Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное
образовательное учреждение высшего образования
«Московский физико-технический институт
(национальный исследовательский университет)»

Кафедра системных исследований

МАШИННОЕ ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ НА PYTHON

Учебно-методическое пособие

Составители: *А. И. Панов*
А. А. Скрынник

МОСКВА
МФТИ
2019

УДК 004.853(075)
ББК 22.18я73
М38

Рецензент
Кандидат физико-математических наук ФИЦ ИУ РАН *Д. А. Макаров*

Машинное обучение с подкреплением на Python : учеб.-
М38 метод. пособие / сост.: А. И. Панов, А. А. Скрынник. – Москва: МФТИ, 2019. – 54 с.

В учебно-методическом пособии рассмотрены упражнения компьютерного практикума по курсу машинного обучения с подкреплением – активно развивающегося направления в искусственном интеллекте.

Рассмотрены основные алгоритмы динамического программирования, вычисления функции полезности, реализация методов, основанных на полезности, и методов, основанных на стратегии. Особое внимание уделено алгоритмам аппроксимации и реализации иерархических подходов.

Предназначено для студентов старших курсов и аспирантов, изучающих методы искусственного интеллекта, машинное обучение и интеллектуальные робототехнические системы.

Учебное издание

МАШИННОЕ ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ НА PYTHON

Учебно-методическое пособие

Составители: **Панов** Александр Игоревич,
Скрынник Алексей Александрович

Редактор *Н. Е. Кобзева*. Корректор *И. А. Волкова*
Компьютерная верстка: *А. И. Панов*, *Н. Е. Кобзева*
Подписано в печать 28.06.2019. Формат 60×84 $^1/_{16}$.
Усл. печ. л. 3,4. Уч.-изд. л. 1,9. Тираж 100 экз. Заказ № 213.

Федеральное государственное автономное образовательное учреждение высшего образования «Московский физико-технический институт (национальный исследовательский университет)»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9
Тел. (495) 408-58-22, e-mail: rio@mipt.ru

Отдел оперативной полиграфии «Физтех-полиграф»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9
E-mail: polygraph@mipt.ru

Содержание

Введение

В данном пособии рассмотрены упражнения для компьютерного практикума по курсу машинного обучения с подкреплением – активно развивающегося направления в искусственном интеллекте. В качестве основного языка программирования используется язык Python версии 3. При подготовке издания использовались следующие материалы и онлайн-ресурсы:

- библиотека OpenAI Gym [2];
- практический онлайн-курс по обучению с подкреплением от компании Yandex[1];
- блог Массимилиано Патачёла[2].

В учебно-методическом пособии рассмотрены программные реализации основных алгоритмов обучения с подкреплением: динамического программирования, вычисления функции полезности, итерации по стратегиям. Представлена программная схема реализации методов, основанных на полезности, и методов, основанных на стратегии. В каждом разделе предлагается решить несколько задач: дополнить код и провести эксперименты с полученной реализацией. Особое внимание уделено алгоритмам аппроксимации и реализации иерархических подходов.

Пособие предназначено для студентов старших курсов и аспирантов, изучающих методы искусственного интеллекта, машинное обучение и интеллектуальные робототехнические системы. Пособие рекомендуется использовать совместно с кратким курсом лекций по обучению с подкреплением [4] или с более подробной книгой по данной теме [5].

[1] `https://github.com/yandexdataschool/Practical_RL`

[2] `https://mpatacchiola.github.io/blog/2016/12/09/dissecting-reinforcement-learning.html`

1. Постановка задачи обучения с подкреплением

Обучение с подкреплением (RL) является направлением машинного обучения и изучает взаимодействие агента со средой. Агенту необходимо максимизировать долговременный выигрыш в данной среде. Агенту не сообщается сведений о правильности его действий, как в большинстве задач машинного обучения, вместо этого агент должен определить выгодные действия самостоятельно, применив их и оценив результат. Испытание действий и отсроченная награда являются основными отличительными признаками задачи RL (см. рис. 1.1).

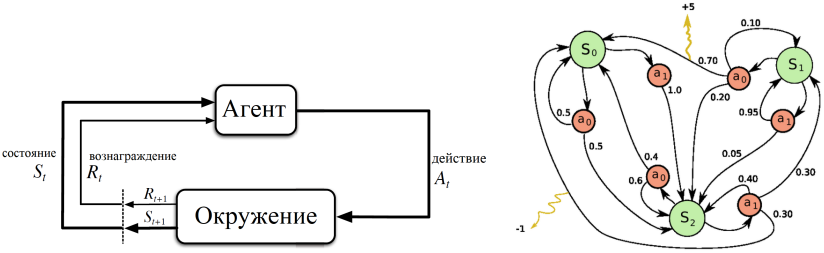

Рис. 1.1. Схема взаимодействия агента со средой (слева) и пример марковского процесса принятия решений (справа)

Основные составляющие модели RL:

- s_t – состояние среды в момент времени t;
- a_t – действие, совершаемое агентом в момент времени t;
- r_t – вознаграждение, получаемое агентом при совершении действия a_t;
- π – стратегия, отвечающая за выбор действия в конкретном состоянии.

В простейших моделях взаимодействие среды и агента представляется в виде *марковского процесса принятия решений* (MDP), где функция перехода определяется как $P(s'|s,a)$, что означает вероятность оказаться в состоянии s' при совершении действия a в состоянии s. Вознаграждение теперь определяется как $r(s,a,s')$.

Будем пользоваться стандартными средами, реализованными в библиотеке OpenAI Gym[3].

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import gym
# создаем окружение
env = gym.make("MountainCar-v0")
# рисуем картинку
plt.imshow(env.render('rgb_array'))
env.close()
```

§1.1. Интерфейс среды в OpenAI Gym

В библиотеке OpenAI Gym за моделирование среды отвечает класс *Env*. Основные методы класса *Env*:

- *reset()* – инициализация окружения, возвращает первое наблюдение;
- *render()* – визуализация текущего состояния среды;
- *step(a)* – выполнить в среде действие a и получить *new_obs* – новое наблюдение после выполнения действия *a*; *reward* – вознаграждение за выполненное действие *a*; *is_done* – *True*, если процесс завершился, *False* иначе; *info* – дополнительная информация.

```
obs0 = env.reset()
print("изначальное состояние среды:", obs0)
# выполняем действие 2
new_obs, reward, is_done, _ = env.step(2)
print("новое состояние:", new_obs,
      "вознаграждение", reward)
```

Наша цель состоит в том, чтобы тележка из среды *MountainCar* достигла флага. Модифицируйте код ниже для выполнения этого задания:

[3] https://gym.openai.com

```
def act(s):
    actions = {'left': 0, 'stop': 1, 'right': 2}
    # в зависимости от полученного состояния среды
    # выбираем действия так, чтобы тележка достигла флага
    # action = actions['left']
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    return action

# создаем окружение с ограничением на число шагов в 249
env = gym.wrappers.TimeLimit(
    gym.make("MountainCar-v0").unwrapped,
    max_episode_steps=250)
# проводим инициализацию и запоминаем начальное состояние
s = env.reset()
done = False
while not done:
    # выполняем действие, получаем s, r, done
    s, r, done, _ = env.step(act(s))
    # визуализируем окружение
    env.render()

env.close()
if s[0] > 0.47:
    print("Задание выполнено!")
else:
    raise NotImplementedError("""
    Исправьте функцию выбора действия!""")
```

§1.2. Вероятностный подход к RL

Пусть наша стратегия – это вероятностное распределение: $\pi(s, a) = P(a|s)$. Рассмотрим пример с задачей Taxi [1]. Для нее мы можем считать, что наша стратегия – это двумерный массив.

```
env = gym.make("Taxi-v2")
env.reset()
env.render()
n_states  = env.observation_space.n
n_actions = env.action_space.n
print("состояний:", n_states, "\nдействий: ", n_actions)
```

Создадим «равномерную» стратегию в виде двумерного массива с равномерным распределением по действиям и сгенерируем игровую сессию с такой стратегией.

```
policy = np.array(
    [[1./n_actions for _ in range(n_actions)]
     for _ in range(n_states)])

def generate_session(policy,t_max=10**4):
    states,actions = [],[]
    total_reward = 0.
    s = env.reset()
    for t in range(t_max):
        # Нужно выбрать действие с вероятностью,
        # указанной в стратегии
        # a =
        #~~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        new_s,r,done,info = env.step(a)
        # запоминаем состояния, действия и вознаграждение
        states.append(s)
        actions.append(a)
        total_reward += r

        s = new_s
        if done:
            break
    return states,actions,total_reward

s,a,r = generate_session(policy)
```

Наша задача – выделить лучшие действия и состояния, т. е. определить такие пары *состояние–действие*, при которых было бы лучшее вознаграждение.

```
def select_elites(states_batch, actions_batch,
                  rewards_batch, percentile=50):
    """
    Выбираем состояния и действия с заданным процентилем
    :param states_batch: states_batch[sess_i][t]
    :param actions_batch: actions_batch[sess_i][t]
    :param rewards_batch: rewards_batch[sess_i]

    :returns: elite_states, elite_actions - одномерные
    списки состояния и действия, выбранных сессий
    """
    # нужно найти порог вознаграждения по процентилю
    # reward_threshold =
    #~~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    # в соответствии с найденным порогом  отобрать
    # подходящие состояния и действия
    # elite_states =
    # elite_actions =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


    return elite_states,elite_actions

states_batch = [
    [1, 2, 3],  # game1
    [4, 2, 0, 2],  # game2
    [3, 1]  # game3
]
actions_batch = [
    [0, 2, 4],  # игра 1
    [3, 2, 0, 1],  # игра 2
    [3, 3]  # игра 3
]
rewards_batch = [
    3,  # игра 1
    4,  # игра 2
    5,  # игра 3
]

test_result_0 = select_elites(states_batch, actions_batch,
                              rewards_batch, percentile=0)
test_result_40 = select_elites(states_batch, actions_batch,
                              rewards_batch, percentile=30)
test_result_90 = select_elites(states_batch, actions_batch,
                              rewards_batch, percentile=90)
test_result_100 = select_elites(states_batch, actions_batch,
                              rewards_batch, percentile=100)

assert np.all(
    test_result_0[0] == [1, 2, 3, 4, 2, 0, 2, 3, 1]) \
       and np.all(
    test_result_0[1] == [0, 2, 4, 3, 2, 0, 1, 3, 3]), \
    "Для процентиля 0 необходимо выбрать все состояния " \
    "и действия в хронологическом порядке"

assert np.all(test_result_40[0] == [4, 2, 0, 2, 3, 1])\
   and np.all(test_result_40[1] == [3, 2, 0, 1, 3, 3]), \
    "Для процентиля 30 необходимо выбрать " \
    "состояния/действия из [3:]"
assert np.all(test_result_90[0] == [3, 1]) and \
```

```
       np.all(test_result_90[1] == [3, 3]), \
    "Для процентиля 90 необходимо выбрать состояния " \
    "действия одной игры"
assert np.all(test_result_100[0] == [3, 1]) and \
       np.all(test_result_100[1] == [3, 3]), \
    "Проверьте использование знаков: >=,  >. " \
    "Также проверьте расчет процентиля"
print("Тесты пройдены!")
```

Теперь переходим к написанию обновляющейся стратегии.

```
def update_policy(elite_states,elite_actions):
    """
    обновление стратегии
    policy[s_i,a_i] ~ #[вхождения  si/ai
    в лучшие states/actions]
    :param elite_states:  список состояний
    :param elite_actions: список действий
    """
    new_policy = np.zeros([n_states,n_actions])
    for state in range(n_states):
        # обновялем стратегию - нормируем новые частоты
        # действий и не забываем про невстречающиеся
        # состояния
        # new_policy[state, a] =
        #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    return new_policy

elite_states, elite_actions = (
    [1, 2, 3, 4, 2, 0, 2, 3, 1],
    [0, 2, 4, 3, 2, 0, 1, 3, 3])

new_policy = update_policy(elite_states, elite_actions)

assert np.isfinite(
    new_policy).all(), "Стратегия не должна содержать " \
                       "NaNs или +-inf. Проверьте " \
                       "деление на ноль. "
assert np.all(
    new_policy >= 0), "Стратегия не должна содержать " \
                      "отрицательных вероятностей "
assert np.allclose(new_policy.sum(axis=-1),
                   1), "Суммарная\ вероятность действий"\
                       "для состояния должна равняться 1"
reference_answer = np.array([
    [1., 0., 0., 0., 0.],
    [0.5, 0., 0., 0.5, 0.],
```

```
    [0., 0.33333333, 0.66666667, 0., 0.],
    [0., 0., 0., 0.5, 0.5]])
assert np.allclose(new_policy[:4, :5], reference_answer)
print("Тесты пройдены!")
```

Визуализируем наш процесс обучения и также будем измерять распределение получаемых за сессию вознаграждений.

```
from IPython.display import clear_output

def show_progress(rewards_batch, log, reward_range=None):
    """
    Удобная функция, которая отображает прогресс обучения.
    Здесь нет <<крутой>> математики, только графики.
    """

    if reward_range is None:
        reward_range = [-990, +10]
    mean_reward = np.mean(rewards_batch)
    threshold = np.percentile(rewards_batch, percentile)
    log.append([mean_reward, threshold])

    clear_output(True)
    print("mean reward = %.3f, threshold=%.3f" % (
        mean_reward,
        threshold))
    plt.figure(figsize=[8, 4])
    plt.subplot(1, 2, 1)
    plt.plot(list(zip(*log))[0], label='Mean rewards')
    plt.plot(list(zip(*log))[1],
             label='Reward thresholds')
    plt.legend(loc=4)
    plt.grid()

    plt.subplot(1, 2, 2)
    plt.hist(rewards_batch, range=reward_range)
    plt.vlines([np.percentile(rewards_batch, percentile)],
               [0], [100], label="percentile",
               color='red')
    plt.legend(loc=1)
    plt.grid()

    plt.show()

policy = np.ones([n_states,n_actions])/n_actions
n_sessions = 250  # количество сессий для сэмплирования
percentile = 50  # процентиль
learning_rate = 0.5
log = []
```

```
for i in range(100):
    # генерируем n_sessions сессий
    # sessions = []
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    states_batch,actions_batch,rewards_batch = \
        zip(*sessions)
    # отбираем лучшие действия и состояния ###
    # elite_states, elite_actions =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


    # обновляем стратегию
    # new_policy =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    policy = learning_rate*new_policy + \
                    (1-learning_rate)*policy
    # визуализация обучения
    show_progress(rewards_batch,log)
```

§1.3. Аппроксимация вероятностного подхода

Попробуем заменить метод обновления вероятностей на нейронную сеть. Будем тестировать нашего нового агента с помощью известной задачи перевернутого маятника с непрерывным множеством действий.

```
env = gym.make("CartPole-v0").env
env.reset()
n_actions = env.action_space.n

plt.imshow(env.render("rgb_array"))
env.close()

# создаем агента
from sklearn.neural_network import MLPClassifier
# создаем полносвязную сеть с двумя слоями по 20 нейронов,
# активация tanh
```

```
# agent =
#~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
raise NotImplementedError
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

agent.fit([env.reset()]*n_actions, range(n_actions))
env.reset()

def generate_session(t_max=1000):

    states,actions = [],[]
    total_reward = 0
    s = env.reset()

    for t in range(t_max):

        # предсказываем вероятности действий по сети
        # и выбираем одно действие
        # probs =
        # a =
        #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        new_s,r,done,info = env.step(a)

        #record sessions like you did before
        states.append(s)
        actions.append(a)
        total_reward+=r

        s = new_s
        if done: break
    return states,actions,total_reward

n_sessions = 100
percentile = 70
log = []

for i in range(100):
    # генерируем n_sessions сессий
    # sessions = [<gen a list>]
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    states_batch,actions_batch,rewards_batch =\
    map(np.array,zip(*sessions))

    # отбираем лучшие действия и состояния
```

```
    # elite_states, elite_actions =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    # обновляем стратегию для предсказания
    # elite_actions(y) из elite_states(X)
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    r_range =[0,np.max(rewards_batch)]
    show_progress(rewards_batch, log, r_range )

    if np.mean(rewards_batch) > 190:
        print("Принято!")
        break

# монитор для сессий
import gym.wrappers
env = gym.wrappers.Monitor(gym.make("CartPole-v0"),
                           directory="videos",force=True)
sessions = [generate_session() for _ in range(100)]
env.close()
# можем посмотреть видео
from IPython.display import HTML
import os

video_names = list(filter(lambda s:s.endswith(".mp4"),
                          os.listdir("./videos/")))
HTML("""
<video width="640" height="480" controls>
  <source src="{}" type="video/mp4">
</video>
""".format("./videos/"+video_names[-1]))
# вместо последнего можно выбрать любой индекс
```

2. Динамическое программирование в обучении с подкреплением

Рассмотрим алгоритм итерации по оценкам полезностей состояния $V(s)$ (Value Iteration):

$$V_{(i+1)}(s) = \max_a \sum_{s'} P(s'|s,a) \cdot [r(s,a,s') + \gamma V_i(s')].$$

На основе оценки V_i можно посчитать функцию оценки полезности действия $Q_i(a,s)$:

$$Q_i(s,a) = \sum_{s'} P(s'|s,a) \cdot [r(s,a,s') + \gamma V_i(s')],$$

$$V_{(i+1)}(s) = \max_a Q_i(s,a).$$

Зададим напрямую модель MDP (см. рис. 1.1):

```
transition_probs = {
  's0':{
    'a0': {'s0': 0.5, 's2': 0.5},
    'a1': {'s2': 1}
  },
  's1':{
    'a0': {'s0': 0.7, 's1': 0.1, 's2': 0.2},
    'a1': {'s1': 0.95, 's2': 0.05}
  },
  's2':{
    'a0': {'s0': 0.4, 's1': 0.6},
    'a1': {'s0': 0.3, 's1': 0.3, 's2':0.4}
  }
}
rewards = {
  's1': {'a0': {'s0': +5}},
  's2': {'a1': {'s0': -1}}
}

from mdp import MDP
import numpy as np
mdp = MDP(transition_probs, rewards, initial_state='s0')

print("all_states =", mdp.get_all_states())
print("possible_actions('s1') = ",
      mdp.get_possible_actions('s1'))
print("next_states('s1', 'a0') = ",
      mdp.get_next_states('s1', 'a0'))
print("reward('s1', 'a0', 's0') = ",
      mdp.get_reward('s1', 'a0', 's0'))
print("transition_prob('s1', 'a0', 's0') = ",
      mdp.get_transition_prob('s1', 'a0', 's0'))
```

§2.1. Вычисление функций полезности

Реализуем итерационное вычисление функций V и Q и применим их для заданного вручную MDP. Вначале вычисляем оценку состояния–действия:

$$Q_i(s,a) = \sum_{s'} P(s'|s,a) \cdot [r(s,a,s') + \gamma V_i(s')].$$

```
def get_action_value(mdp, state_values, state, action,
                     gamma):
    """ Вычисляем Q(s,a) по формуле выше """
    # вычисляем оценку состояния
    # Q =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    return Q

test_Vs = {s : i for i, s in
    enumerate(sorted(mdp.get_all_states()))}
assert np.allclose(
    get_action_value(mdp, test_Vs, 's2', 'a1', 0.9), 0.69)
```

Теперь оцениваем полезность самого состояния:

$$V_{(i+1)}(s) = \max_a \sum_{s'} P(s'|s,a) \cdot [r(s,a,s') + \gamma V_i(s')] = \max_a Q_i(s,a).$$

```
def get_new_state_value(mdp, state_values, state, gamma):
    """ Считаем следующее V(s) по формуле выше."""
    if mdp.is_terminal(state):
        return 0
    # V =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    return V

test_Vs_copy = dict(test_Vs)
assert np.allclose(
    get_new_state_value(mdp, test_Vs, 's0', 0.9), 1.8)
```

Теперь создаем основной цикл итерационной оценки полезности состояний с критерием остановки, который проверяет величину изменения оценки.

```
def value_iteration(mdp, state_values=None,
    gamma = 0.9, num_iter = 1000, min_difference = 1e-5):
    """ выполняет num_iter шагов итерации по значениям"""
    # инициализируем V(s)
    state_values = state_values or \
    {s : 0 for s in mdp.get_all_states()}

    for i in range(num_iter):
        # Вычисляем новые полезности состояний,
        # используя функции, определенные выше.
        # Должен получиться словарь {s: new_V(s)}
        # new_state_values =
        #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        assert isinstance(new_state_values, dict)

        # Считаем разницу
        # diff =
        #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        print("iter %4i | diff: %6.5f | V(start): %.3f "%
          (i, diff, new_state_values[mdp._initial_state]))

        state_values = new_state_values
        if diff < min_difference:
            print("Принято! Алгоритм сходится!")
            break

    return state_values

state_values = value_iteration(mdp,
        num_iter = 100, min_difference = 0.001)

print("Final state values:", state_values)
assert abs(state_values['s0'] - 8.032)  < 0.01
```

Имея вычисленные полезности и зная модель переходов, легко найти оптимальную стратегию:

$$\pi^*(s) = \arg\max_a \sum_{s'} P(s'|s,a) \cdot [r(s,a,s') + \gamma V_i(s')] = \arg\max_a Q_i(s,a).$$

```
def get_optimal_action(mdp, state_values, state,
                       gamma=0.9):
    """ Finds optimal action using formula above. """
    if mdp.is_terminal(state): return None

    actions = mdp.get_possible_actions(state)
    # выбираем лучшее действие
    # i =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    return actions[i]

assert get_optimal_action(mdp,
       state_values, 's0', gamma=0.9) == 'a1'
```

§2.2. Эксперименты со средой FrozenLake

Теперь проверим работу итерации по ценностям на классической задаче FrozenLake и визуализируем нашу стратегию. Проведем эксперименты с различными вариантами окружения.

```
from mdp import FrozenLakeEnv
mdp = FrozenLakeEnv(slip_chance=0)

mdp.render()
state_values = value_iteration(mdp)

def draw_policy(mdp, state_values, gamma=0.9):
    """функция визуализации стратегии"""
    plt.figure(figsize=(3, 3))
    h, w = mdp.desc.shape
    states = sorted(mdp.get_all_states())
    V = np.array([state_values[s] for s in states])
    Pi = {
        s: get_optimal_action(mdp, state_values, s, gamma)
        for s in states}
    plt.imshow(V.reshape(w, h),
               cmap='gray', interpolation='none',
               clim=(0, 1))
    ax = plt.gca()
    ax.set_xticks(np.arange(h) - .5)
    ax.set_yticks(np.arange(w) - .5)
    ax.set_xticklabels([])
    ax.set_yticklabels([])
```

```
    Y, X = np.mgrid[0:4, 0:4]
    a2uv = {'left': (-1, 0), 'down': (0, -1),
            'right': (1, 0), 'up': (-1, 0)}
    for y in range(h):
        for x in range(w):
            plt.text(x, y, str(mdp.desc[y, x].item()),
                     color='g', size=12,
                     verticalalignment='center',
                     horizontalalignment='center',
                     fontweight='bold')
            a = Pi[y, x]
            if a is None: continue
            u, v = a2uv[a]
            plt.arrow(x, y, u * .3, -v * .3,
                      color='m', head_width=0.1,
                      head_length=0.1)
    plt.grid(color='b', lw=2, ls='-')
    plt.show()

from IPython.display import clear_output
from time import sleep
import matplotlib.pyplot as plt

mdp = FrozenLakeEnv(map_name='8x8',slip_chance=0.1)
state_values = {s : 0 for s in mdp.get_all_states()}

for i in range(30):
    clear_output(True)
    print("after iteration %i"%i)
    state_values = value_iteration(mdp,
                               state_values, num_iter=1)
    draw_policy(mdp, state_values)
    sleep(0.5)

# Получаем среднее вознаграждение агента
mdp = FrozenLakeEnv(slip_chance=0.2, map_name='8x8')
state_values = value_iteration(mdp)

total_rewards = []
for game_i in range(1000):
    s = mdp.reset()
    rewards = []
    for t in range(100):
        # выполняем оптимальное действие в окружении
        # s, r, done, _ =
        #^^^^^^^^ Ваш код здесь ^^^^^^^^^^^
        raise NotImplementedError
        #^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

        rewards.append(r)
```

```
        if done: break
    total_rewards.append(np.sum(rewards))

print("Среднее вознаграждение:", np.mean(total_rewards))
assert(0.6 <= np.mean(total_rewards) <= 0.8)
print("Принято!")
```

§2.3. Алгоритм итерации по стратегиям

Теперь рассмотрим следующий *алгоритм итерации по стратегиям* (PI):

1. Инициализация π_0 (например, случайный выбор действий).
2. For $n = 0, 1, 2, \ldots$
3. Вычисляем функцию V^{π_n}.
4. Используя V^{π_n}, рассчитываем функцию Q^{π_n}.
5. Рассчитываем новую стратегию $\pi_{n+1}(s) = \operatorname{argmax}_a Q^{\pi_n}(s, a)$.

PI использует оценку полезности состояния в качестве промежуточного шаг. Вначале оценим полезности, используя текущую стратегию:

$$V^{\pi}(s) = \sum_{s'} P(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')].$$

Мы будем искать точное решение, хотя могли использовать и предыдущий итерационный подход. Для этого будем решать систему линейных уравнений относительно $V^{\pi}(s_i)$ с помощью *np.linalg.solve*.

```
from numpy.linalg import solve

def compute_vpi(mdp, policy, gamma):
    """
    Считем V^pi(s) для всех состояний согласно стратегии.
    :param policy: словарь состояние->действие {s : a}
    :returns: словарь {state : V^pi(state)}
    """
    states = mdp.get_all_states()
    A, b = [], []
    for i, state in enumerate(states):
        if state in policy:
            a = policy[state]
            # формируем матрицу A (... A.append(...))
            #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
            raise NotImplementedError
```

```
            #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            # и вектор b (b.append(...))
            #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
            raise NotImplementedError
            #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        else:
            # формируем матрицу A (... A.append(...))
            #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
            raise NotImplementedError
            #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            # вектор b (b.append(...))
            #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
            raise NotImplementedError
            #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    A = np.array(A)
    b = np.array(b)

    values = solve(A, b)

    state_values = {states[i] : values[i]
                    for i in range(len(states))}
    return state_values

transition_probs = {
    's0': {
        'a0': {'s0': 0.5, 's2': 0.5},
        'a1': {'s2': 1}
    },
    's1': {
        'a0': {'s0': 0.7, 's1': 0.1, 's2': 0.2},
        'a1': {'s1': 0.95, 's2': 0.05}
    },
    's2': {
        'a0': {'s0': 0.4, 's1': 0.6},
        'a1': {'s0': 0.3, 's1': 0.3, 's2': 0.4}
    }
}
rewards = {
    's1': {'a0': {'s0': +5}},
    's2': {'a1': {'s0': -1}}
}
mdp = MDP(transition_probs, rewards, initial_state='s0')

gamma = 0.9  # коэффициент дисконтирования для MDP

test_policy = {
```

```
    s: np.random.choice(mdp.get_possible_actions(s))
    for s in mdp.get_all_states()}
new_vpi = compute_vpi(mdp, test_policy, gamma)

print(new_vpi)
assert type(new_vpi) is dict, \
    "функция compute_vpi должна возвращать словарь \
    {состояние s : V^pi(s) }"
```

Теперь обновляем стратегию на основе новых значений полезностей.

```
def compute_new_policy(mdp, vpi, gamma):
    """
    Рассчитываем новую стратегию
    :param vpi: словарь {state : V^pi(state) }
    :returns: словарь {state : оптимальное действие}
    """
    Q = {}
    for state in mdp.get_all_states():
        Q[state] = {}
        for a in mdp.get_possible_actions(state):
            values = []
            for next_state in mdp.get_next_states(state,
                                                  a):
                r = mdp.get_reward(state, a, next_state)
                p = mdp.get_transition_prob(state, a,
                                            next_state)
                # values.append(...)
                #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
                raise NotImplementedError
                #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            Q[state][a] = sum(values)

    policy = {}
    for state in mdp.get_all_states():
        actions = mdp.get_possible_actions(state)
        if actions:
            # выбираем оптимальное действие в state
            # policy[state] = ...
            #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
            raise NotImplementedError
            #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    return policy

new_policy = compute_new_policy(mdp, new_vpi, gamma)

print(new_policy)
```

```
assert type(new_policy) is dict, \
"функция compute_new_policy должна возвращать словарь \
{состояние s: оптимальное действие}"
```

Собираем ранее определенные функции в единый цикл. Проводим эксперименты на уже знакомом нам окружении FrozenLake.

```
def policy_iteration(mdp, policy=None, gamma = 0.9,
                num_iter = 1000, min_difference = 1e-5):
    """
    Запускаем цикл итерации по стратегиям
    Если стратегия не определена, задаем случайную
    """
    for i in range(num_iter):
        if not policy:
            policy = {}
            for s in mdp.get_all_states():
                if mdp.get_possible_actions(s):
                    policy[s] = np.random \
                    .choice(mdp.get_possible_actions(s))
        # state_values =
        #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        # policy =
        #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    return state_values, policy

mdp = FrozenLakeEnv(slip_chance=0.1)
state_values, policy = policy_iteration(mdp)
total_rewards = []
for game_i in range(1000):
    s = mdp.reset()
    rewards = []
    for t in range(100):
        s, r, done, _ = mdp.step(policy[s])
        rewards.append(r)
        if done: break
    total_rewards.append(np.sum(rewards))
print("average reward: ", np.mean(total_rewards))
assert(0.8 <= np.mean(total_rewards) <= 0.95)
print("Принято!")
```

3. Алгоритм Q-обучения

Одним из наиболее популярных алгоритм обучения на основе временных различий является *Q-обучение*. Агент, который принимает решения на основе Q-функции, не требует модель для обучения и выбора действий, т. е. такой агент также свободен от модели (model-free), как и *TD*-агент. Уравнение Беллмана для значения Q-функции в равновесии записывается как

$$Q(s,a) = r(s) + \gamma \sum_{s}^{'} T(s,a,s') \max_{a'} Q(a',s').$$

Уравнение для итерационного обновления значений Q-функции выглядит следующим образом:

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

§3.1. Класс QLearningAgent

Оформим итерационное обновление значений Q-функции в виде класса QLearningAgent.

```
import random, math

import numpy as np
from collections import defaultdict


class QLearningAgent():
    """
    Q-Learning агент

    Замечание: избегайте прямого использования
    self._q_values, для этого определены
    функции: get_q_value, set_q_value
    """

    def __init__(self, alpha, epsilon, discount,
                 get_legal_actions):
        self.get_legal_actions = get_legal_actions
        self._q_values = \
            defaultdict(lambda: defaultdict(lambda: 0))
        self.alpha = alpha
        self.epsilon = epsilon
        self.discount = discount
```

```
    def get_q_value(self, state, action):
        return self._q_values[state][action]

    def set_q_value(self, state, action, value):
        self._q_values[state][action] = value
```

Добавим нашему агенту возможность вычислять оценки V.

```
def get_value(self, state):
    """
      Возвращает значение функции полезности,
      рассчитанной по Q[state, action],
    """
    possible_actions = self.get_legal_actions(state)

    # value =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    return value

QLearningAgent.get_value = get_value
```

Стратегия агента будет заключаться в выборе лучшего, в соответствии с оценками Q, действия.

```
def get_policy(self, state):
    """
      Выбирает лучшее действие согласно стратегии.
    """
    possible_actions = self.get_legal_actions(state)

    # выбираем лучшее действие согласно стратегии
    #best_action =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    return best_action

QLearningAgent.get_policy = get_policy
```

Для конкретной ситуации мы будем выбирать действие, используя ϵ-жадный подход для сохранения возможности исследования среды.

```
def get_action(self, state):
    """
      Выбирает действие, предпринимаемое в данном
      состоянии, включая исследование.
      С вероятностью self.epsilon берем случайное
      действие, иначе - действие согласно стратегии
      (self.get_policy)
    """
    possible_actions = self.get_legal_actions(state)

    # выбираем действие, используя eps-greedy подход
    # action =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    return action

QLearningAgent.get_action = get_action

def update(self, state, action, next_state, reward):
    """
      функция Q-обновления
    """
    # выполняем Q-обновление,
    # используем методы getQValue и setQValue
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

QLearningAgent.update = update
```

Проводим эксперименты с агентом на задаче Taxi.

```
import gym
env = gym.make("Taxi-v2")

n_actions = env.action_space.n

def play_and_train(env, agent, t_max=10**4):
    """функция запускает полную игру,
    используя стратегию агента (agent.get_action(s)),
    выполняет обновление агента (agent.update(...))
    и возвращает общее вознаграждение
    """
    total_reward = 0.0
    s = env.reset()

    for t in range(t_max):
```

```
        # выбираем действие
        # a =
        #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        next_s, r, done, _ = env.step(a)

        # выполняем обновление стратегии
        # agent.update()
        #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        s = next_s
        total_reward +=r
        if done:
            break

    return total_reward

import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import clear_output

agent = QLearningAgent(alpha=0.5, epsilon=0.1, discount=0.9,
                       get_legal_actions=lambda s: range(n_actions))

assert 'get_policy' in dir(agent)
rewards = []
for i in range(5000):
    rewards.append(play_and_train(env, agent))

    if i % 100 == 0:
        clear_output(True)
        print('eps =', agent.epsilon,
              'mean reward =', np.mean(rewards[-10:]))
        print("alpha=", agent.alpha)
        plt.plot(rewards)
        plt.show()
```

§3.2. Q-обучение с непрерывным множеством состояний

Рассмотрим задачу о перевернутом маятнике CartPole. Это окружение имеет непрерывное множество состояний – попробуем их сгруппировать. Для этого попытаемся использовать ‘round(x, n_digits)‘ для округления действительных чисел.

```
env = gym.make("CartPole-v0")
n_actions = env.action_space.n

print("начальное состояние: %s" % (env.reset()))
plt.imshow(env.render('rgb_array'))

env.close()
```

Оценим распределение наблюдений – проведем несколько эпизодов и запомним встретившиеся состояния.

```
all_states = []
for _ in range(1000):
    all_states.append(env.reset())
    done = False
    while not done:
        action = env.action_space.sample()
        s, r, done, _ = env.step(action)
        all_states.append(s)
        if done:
            break

all_states = np.array(all_states)

for obs_i in range(env.observation_space.shape[0]):
    plt.hist(all_states[:, obs_i], bins=20)
    plt.show()
```

Теперь создадим обертку для окружения, которая проводит бинаризацию состояний.

```
from gym.core import ObservationWrapper

class Binarizer(ObservationWrapper):

    def _observation(self, state):

        # state = <round state to some amount digits.>
        # подсказка: используйте round(x,n_digits)
        # необходимо выбрать разные параметры n_digits
        # для каждой размерности
        #^^^^^^^^ Ваш код здесь ^^^^^^^^^^^
        raise NotImplementedError
        #^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

        return tuple(state)

env = Binarizer(gym.make("CartPole-v0"))
```

```
all_states = []
for _ in range(1000):
    all_states.append(env.reset())
    done = False
    while not done:
        action = env.action_space.sample()
        s, r, done, _ = env.step(action)
        all_states.append(s)
        if done:
            break

all_states = np.array(all_states)

for obs_i in range(env.observation_space.shape[0]):

    plt.hist(all_states[:, obs_i], bins=20)
    plt.show()
```

Теперь запустим процесс обучения нашего Q-агента. Если бинаризация очень грубая, агент может не обучаться. Если бинаризация слишком точная, сходимость процесса обучения может занять слишком большое количество шагов. Размерность состояний в диапазоне от 10^3–10^4 является оптимальной. Успешным является агент, получающий вознаграждение $>= 50$.

```
agent = QLearningAgent(alpha=0.7, epsilon=0.25,
                       discount=0.99,
                       get_legal_actions=lambda s: range(
                           n_actions))

rewards = []
for i in range(10001):
    rewards.append(play_and_train(env, agent))

    # опционально: добавьте уменьшение eps
    if i % 1000 == 0:
        clear_output(True)
        print('eps =', agent.epsilon, 'mean reward =',
              np.mean(rewards[-100:]))
        plt.plot(rewards)
        plt.show()
```

4. Аппроксимация Q-функции

В данном разделе будет использоваться библиотека tensorflow [3] для настройки аппроксиматора (для обучения нейронной сети). Для реализации дифференцируемого графа вычислений можно использовать и любую другую библиотеку (pytorch и др.).

```
import gym
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Будем тестировать наши модели на классической задаче с перевернутым маятником.

```
env = gym.make("CartPole-v0").env
env.reset()
n_actions = env.action_space.n
state_dim = env.observation_space.shape

plt.imshow(env.render("rgb_array"))

env.close()
```

§4.1. Построение нейросетевого аппроксиматора

Так как описание состояния в задаче с маятником представляет собой не «сырые» признаки, а уже семантически интерпретируемые (координаты, углы), нам не нужна для начала сложная архитектура, поэтому начнем с изображенной на рис. 4.1.

Первое время будем использовать только полносвязные слои (L. Dense) и линейные активационные функции. Сигмоиды и другие похожие функции активации не будут работать с ненормализованными входными данными.

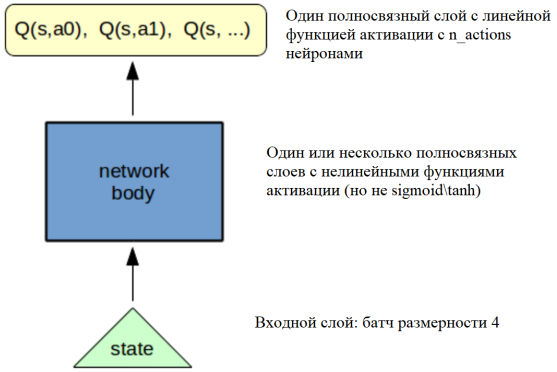

Рис. 4.1. Архитектура сети

```
import tensorflow as tf
import keras
import keras.layers as L
tf.reset_default_graph()
sess = tf.InteractiveSession()
keras.backend.set_session(sess)
```

```
network = keras.models.Sequential()
network.add(L.InputLayer(state_dim))
# строим сеть!
#~~~~~~~~ Ваш код здесь ~~~~~~~~~~
raise NotImplementedError
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

import random
def get_action(state, epsilon=0):
    """
    сэмплируем (eps greedy) действие
    """
    q_values = network.predict(state[None])[0]

    ### Ваш код здесь - нужно выбрать действия eps-жадно
    # action =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    return action
```

```
assert network.output_shape == (None, n_actions), \
    "убедитесь, что стратегия переводит \
    s -> [Q(s,a0), ..., Q(s, a_last)]"
assert network.layers[-1].activation == \
       keras.activations.linear, \
    "убедитесь, что вы предсказываете q без нелинейности"
# проверяем исследование
s = env.reset()
assert np.shape(get_action(s)) == (), \
    "убедитесь, что возвращаете одно действие"
for eps in [0., 0.1, 0.5, 1.0]:
    na = n_actions
    st = np.bincount([get_action(s, epsilon=eps) \
                      for i in range(10000)],
                     minlength=na)
    ba = st.argmax()
    assert abs(
        st[ba] - 10000 * (1 - eps + eps / na)) < 200
    for oa in range(na):
        if oa != ba:
            assert abs(st[oa] - 10000 * (eps / na)) < 200
    print('e=%.1f tests passed' % eps)
```

§4.2. Q-обучение через градиентный спуск

Теперь будем приближать Q-функцию агента, минимизируя *TD*-функцию потерь:

$$L = \frac{1}{N}\sum_i (Q_\theta(s,a) - [r(s,a) + \gamma \cdot \max_{a'} Q_-(s',a')])^2.$$

Ключевая особенность этой функции состоит в том, что мы используем $Q_-(s',a')$. Эта та же самая функция, что и Q_θ, являющаяся выходом нейронной сети, но при обучении графа вычислений мы не пропускаем через эти слои градиенты. Для этого используется функция *tf.stop_ gradient*.

```
# Создаем placeholders для <s, a, r, s'>,
# а также индикатор окончания эпизода (is_done = True)
states_ph = tf.placeholder('float32',
                           shape=(None,) + state_dim)
actions_ph = tf.placeholder('int32', shape=[None])
rewards_ph = tf.placeholder('float32', shape=[None])
next_states_ph = tf.placeholder('float32',
                           shape=(None,) + state_dim)
is_done_ph = tf.placeholder('bool', shape=[None])
```

```
# получаем q для всех действий в текущем состоянии
predicted_qvalues = network(states_ph)

# получаем q-values для выбранного действия
predicted_qvalues_for_actions =\
tf.reduce_sum(
predicted_qvalues * tf.one_hot(actions_ph, n_actions),
    axis=1)

gamma = 0.99

# применяем сеть для получения q-value для next_states_ph
# predicted_next_qvalues =
#~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
raise NotImplementedError
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# вычисляем V*(next_states)
# по предсказанным следующим q-values
# next_state_values =
#~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
raise NotImplementedError
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Вычисляем target q-values для функции потерь
# target_qvalues_for_actions =
#~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
raise NotImplementedError
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# для последнего действия используем
# упрощенную формулу Q(s,a) = r(s,a)
target_qvalues_for_actions =\
tf.where(is_done_ph, rewards_ph,
         target_qvalues_for_actions)

### среднеквадратичная функция потерь stop_gradient
# loss =
#~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
raise NotImplementedError
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# применяем AdamOptimizer
train_step = tf.train.AdamOptimizer(1e-4).minimize(loss)
```

Проведем ряд проверок для того, чтобы убедиться в правильной конфигурации графа вычислений.

```
assert tf.gradients(loss, \
    [predicted_qvalues_for_actions])[0] is not None, \
"убедитесь, что обновление выполняется\
только для выбранного действия"
assert tf.gradients(loss, \
    [predicted_next_qvalues])[0] is None, \
"убедитесь, что вы не распространяете градиент Q_(s',a')"
assert predicted_next_qvalues.shape.ndims == 2, \
"убедитесь, что вы предсказываете q для всех действий\
следующего состояния"
assert next_state_values.shape.ndims == 1, \
"убедитесь, что вы вычислили V(s') как максимум\
только по оси действий, а не по всем осям"
assert target_qvalues_for_actions.shape.ndims == 1, \
"что-то не так с целевыми q-значениями,\
они должны быть векторами"
```

§4.3. Эксперименты и результаты

```
def generate_session(t_max=1000, epsilon=0, train=False):
    """генерация сессии"""
    total_reward = 0
    s = env.reset()

    for t in range(t_max):
        a = get_action(s, epsilon=epsilon)
        next_s, r, done, _ = env.step(a)

        if train:
            sess.run(train_step,{
                states_ph: [s], actions_ph: [a],
                rewards_ph: [r], next_states_ph: [next_s],
                is_done_ph: [done]
            })

        total_reward += r
        s = next_s
        if done: break

    return total_reward

epsilon = 0.8

for i in range(100):
    session_rewards = [generate_session(epsilon=epsilon,
                        train=True) for _ in range(500)]
    print("epoch #{}\tmean r = {:.3f}\tepsilon = {:.3f}"
```

```
        .format(i, np.mean(session_rewards), epsilon))

    epsilon *= 0.95
    epsilon = max(0.1, epsilon)
    assert epsilon >= 1e-4, \
    "убедитесь, что epsilon не становится < 0"
    if np.mean(session_rewards) > 300:
        print ("Принято!")
        break
```

Комментарии к получаемым результатам:

- *mean_reward* – это среднее вознаграждение за эпизод. В случае корректной реализации, этот показатель будет низким первые 5 эпох и только затем будет возрастать, и сойдется на 20–30 эпох в зависимости от архитектуры сети.
- Если сеть не достигает нужных результатов к концу цикла, попробуйте увеличить число нейронов в скрытом слое или поменяйте ϵ.
- Переменная *epsilon* обеспечивает стремление агента исследовать среду. Можно искусственно изменять малые значения ϵ при низких результатах на 0.1–0.5.

5. Иерархический подход к обучению с подкреплением

В этом разделе нашей задачей будет создание набора умений (мета-действий), каждое из которых должно быть направлено на достижение определенных состояний в задаче Taxi. Для обучения мы будем использовать класс QLearningAgent, реализованный в прошлом разделе.

```
# импортируем файлы и создаем окружение
import gym
import random
import numpy as np

environment = gym.make('Taxi-v2')
environment.render()

# импортируем классс Q-агента из прошлого занятия
from q_agent import QLearningAgent
```

§5.1. Создание иерархической среды

Разберемся как реализована среда Taxi[4]. Создадим четыре окружения, аналогичных Taxi, в которых целью агента будет достижение одной из точек: R, G, B, Y соответственно.

```
class TaxiStepWrapper(gym.Wrapper):
    def __init__(self, env, target_id, target_reward):
        super().__init__(env)
        self._target = target_id
        self._target_reward = target_reward

    def _step(self, action):
        # получаем параметры (state, reward, _, obs),
        # которые передает среда, используя метод step;
        # проверяем является ли состояние завершающим
        # для нашего модифицированного окружения;
        # изменяем вознаграждение (reward)
        # и флаг завершения эпизода (is_done);
        # за каждое действие будем давать вознаграждение -1,
        # за достижение цели - self._target_reward
        #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        return state, reward, is_done, obs
```

[4] https://github.com/openai/gym/blob/master/gym/envs/toy_text/taxi.py

Проверим нашу обертку (wrapper), используя случайную стратегию. Порядок точек должен быть R, G, Y, B.

```
for target in range(4):
    # создаем окружение с заданным целевым состоянием
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


    # применяем случайную стратегию,
    # пока эпизод не завершится
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


    wrapped_env.render()
    print("state:{s} reward:{r}\n".format(**locals()))

# воспользуемся методом play_and_train,
# который мы реализовали на прошлом семинаре
def play_and_train(env, agent, t_max=10 ** 4):
    total_discounted_reward = 0.0
    s = env.reset()
    for t in range(t_max):
        a = agent.get_action(s)
        next_s, r, done, _ = env.step(a)
        agent.update(s, a, next_s, r)
        s = next_s
        total_discounted_reward += r
        if done:
            break
    return total_discounted_reward
```

§5.2. Обучение умениям

Вначале обучим агентов на созданных нами окружениях. Создадим упрощенный вариант умений: каждое умение будет иметь стратегию, множество начальных состояний и множество конечных состояний.

```
n_actions = environment.action_space.n

# параметры, которые будут использовать агенты
```

```
params = {"alpha": 0.1, "epsilon": 0.1,
"gamma": 0.99, "get_legal_actions": lambda s: range(4)}

# создаем агентов
agents_for_options = [QLearningAgent(**params)\
                      for _ in range(4)]

for index in range(4):
    # создаем окружение с заданным целевым состоянием,
    # используя созданные окружения, обучаем агентов
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# реализуем класс умений
class Option:
    def __init__(self, policy, termination_prob, initial):
        self.policy = policy
        self.termination_prob = termination_prob
        self.initial_states = initial

    def can_start(self, state):
        return state in self.initial_states

    def terminate(self, state):
        return random.random() <= self.termination_prob[
            state]

    def get_action(self, state):
        return self.policy.get_action(state)

options = []
for index, agent in enumerate(agents_for_options):
    # Создаем словарь termination_prob, в котором каждому
    # состоянию  нужно задать вероятность завершения
    # умения. В нашем случае зададим 1.0 или 0.0
    # в зависимости от состояния.
    # Создаем множество initial, добавляем в него
    # состояния, из которых умение может быть
    # вызвано (все кроме целевых)
    termination_prob = {}
    initial_states = set()
    termination_states = set()
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


    options.append(Option(policy=agent, \
    termination_prob=termination_prob, initial=initial))
```

Напишем функцию, которая будет запускать умение и возвращать дисконтированное вознаграждение, опираясь на число совершенных действий:

$$R = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{t-1} r_t.$$

```
def apply_option(option, gamma, env, debug=False):
    reward = 0
    steps = 0

    if not option.can_start(state):
        raise KeyError

    # Взаимодействуем со средой, пока умение или окружение
    # не завершится, считаем дисконтированное
    # вознаграждение reward (используем steps),
    # также добавим render окружение при флаге debug
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    # state, reward, is_done, obs
    return s, reward, d, obs

# проверим работу метода
env = gym.make('Taxi-v2')
s = env.reset()

r = apply_option(options[0], 0.99, env, debug=True)
```

Кажется, что все хорошо, но мы забыли рассмотреть вариант, когда пассажир может находиться в такси! Переведем среду в состояние, где пассажира мы уже подобрали и посмотрим, как ведет себя одно из умений.

```
s = env.reset()
env.unwrapped.s = 499
env.render()
print("\n" * 2)
r = apply_option(options[0], 0.99, env, debug=True)
```

Оказывается, что умение не обучилось действовать в такой ситуации. Исправим нашу функцию обучения так, чтобы умения работали корректно для всех возможных состояний среды, и сгенерируем их заново.

```
def play_and_train_modified(env, agent, t_max=10 ** 4):
    # Зададим новую функцию play_and_train, которая
    # в качестве начального состояния выбирает любое
    # состояние среды, включая и то, когда пассажир
    # находится в такси

    total_discounted_reward = 0.0
    s = env.reset()

    # Выбираем случайное состояние среды
    # (используем метод env.unwrapped)
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


    for t in range(t_max):
        a = agent.get_action(s)
        next_s, r, done, _ = env.step(a)
        agent.update(s, a, next_s, r)
        s = next_s
        total_discounted_reward += r
        if done:
            break
    return total_discounted_reward


for index in range(4):
    for _ in range(5250):
        wrapped_env = TaxiStepWrapper(env=environment,
                target_id=index, target_reward=50)
        play_and_train_modified(env=wrapped_env,
                agent=agents_for_options[index])
```

Запустим исправленный выше код несколько раз и убедимся, что агент обучился для всех случаев!

```
env = environment
s = env.reset()

env.unwrapped.s = random.randint(0, 499)
apply_option(options[0], 0.99, env, debug=True)
```

§5.3. Объединение умений в иерархию

Теперь необходимо реализовать иерархию, используя элементарные (умения из одного действия) и обученные умения. Элементарные действия: посадка и высадка пассажира.

```
# для действий 4-5 (pickup, dropoff) создаем
# элементарные умения:
class OneActionAgent:
    def __init__(self, action):
        self.action = action

    def get_action(self, state):
        return self.action

    def update(*args, **kwargs):
        pass

options = options[:4]
for action in range(4, 6):
    # элементарное умение начинается в любом состоянии,
    # выполняет любое действие и завершается
    initial = set(range(environment.observation_space.n))
    termination_prob = {_:1.0 \
    for _ in  range(environment.observation_space.n)}
    options.append(Option(policy=OneActionAgent(action),
    termination_prob=termination_prob, initial=initial))

env = environment
s = env.reset()

env.unwrapped.s = random.randint(0, 499)
apply_option(options[0], 0.99, env, debug=True)
apply_option(options[4], 0.99, env, debug=True)
```

Реализуем обертку для окружения, которая вместо действий применяет умения (в качестве входных параметров используется список умений).

```
class OptionTaxiStepWrapper(gym.Wrapper):
    def __init__(self, env, options, gamma=0.99):
        self.options = options
        self.gamma = gamma
        super().__init__(env)

    def _step(self, action):
        state, reward, is_done, obs =\
        apply_option(self.options[action],
            self.gamma, self.unwrapped)
        return state, reward, is_done, obs

option_agent = QLearningAgent(alpha=0.1, epsilon=0.1, gamma=0.99,
                          get_legal_actions=lambda s: range(len(options)))

import matplotlib.pyplot as plt
```

```
%matplotlib inline
from IPython.display import clear_output

# создаем окружение, использующее опции
env = OptionTaxiStepWrapper(gym.make('Taxi-v2'),
                            options=options)
rewards = []
for episode in range(500):
    rewards.append(
        play_and_train(env=env, agent=option_agent))

    if episode % 100 == 0:
        clear_output(True)
        option_agent.epsilon *= 0.99
        plt.plot(rewards)
        plt.show()
```

6. Алгоритмы градиента стратегии

В некоторых задачах для нахождения удовлетворяющей стратегии необязательно изучать структуру всей среды. Например, в задаче поднятия кубика робототехнической рукой вместо точной аппроксимации $Q(s, a)$ достаточно знать, что выгоднее двигаться вправо, если кубик справа, и влево в ином случае. Алгоритм Reinforce (Monte Carlo policy gradient) – это алгоритм поиска стратегий, в котором параметры, задающие стохастическую стратегию, изменяются в соответствии с градиентом математического ожидания награды:

$$J(\theta) = E_{\tau \sim p_\theta(\tau)} \sum_t \gamma^t r(s_t, a_t)),$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta).$$

Эксперименты с реализацией данного алгоритма будем проводить на среде с перевернутым маятником CartPole.

```
import numpy as np
import tensorflow as tf

import gym

# Будем обучать агента,
# используя сначала простое окружение CartPole-v0,
# а затем усложним задачу, взяв Acrobot-v1

# Создаем окружение
env = gym.make("CartPole-v0")

observation_shape = env.observation_space.shape
n_actions = env.action_space.n
gamma = 0.95

print("Observation Space", env.observation_space)
print("Action Space", env.action_space)
```

§6.1. Создание аппроксиматора стратегии

Стратегию будем задавать весами нейронной сети. Она является стохастической, т. е. в состоянии s она представляет собой некоторое распределение $\pi_\theta(s)$, поэтому на вход сети будет подаваться состояние s, а на выходе будут вероятности действий. В начале определим вход сети.

```
# Задаем переменные, которые будут
# подаваться на вход нейронной сети

# Состояния
# observations =
#~~~~~~~~ Ваш код здесь ~~~~~~~~~~
raise NotImplementedError
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Действия
# actions =
#~~~~~~~~ Ваш код здесь ~~~~~~~~~~
raise NotImplementedError
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Вознаграждение
# discounted_episode_rewards =
#~~~~~~~~ Ваш код здесь ~~~~~~~~~~
raise NotImplementedError
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

all_inputs = [observations,
        actions, discounted_episode_rewards]
```

Затем перейдем к определению графа вычислений.

```
sess = tf.InteractiveSession()

# Задаем внутренние и выходной слои нейронной сети
# nn1 = ..., nn2 = ..., nn3 = ...
#~~~~~~~~ Ваш код здесь ~~~~~~~~~~
raise NotImplementedError
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


probs_out = tf.nn.softmax(nn3)
# Выход последнего слоя преобразуется
# в стохастическую стратегию, поэтому
# количество нейронов должно быть равно n_actions

def discount_and_normalize_rewards(episode_rewards):
    discounted_episode_rewards = np.zeros_like(
        episode_rewards)
    cumulative = 0.0

    # Считаем дисконтированное вознаграждение
    # "G = r + gamma*r' + gamma^2*r'' + ..."
    for i in reversed(range(len(episode_rewards))):
        cumulative = cumulative * gamma\
```

```
        + episode_rewards[i]
        discounted_episode_rewards[i] = cumulative

    # Нормализуем данные
    mean = np.mean(discounted_episode_rewards)
    std = np.std(discounted_episode_rewards)
    discounted_episode_rewards =\
    (discounted_episode_rewards - mean) / (std)

    return discounted_episode_rewards
```

§6.2. Функция потерь для градиента стратегии

Теперь определим *функцию потерь* (Crossentropy loss). Градиент стратегии выглядит следующим образом:

$$\nabla_\theta J_\theta \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) R.$$

Чтобы автоматически вычислить градиент, необходимо задать граф, который имеет градиент такого же вида. Для этого используется *псевдофункция потерь*:

$$\tilde{J}_\theta = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log \pi_\theta(a_{i,t}|s_{i,t}) R.$$

```
# Применяем кроссэнтропию к ПОСЛЕДНЕМУ слою сети
# tf.nn.softmax_cross_entropy_with_logits_v2
# neg_log_prob =
#^^^^^^^^ Ваш код здесь ^^^^^^^^^^^^
raise NotImplementedError
#^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

# Умножаем на дисконтированное вознаграждение
# и берем среднее, чтобы получить искомую функцию потерь
# loss =
#^^^^^^^^ Ваш код здесь ^^^^^^^^^^^^
raise NotImplementedError
#^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

# Задаем метод оптимизации, который будем использовать
```

```
# (например adam с lr 0.01)
# optimizer =
#~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
raise NotImplementedError
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

train_op = optimizer.minimize(loss,
    global_step=tf.contrib.framework.get_global_step())
sess.run(tf.global_variables_initializer())
```

Вычисляемый таким образом градиент имеет большую дисперсию. Для ее уменьшения можно воспользоваться техникой смещения:

$$\tilde{J}_\theta = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\log\pi_\theta(a_{i,t}|s_{i,t})(R-b),$$

где b может быть константой (это несильно улучшит работу алгоритма), а может быть функцией от s, например $V(s)$. $V(s)$ может быть аппроксимирована другой нейронной сетью. Настройка может проходить по методу наименьших квадратов: необходимо задать функцию ошибок, и можно добавить ее к функции потерь с некоторым коэффициентом или минимизировать отдельно[5].

```
allRewards = []
total_rewards = 0
maximumRewardRecorded = 0
episode = 0
episode_states, episode_actions, = [], []
episode_rewards = []
max_episodes = 10000

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for episode in range(max_episodes):

        episode_rewards_sum = 0
        state = env.reset()
        env.render()

        while True:

            # Получаем распределение вероятностей
            # действий, согласно стохастической
            # стратегии агента
```

[5]Например, см. https://github.com/yrlu/reinforcement_learning/blob/master/policy_gradient/reinforce_w_baseline.py

```
            # probs =
            #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
            raise NotImplementedError
            #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            # Выбираем действие (согласно распределению)
            # action =
            #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
            raise NotImplementedError
            #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


            # Применяем действие в среде
            #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
            raise NotImplementedError
            #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            episode_states.append(state)
            action_ = np.zeros(n_actions)
            action_[action] = 1

            episode_actions.append(action_)

            episode_rewards.append(reward)
            if done:
                episode_rewards_sum = np.sum(
                    episode_rewards)

                allRewards.append(episode_rewards_sum)

                total_rewards = np.sum(allRewards)

                mean_reward = np.divide(total_rewards,
                                        episode + 1)

                maximumRewardRecorded = np.amax(
                    allRewards)
                if episode % 50 == 0:
                    print("==============" * 5)
                    print("Episode: ", episode)
                    print("Reward: ", episode_rewards_sum)
                    print("Mean Reward", mean_reward)
                    print("Max reward so far: ",
                          maximumRewardRecorded)

                # Считаем дисконтированное вознаграждение
                # discounted_rewards =
                #~~~~~~~~ Ваш код здесь ~~~~~~~~~~~
                raise NotImplementedError
                #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
            # Считаем значение функции потерь
            # и настраиваем веса сети
            # при помощи оптимизатора
            # loss_, _ =
            #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
            raise NotImplementedError
            #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

            episode_states = []
            episode_actions = []
            episode_rewards = []

            break

        state = new_state
```

С использованием реализованного алгоритма необходимо построить график получаемого суммарного вознаграждения от номера эпизода. Затем можно усложнить архитектуру сети и проверить улучшает ли это производительность.

7. Планирование и обучение с подкреплением

В данном разделе мы реализуем класс DynaQAgent, который будет представлять собой табличный Dyna-Q агент. Сравним его с Q-learning агентом. Все эксперименты проведем с задачей Taxi-v2.

```
from collections import defaultdict
import random
import math
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import gym
```

§7.1. Алгоритм Dyna-Q

Создадим класс DynaQAgent, который будет решать задачу обучения методом Dyna-Q, согласно приведенному на рис. 7.1 алгоритму.

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \epsilon$-greedy(S,Q)
- (c) Execute action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Рис. 7.1. Алгоритм Dyna-Q

```
from collections import defaultdict
import random
import math
import numpy as np

class DynaQAgent:
```

```
    def __init__(self, alpha, epsilon, discount,
                 n_steps, get_legal_actions):
        self.get_legal_actions = get_legal_actions
        self._qvalues =\
        defaultdict(lambda: defaultdict(lambda: 0))
        self._memoryModel = []
        self.alpha = alpha
        self.epsilon = epsilon
        self.discount = discount
        self.n_steps = n_steps

    def get_qvalue(self, state, action):
        return self._qvalues[state][action]

    def set_qvalue(self, state, action, value):
        self._qvalues[state][action] = value

    def get_model(self, state, action):
        return self._memoryModel[state][action]

    def set_model(self, state, action, value):
        self._memoryModel[state][action] = value

    def get_value(self, state):

        possible_actions = self.get_legal_actions(state)
        value = max(self.get_qvalue(state, action)\
                    for action in possible_actions)

        return value
```

В соответствии с алгоритмом представленном выше будем добавлять методы к нашему классу. Начнем с обновления Q-значений.

```
def update(self, state, action, reward, next_state):
    alpha = self.alpha
    gamma = self.discount
    n_steps = self.n_steps
    # Q_update =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    self.set_qvalue(state, action, Q_update)
    self._memoryModel.append((state,
            action, reward, next_state))
    self.search()

DynaQAgent.update = update
```

Следующий шаг – этап планирования (поиска).

```
def search(self):
    n_steps = self.n_steps
    alpha = self.alpha
    gamma = self.discount
    for _ in range(n_steps):
        # выбираем случайно из памяти
        # state, action, reward, next_state =
        #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        # Q_update =
        #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
        raise NotImplementedError
        #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        self.set_qvalue(state, action, Q_update)

DynaQAgent.search = search
```

В соответствии со сформированной таблицей значений выбираем действия и используем ϵ-жадную стратегию.

```
def get_best_action(self, state):
    possible_actions = self.get_legal_actions(state)

    # выбираем лучшее действие
    # action =
    #~~~~~~~~ Ваш код здесь ~~~~~~~~~~
    raise NotImplementedError
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    return action

DynaQAgent.get_best_action = get_best_action

def get_action(self, state):
    possible_actions = self.get_legal_actions(state)
    action = None
    epsilon = self.epsilon

    if random.random() > epsilon:
        chosen_action = self.get_best_action(state)
    else:
        chosen_action = random.choice(possible_actions)

    return chosen_action

DynaQAgent.get_action = get_action
```

§7.2. Сравнение работы Dyna-Q с Q-обучением

Сравним получившийся алгоритм с классическим Q-обучением на примере окружения Taxi-v2. Вначале определим среду и агентов.

```
import gym
env = gym.make("Taxi-v2")
n_actions = env.action_space.n
env.render()

from qlearning import QLearningAgent

a = defaultdict(lambda:0)
agent_ql = QLearningAgent(alpha=0.25, epsilon=0.05,
    discount=0.9,
    get_legal_actions=lambda s: range(n_actions))

agent_dynaq = DynaQAgent(alpha=0.25, epsilon=0.05,
    discount=0.9, n_steps=50,
    get_legal_actions=lambda s: range(n_actions))
```

Реализуем функцию *play_and_train*, которая будет обеспечивать взимодействие агента со средой в течение всего эпизода, проводить обучение агента на каждой паре состояние–действие и возвращать полученную награду.

```
def play_and_train(env, agent, t_max=10**4):
    """Функция запускает целый эпизод:
    - действия выбираются с помощью agent.get_action(...);
    - обучение агента - agent.update(...),
    и возвращает общее вознаграждение"""
    total_reward = 0.0
    s = env.reset()

    for t in range(t_max):
        a = agent.get_action(s)

        next_s, r, done, _ = env.step(a)
        agent.update(s, a, r, next_s)

        s = next_s
        total_reward += r
        if done:
            break

    return total_reward
```

Результаты полученных экспериментов отобразим на графике.

```
from IPython.display import clear_output
from pandas import DataFrame


def moving_average(x, span=100): return DataFrame(
    {'x': np.asarray(x)}).x.ewm(span=span).mean().values


rewards_dynaq, rewards_ql = [], []

for i in range(1000):
    rewards_dynaq.append(play_and_train(env, agent_dynaq))
    rewards_ql.append(play_and_train(env, agent_ql))

    if i % 10 == 0:
        clear_output(True)
        print('DYNA-Q mean reward =',
              np.mean(rewards_dynaq[-100:]))
        print('QLEARNING mean reward =',
              np.mean(rewards_ql[-100:]))
        plt.plot(moving_average(rewards_dynaq),
                 label='dyna-q')
        plt.plot(moving_average(rewards_ql),
                 label='q-learning')
        plt.grid()
        plt.legend()
        plt.show()
```

Заключение

Учебно-методическое пособие является сборником практических задач для курса машинного обучения с подкреплением [4]. Схемы программного кода на языке программирования Python 3 сопровождаются кратким вводным теоретическим материалом и комментариями по отдельным блокам кода.

В качестве разделов в пособие включены методы динамического программирования, методы, основанные на полезности, методы, основанные на стратегии, градиент стратегии и введение в иерархический подход. Наряду с разбором методов аппроксимации, пособие охватывает основные разделы обучения с подкреплением.

Данное пособие будет полезно студентам и аспирантам, специализирующимся по направлению искусственного интеллекта и, в частности, может выступать в качестве дополнительных глав для курсов машинного обучения и интеллектуальных систем в робототехнике.

Литература

1. *Dietterich T.G.* Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition // Journal of Artificial Intelligence Research. 2000. V. 13. P. 227–303. URL: `https://arxiv.org/abs/9905014`
2. *Brockman G., et al.* OpenAI Gym // arXiv.org. 2016. URL: `https://arxiv.org/abs/1606.01540`
3. *Abadi M., et al.* TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. (Software available on URL: `https://www.tensorflow.org/`)
4. *Панов А.И.* Введение в методы машинного обучения с подкреплением : учебное пособие. Москва : МФТИ, 2019. 56 с.
5. *Саттон Р.С., Барто Э.Г.* Обучение с подкреплением. 2-е изд. Москва : БИНОМ. Лаборатория знаний, 2011. 399 с.