданным, а если δ < 0 - то негативно неожиданным.

Такой итерационный метод называется Q-learning. Это типичный пример дообучения, когда обучение происходит на каждой новой полученной порции данных.

Метод Q-обучения (Q-learning)

Параметр жадности ε начнем со значения eps1, экспоненциально уменьшая его до значения eps2 в течении decays эпох, после чего положим $\varepsilon = 0$

```
Q = np.zeros( (num_states, num_actions) )  # начальные значения 0

eps1, eps2, decays = 1, 0.001, 50000       # параметры epsilon-распада
epsilon = eps1
decay   = math.exp(math.log(eps2/eps1)/decays)

def policy(s):                             # epsilon-жадная политика
    if np.random.random() < epsilon:       # случайно любое действие
        return np.random.randint(n_actions)
    return np.argmax(Q[s])                 # иначе лучшее

def run_episode(ticks=1000):
    s0 = env.reset()                       # начальное состояние среды
    for _ in range(ticks):
        a0 = policy(s0)                    # выбираем действие
        s1, r1, done = env.step(a)         # новое состояние и награда

        Q[s0, a0] += lm * ( r1 + gamma * np.max(Q[s1]) - Q[s0,a0] )

        if done:                           # терминальное состояние
            return

        s0 = s                             # продолжаем с нового состояния
```

Итерационный поиск политики

Если модель среды P(s|a) известна, можно воспользоваться простым итерационным алгоритмом вычисления политики (policy iteration algorithm). В этом алгоритме сначала задаётся некоторая детерминированная политика $\pi(a|s)$ (она обычно инициализируется случайными значениями). На её основе вычисляется V-функция. Затем, с её помощью строиться уточнённая (улучшенная) политика и вычисления повторяются.

Выполняются два повторяющихся этапа:

1. Оценка политики (policy evaluation). Итерационно вычисляем для всех состояний, пока с некоторой точностью функция ценности не перестанет изменяться:

$$V_{k+1}(s) = \sum_{s',r'} P(s', r' \mid s, a) \left[r' + \gamma V_k(s') \right], \qquad a = \pi(s)$$

В принципе, можно использовать два массива для k-ой итерации V_k и для следующей V_{k+1} и затем менять их местами. Однако хорошо работает обновление по месту, при котором для некоторых состояний уже используется улучшенное значение ценности.

2. Улучшение политики (improving the policy) при котором вычисляется политика на основании жадной стратегии:

$$\pi(s) = \arg\max_a Q(s, a) = \arg\max_a \sum_{s',r'} P(s', r' \mid s, a) \left[r' + \gamma V_k(s') \right]$$

Пример -- Frozen Lake

Рассмотрим в качестве примера среду Frozen Lake из набора сред Gymnasium. Есть квадратная неизменная карта замёрзшего озера, но часть ячеек которого содержит полыньи. Необходимо провести гнома из левого верхнего угла карты (стартовое состояние) в правый нижний (целевое состояние), не провалившись в полынью.

Однако, гном *не вполне здоров (нетрезвый)* и его шатает вправо-влево. Поэтому на скользком льду (в режиме **is_slippery=True**) он с вероятностью 1/3 сместится туда, куда планирует, а с вероятностями 1/3 сдвинется в одном из перпенди-кулярных направлений. При попытке выйти за карту гном останется на прежней ячейке-упрется в стену- или его шатнёт в перпендикулярном направлении.

В примере есть 16 состояний (номера ячеек) и 4 действия (0: LEFT, 1: DOWN, 2: RIGHT, 3: UP).
При попадании в полынью или в целевую ячейку эпизод оканчивается.
При достижении целевой ячейки даётся награда 1, иначе награда 0.
Среда FrozenLake **сильно** стохастична и имеет очень скудную награду (только при удачном конце эпизода).

Пример -- Frozen Lake

Оптимальная политика, приведенная на рисунке, может показаться неожиданной. Однако, с учётом стохастичности среды она реальна. Рассмотрим, например, состояние 6. Если бы гном попытался сместится вниз, это бы у него получилось бы только с вероятностью 1/3, а с вероятностью 2/3 он бы оказался в соседней полынье. Но при движении влево его с вероятностью 1/3 качнёт вниз (куда нужно), а в полынью он провалится уже с вероятностью 1/3.
Следуя такой политике, гном гарантированно (хотя с возвратами) пройдёт ячейки 0,4,8,9, после чего его маршрут может разветвиться, и в том числе зациклиться.

В примере есть 16 состояний (номера ячеек) и 4 действия (0: LEFT, 1: DOWN, 2: RIGHT, 3: UP).

```
def policy(s):
    return [0,3,3,3, 0,0,0,0, 3,1,0,0, 0,2,1,0][s]
```

Пример -- Frozen Lake

Из вероятностных соображений несложно вычислить V-функцию при оптимальной стратегии. Например, для состояния 0 имеем

$$V0 = (1/3)\, V0 + (1/3)\, V1 + (1/3)\, V4$$

а для состояния 14 при γ = 1 имеем

$$V14 = (1/3)\, V14 + (1/3)\, V13 + (1/3) V15(1*0)$$

Записывая подобные соотношения для состояний 0, 1, 2, 3, 4, 8, 9, 13, 14, 6, 10, получаем следующую матрицу:

$$V(s) = \frac{1}{17} \begin{pmatrix} 14 & 14 & 14 & 14 \\ 14 & 0 & 9 & 0 \\ 14 & 14 & 13 & 0 \\ 0 & 15 & 16 & 0 \end{pmatrix}$$

Метод Q-learning справляется с этой задачей за 10000 эпизодов. При этом метод неустойчив при gamma=1 и не сходится при малых decays. В этой задаче награда выдаётся только в целевом состоянии. Пока агент его не достигнет матрица *Q(s,a)* будет оставаться нулевой. Только после первого попадания в цель, начнётся распространение значений по *Q(s,a)*, пока она не стабилизируется, поэтому в данном случае использование ε-стратегии необходимо.

Пример -- Frozen Lake

Изучите среду Frozen Lake, применив к ней метод Q-learning:

1. Начните -без скольжения env = gym.make("FrozenLake-v1", is_slippery=False)

2. Сравните -со скольжением env = gym.make("FrozenLake-v1", is_slippery=True)

3. Постройте кривые обучения.

4. Изучите свойства гиперпараметров дисконтирования decays, gamma, lm.

Пример -- Frozen Lake