

Как я понял, выполнение первого задания довольно прямолинейно: без изменения самого алгоритма, нужно нагенерировать политик для разных значений γ , и после сравнить их.

```
epochs = 20
evaluation_step_n = 60
gammas = np.linspace(0, 1, 20)
policies = []

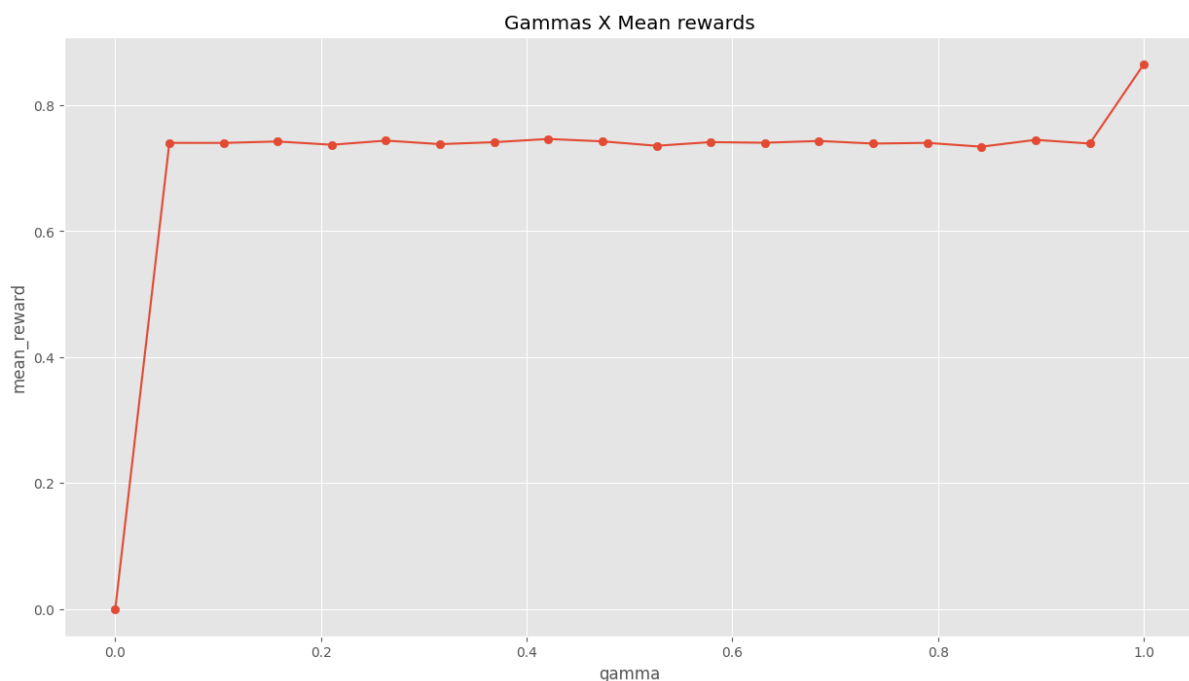
policy = init_policy(env)
for gamma in gammas:
    for epoch in range(epochs):
        q_values = policy_evaluation(policy, gamma, evaluation_step_n)
        policy = policy_improvement(q_values)
    policies += [policy]
```

Тут я просто добавил лист для значений γ состоящий из 20 элементов, и лист для политик. После этого нужно просто было их сравнить, запуская среду несколько раз и беря среднюю награду. Я решил запускать 10_000 раз, так как считается оно быстро. Результат виден ниже.

```
mean_rewards = []
for policy in policies:
    mean_reward = 0
    for i in tqdm(range(10000), colour='blue', leave=True):
        total_reward = 0
        state = env.reset()
        for _ in range(100):
            action = np.random.choice(env.get_possible_actions(state), p=list(policy[state].values()))
            state, reward, done, _ = env.step(action)
            #env.render()
            #time.sleep(0.5)
            total_reward += reward

        if done:
            break

    mean_reward += total_reward
    mean_reward /= 10000
    mean_rewards += [mean_reward]
    print(mean_reward)
```



2. Вместо того, чтобы написать и попробовать я постарался сначала подумать. Мне начало казаться, что уравнение Беллмана подразумевает, что мы будем начинать не с нулевых values, а с values, полученных на предыдущем шаге. Это не совсем так, однако, как я понял алгоритм будет работать в силу следующих причин:

- 1) в функции `policy_evaluation()`, как она была прописана в рамках практического занятия есть гипер(?)параметр `evaluation_step_n`, который решает, сколько итераций values должно пройти, перед тем как эпоха закончится. Во время практического занятия значение было поставлено на 60, однако и после 60 шагов values инициализировались заново.
- 2) у параметра `evaluation_step_n` нет какого-то строгого ограничения сверху - единственное, что стоит учитывать - в какой-то момент values сойдутся и перестанут обновляться, однако алгоритм не сломается

```
J: epochs = 20
evaluation_step_n = 100
gamma = 0.9

policy_with_resetting = train(epochs, evaluation_step_n, gamma, env, reset_values=False)
policy_without_resetting = train(epochs, evaluation_step_n, gamma, env, reset_values=False)

reward_with_resetting = test(policy_with_resetting)
reward_without_resetting = test(policy_without_resetting)

J: print(reward_with_resetting)
print(reward_without_resetting)

1.0
1.0
```

Как видно, оба алгоритма достигают цели.

По поводу того, будет ли он работать лучше: я вижу два способа оценить алгоритм:

- 1) достижение оптимальной политики за меньшее `evaluation_step_n`
- 2) так же как и в предыдущем пункте обращаться к среде `n` раз и брать среднюю награду

Второй проверить проще

```

epochs = 20
evaluation_step_n = 100
gamma = 0.9

policy_with_resetting = train(epochs, evaluation_step_n, gamma, env, reset_values=False)
policy_without_resetting = train(epochs, evaluation_step_n, gamma, env, reset_values=False)

rewards_with_resetting = [test(policy_with_resetting) for _ in tqdm(range(10000))]
rewards_without_resetting = [test(policy_without_resetting) for _ in tqdm(range(10000))]

100% ██████████ 10000/10000 [00:12<00:00, 942.29it/s]
100% ██████████ 10000/10000 [00:12<00:00, 805.54it/s]

print(f'Mean reward for 10_000 tries with resetting [{np.mean(rewards_with_resetting)}]')
print(f'Mean reward for 10_000 tries with resetting [{np.mean(rewards_without_resetting)}]')

Mean reward for 10_000 tries with resetting [0.7373]
Mean reward for 10_000 tries with resetting [0.7452]

```

3. Value iteration

Для того, чтобы выполнить данную задачу, я в первую очередь руководствовался слайдом (что логично).

Value Iteration

Let $v^0(s)$, $s \in \mathcal{S}$ and $K \in \mathbb{N}$.

For each $k \in \overline{0, K}$, do

$$v^{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}(s, a) + \gamma \sum_{s'} \mathcal{P}(s'|s, a) v^k(s') \right), \quad s \in \mathcal{S}$$

Theorem

$v^k \rightarrow v_*$, $k \rightarrow \infty$. Convergence rate $O(mn^2)$

Я интерпретировал данное уравнение следующим образом:

```

iterate through actions per state:
  find
  reward + gamma * sum(transition_prob * values_next_state)
  choose max|

```

Одной из сложностей является то, что функция награды в среде frozen lake зависит также от следующего состояния. И по идее, она будет учитываться по несколько раз для каждого действия. Однако, это не противоречит самой логике уравнения Беллмана, ибо цель - найти оптимальное действие. Однако, это все более менее теряет смысл, ибо награда во всех состояниях, кроме финального = 0.

Тем не менее, насколько я понял, в value iteration нужно:

- 1) инициализировать values, которые будут зависеть от текущего состояния

```
: def init_values(env):  
    return {state: 0 for state in env.get_all_states()}
```

- 2) для каждого состояния

- a) для каждого действия

- i) для каждого из следующих состояний:

- (1) найти награду, вероятность перехода, значение value
- (2) просуммировать и умножить на gamma

- b) выбрать действие с максимальным значением, и записать его в словарь values

Также, очевидным является, что для последующего составления политики нам нужно запоминать не только максимальное значение для действия, но и само действие с максимальным значением, что можно записывать в отдельный словарь, который я назвал action_value_dict, вероятнее всего в лекции это называлось action_value_function (q_values). Мне важно понять, прав ли я в этом моменте.

Все действия, описанные выше привели к написанию следующей функции:

```
def value_function(env, values, gamma):  
    state_action_dict = {}  
    for state in env.get_all_states():  
        state_action_dict[state] = {}  
        for action in env.get_possible_actions(state):  
            state_action_dict[state][action] = 0  
            for next_state in env.get_next_states(state, action):  
                reward = env.get_reward(state, action, next_state)  
                transition_prob = env.get_transition_prob(state, action, next_state)  
                next_value = values[next_state]  
                state_action_dict[state][action] += transition_prob * next_value  
                state_action_dict[state][action] += reward  
            state_action_dict[state][action] *= gamma  
  
    if state_action_dict[state]:  
        max_action_value = max(state_action_dict[state], key=state_action_dict[state].get)  
        values[state] = state_action_dict[state][max_action_value]  
  
    return state_action_dict
```

Далее нужно просто написать функцию, которая бы выдавала политику на основании вышеописанных действий. Тут все достаточно прямолинейно: согласно теореме, оптимальные values будут найдены после кол-во действий * кол-во состояний**2 итераций, и после этого нужно будет просто из state_action_dict извлечь действие с

максимальным значением для каждого состояния

```
def value_iteration(n_iterations, env, gamma):
    values = init_values(env)


    for _ in tqdm(range(n_iterations)):
        state_action_dict = value_function(env, values, gamma)
        policy = {}
        for state in env.get_all_states():
            if state_action_dict[state]:
                max_action_value = max(state_action_dict[state], key=state_action_dict[state].get)
                policy[state] = max_action_value
    return policy
```

После этого остается написать функцию, которая бы оценивала политику.

```
def evaluate(env, policy, vizualize=False):
    state = env.reset()
    for _ in range(100):
        action = policy[state]
        state, reward, done, _ = env.step(action)
        if vizualize:
            env.render()
        if done:
            break
    return reward

n_iterations = states_n ** 2 * total_actions_n
gamma = 0.9

policy = value_iteration(n_iterations, env, gamma)

100%  1024/1024 [00:00<00:00, 2277.91it/s]

evaluate(env, policy)

1.0

rewards = [evaluate(env, policy) for _ in range(1000)]
mean_reward = sum(rewards) / len(rewards)
print(mean_reward)

0.72
```

Для того, чтобы сравнить количество обращений к среде, я решил просто посчитать количество вызовов методов класса MDP из Frozen_Lake.py (за исключением методов reset и step).

Для этого я просто добавил параметр (возможно слово параметр это неверный термин) к классу MDP, который увеличивается на 1 каждый раз, когда происходит вызов метода за все обучение в целом. Не факт, что это лучший способ оценить

сложность алгоритма, в силу того, что в policy iteration, опять же, есть внутренний цикл. Тем не менее, количество обращений к среде на протяжении обучения для policy iteration выглядит следующим образом:

```
epoch = 0      env.counter = 16081
epoch = 1      env.counter = 32100
epoch = 2      env.counter = 48119
epoch = 3      env.counter = 64138
epoch = 4      env.counter = 80157
epoch = 5      env.counter = 96176
epoch = 6      env.counter = 112195
epoch = 7      env.counter = 128214
epoch = 8      env.counter = 144233
epoch = 9      env.counter = 160252
epoch = 10     env.counter = 176271
epoch = 11     env.counter = 192290
epoch = 12     env.counter = 208309
epoch = 13     env.counter = 224328
epoch = 14     env.counter = 240347
epoch = 15     env.counter = 256366
epoch = 16     env.counter = 272385
epoch = 17     env.counter = 288404
epoch = 18     env.counter = 304423
epoch = 19     env.counter = 320442
```

и для value iteration я решил выводить раз в 100 эпох.

```
100% ██████████ 1024/1024 [00:00<00:00, 1705.52it/s]
iteration = 0      env.counter = 748
iteration = 100    env.counter = 75248
iteration = 200    env.counter = 149748
iteration = 300    env.counter = 224248
iteration = 400    env.counter = 298748
iteration = 500    env.counter = 373248
iteration = 600    env.counter = 447748
iteration = 700    env.counter = 522248
iteration = 800    env.counter = 596748
iteration = 900    env.counter = 671248
iteration = 1000   env.counter = 745748
```


Как видно, для кол-во действий * кол-во состояний**2 итераций потребовалось 745 тысяч обращений к среде, что больше, чем для обучения с policy iteration. После этого я задумался, что 1024 итерации может быть излишне много. Постепенно уменьшая количество итераций и запуская в конце обучения среду 1000 раз для оценки алгоритма я дошел до того, что минимальное количество итераций, необходимое для обучения равно 4 !!! (я крайне удивился). Помимо всего прочего, при увеличении количества итераций средняя награда за 1000 эпох падает. Таким образом, при 4 итерациях:

```
: rewards = [evaluate(env, policy) for _ in range(1000)]
mean_reward = sum(rewards) / len(rewards)
print(mean_reward)

0.833
```

```
#n_iterations = states_n ** 2 * total_actions_n
n_iterations = 4
gamma = 0.9

policy = value_iteration(n_iterations, env, gamma)
```


100%  4/4 [00:00<00:00, 148.91it/s]

iteration = 0	env.counter = 748
iteration = 1	env.counter = 1493
iteration = 2	env.counter = 2238
iteration = 3	env.counter = 2983

Меньше 3000 обращений к среде за все обучение довольно впечатлило.

На всякий случай, я решил перепроверить и через .py скрипт, и результат изменился не сильно

```
iteration = 0    env.counter = 748
iteration = 1    env.counter = 1493
iteration = 2    env.counter = 2238
iteration = 3    env.counter = 2983
```

100% 

mean_reward = 0.809 for 1000 iterations