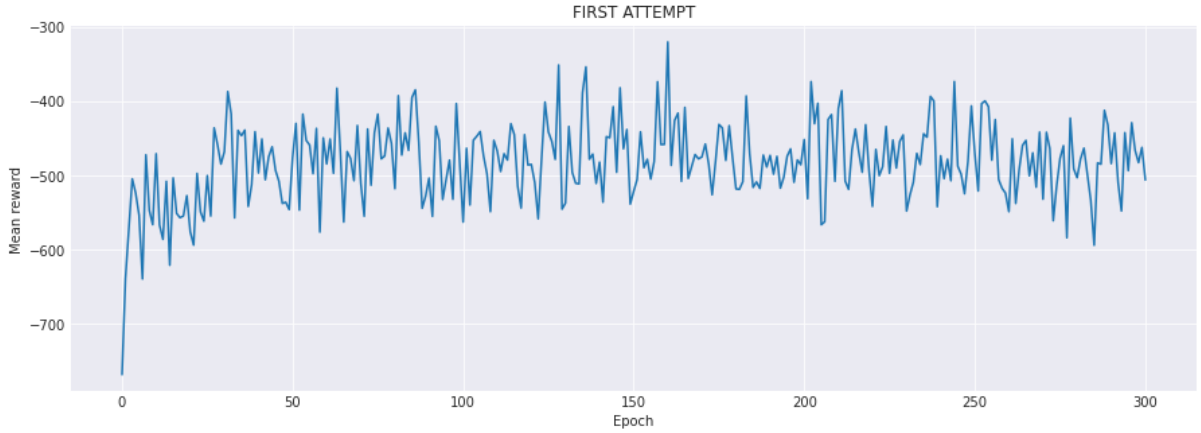
1.1 Обучение агента

Я начал эксперименты с копирования кода, представленного на практическом занятии 1. 50 траекторий за эпоху, 300 эпох, q_value=0.9. Результат был далек от удовлетворительного: средняя награда не поднималась выше -300 за эпоху.



Логично, что это связано с тем, что доступных состояний, как и действий существенно больше, чем в задании, рассмотренном на практическом занятии; агент зачастую совершал невозможные действия (попытка въехать в стену, попытка забрать/высадить пассажира в состоянии, в котором это невозможно).

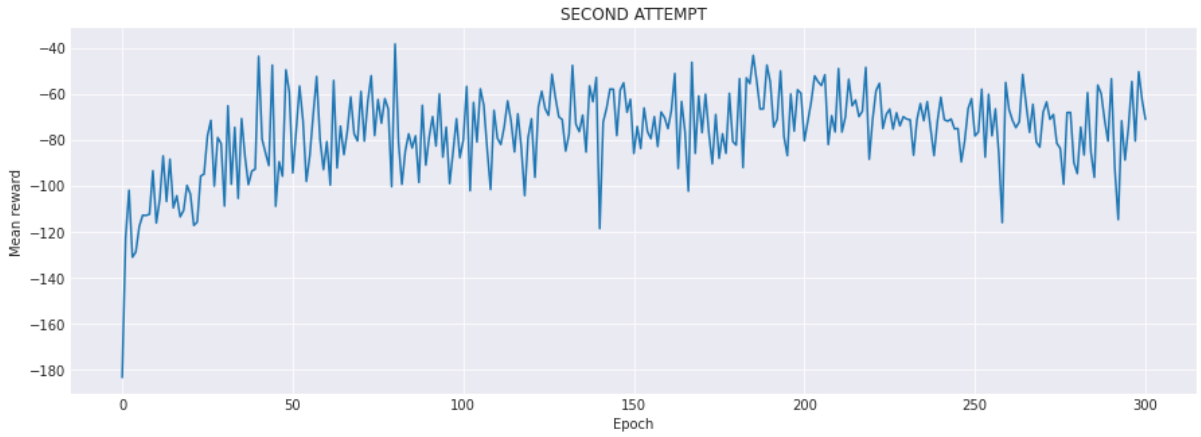
Следовательно, нужно исключить подобное поведение. Сделать это можно, используя саму среду и метод `action_mask`. Однако, насколько я понимаю, даже если бы данная опция была недоступна, получить матрицу доступных действий можно было бы, используя BFS.

```
def get_possible_actions(env, state_n, action_n):
 possible_actions = np.zeros((state_n, action_n))
 for state in range(state_n):
   possible_actions[state] = env.action_mask(state) / sum(env.action_mask(state))
 return possible_actions
```

Получив матрицу возможных действий, мы можем установить ее в качестве начальной политики:

```
class CEM():
 def __init__(self, state_n: int, action_n: int, possible_actions=None):
   self.state_n = state_n          # number of possible states
   self.action_n = action_n        # number of possible actions
   if possible_actions is not None:
     self.policy = possible_actions      # matrix that presents probs of choosing an action
for particular state
   else:
     self.policy = np.ones((self.state_n, self.action_n)) / self.action_n
```

Данные действия привели к существенному улучшению средних наград:



На данном этапе, я также решил добавить функцию `test` и метод `get_actions_for_test`, который использует `numpy.argmax()` вместо `numpy.choice()`.

```
def test(env, agent, visualize=False):
 state = env.reset()
 if visualize:
   env.render()
   time.sleep(1)

 done = False
 last_trajectory = []
 last_reward = 0
 while not done:
   action = agent.get_action_for_test(state)
   last_trajectory += [action]
   state, reward, done, _ = env.step(action)
   last_reward += reward
   if visualize:
     env.render()
     time.sleep(0.2)
 return last_trajectory, last_reward
```

```
  def get_action_for_test(self, state):
   '''Choosing an action with the highest probability'''
   return np.argmax(self.policy[state])
```

Возможно, лучшим названием для функции было бы `validate`, так как в основном я использовал ее для вывода информации в процессе обучения. Для меня большим сюрпризом, что награда и длина успешной траектории на самом деле может уменьшаться в процессе обучения, то есть агент “деградирует”:

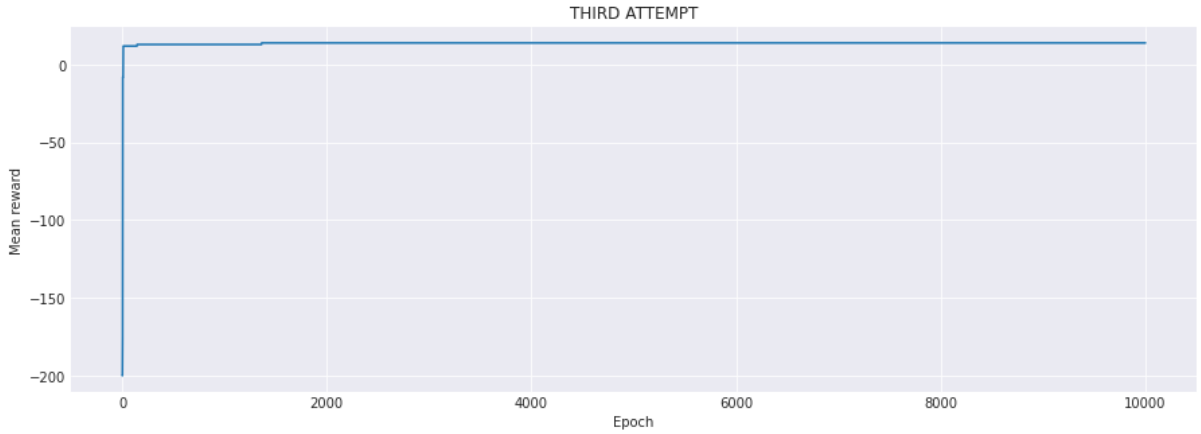
```
Epoch [0]                  Train mean reward [-183.44]
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
200        ← test trajectory length      ^ test trajectory
-200       ← test reward
Epoch [50]                 Train mean reward [-94.48]
Epoch [100]                Train mean reward [-80.12]
[0, 0, 0, 4, 1, 1, 2, 2, 1, 1, 2, 2, 5]
13         ← test trajectory length      ^ test trajectory (improvement !!)
8          ← test reward
Epoch [150]                Train mean reward [-86.0]
Epoch [200]                Train mean reward [-80.46]
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
200        ← test trajectory length      ^ test trajectory (degradation !!)
-200       ← test reward
Epoch [250]                Train mean reward [-78.36]
Epoch [300]                Train mean reward [-71.2]
[0, 3, 0, 0, 0, 4, 1, 1, 2, 2, 1, 1, 2, 2, 5]
15         ← test trajectory length      ^ test trajectory (improvement, but it was better)
6          ← test reward
Training took 85.95713924700067 secs
[1, 3, 0, 0, 3, 1, 3, 0, 0, 4, 1, 1, 2, 1, 1, 3, 5]
4          ← test reward                 ^ test trajectory (technically, degradation)
```

Как я полагал, это связано с тем, что политика агента обновляется на элитных траекториях, а в элитные траектории может попасть траектория, хуже, чем в предыдущих эпохах, в случае, если награда за эту траекторию была выше, чем q_value (значение квантиля 0.9). Соответственно, следующий шаг – обновление функции `get_elite_trajectories()`:

```
def get_elite_trajectories(trajectories, q, curr_max_reward):

 rewards = [trajectory['max_reward'] for trajectory in trajectories]
 q_value = np.quantile(a=rewards, q=q)
  if q_value > curr_max_reward:
   return np.mean(rewards), [trajectory for trajectory in trajectories
if trajectory['max_reward'] > q_value]
 else:
   return np.mean(rewards), []
```

Обновление заключается в том, что теперь один из аргументов – текущая максимальная награда, и политика не будет обновляться, если не достигнут результат лучше, чем на прошлых итерациях.
Данная идея немного улучшила результат.

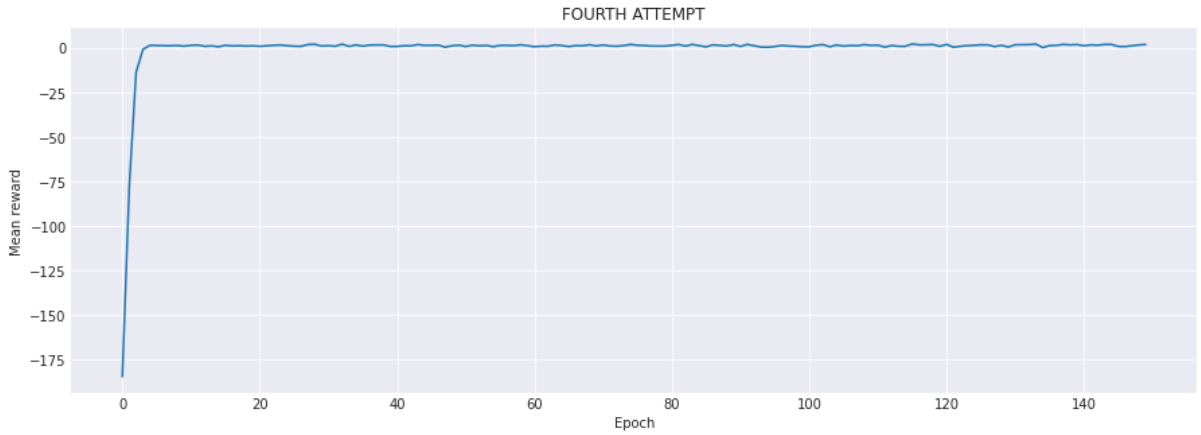
Однако, после этого, для того, чтобы достигнуть максимально быстрого улучшения награды, я решил сократить количество траекторий за эпоху до одной, и увеличить количество эпох соответственно. Это породило довольно внушающий результат:



Положительная награда была достигнута за 10 эпох = 10 траекторий:

```
history[:15]
[-200, -200, -200, -175, -175, -8, -8, -8, -8, -8, 12, 12, 12, 12, 12]
```

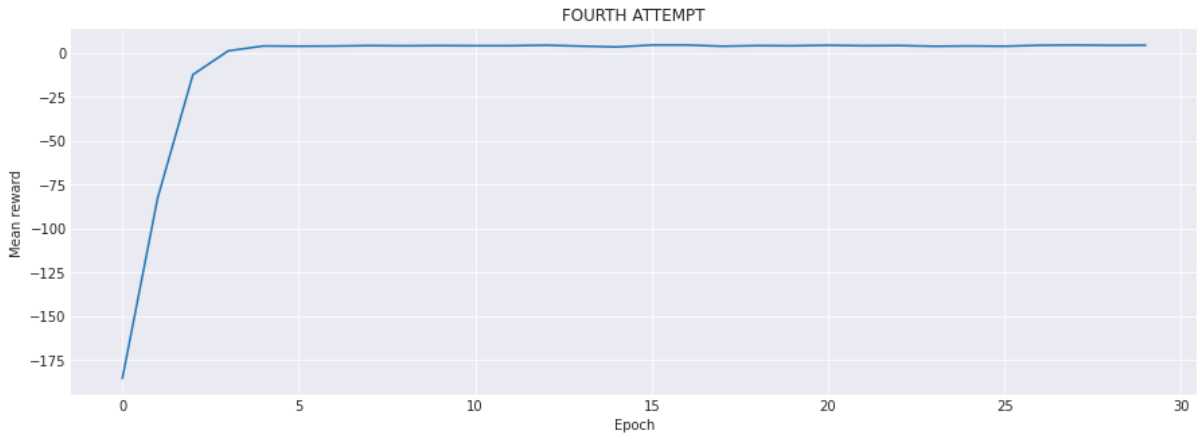
После этого награда выросла до 13 на 145 шаге и до 14 на 1362. После этого награда не менялась.
Но это не означает, что на каждой из сгенерированных траекторий награда была таковой.
Как я понял, в библиотеке существует баг, при котором начисляется награда за доставку пассажира даже если пассажир по факту не был подобран (судя по env.render()).
После этого я впал во фрустрацию и решил написать код, чтобы он хоть как-то работал. Параметры: q_value = 0.9, количество траекторий за эпоху = 1000, количество эпох = 150



На этот раз, результат реален.
Как видно, количество эпох вполне может быть сокращено до 30, либо можно прописать early_stopping, но времени до дедлайна осталось мало.
На моей машине: *Training took 217.6291199940024 secs*
В колабе: *Training took 210.9460195699976 secs*

При сокращении количества эпох до 30, в колабе:

Epoch [0] train mean reward [-185.531]
Epoch [5] train mean reward [3.699]
Epoch [10] train mean reward [3.955]
Epoch [15] train mean reward [4.34]
Epoch [20] train mean reward [4.222]
Epoch [25] train mean reward [3.686]
Training took 48.549985871999525 secs



1.2.1 Laplace smoothing

Как я понимаю, в данном случае применять сглаживание Лапласа напрямую не имеет особого смысла, так как заранее известно, что некоторые действия невозможны, т.е. вероятность некоторых действий должна быть равна нулю.

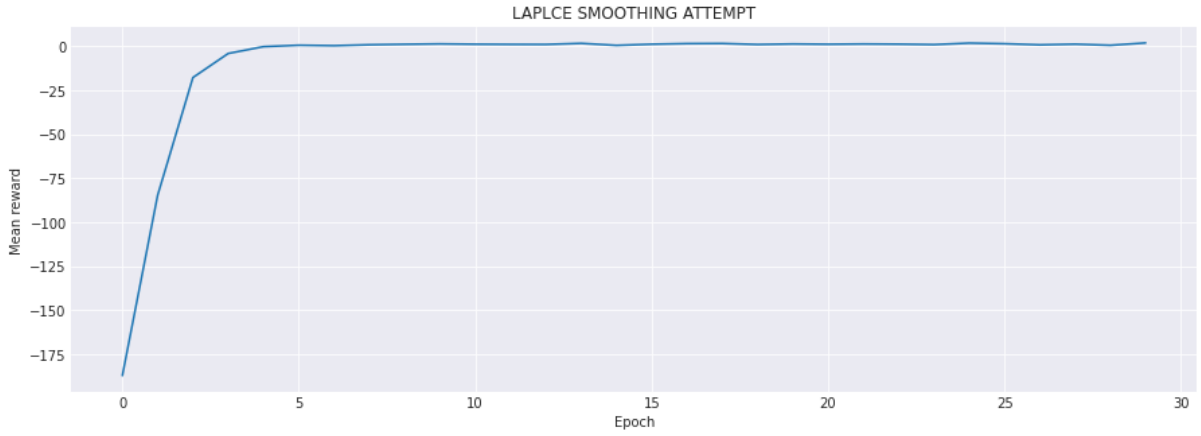
Соответственно, нам нужно прибавлять какое-то число к ненулевым по условию вероятностям. Для этого нужно сделать небольшое изменение в методе update_policy() класса CEM
Более конкретно, нужно заменить

```
# update policy
   for state in range(self.state_n):

     # change probs if action in elite trajectories
     if sum(pre_policy[state]) != 0:
       self.policy[state] = pre_policy[state] / sum(pre_policy[state])
```

на

```
   # update policy
   for state in range(self.state_n):

     # change probs if action in elite trajectories
     if sum(pre_policy[state]) != 0:
       self.policy[state] = (pre_policy[state] + self.policy[state]) / (sum(pre_policy[state]) + 1)
```

Однако, это не произвело качественного воздействия на обучение с последними обозначенными параметрами:

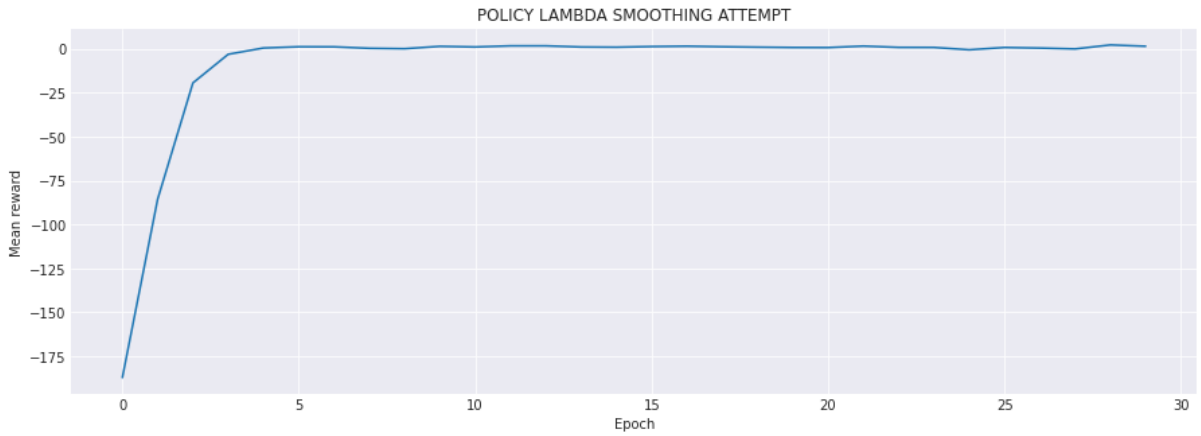


1.2.2 Policy smoothing

Для того, чтобы применить Policy Smoothing, нужно, опять же, немного поменять метод обновления политики.

```
# change probs if action in elite trajectories
     if sum(pre_policy[state]) != 0:
       if policy_lambda:
         if policy_lambda > 1 or policy_lambda <= 0:
           raise ValueError("Policy lambda must be in range (0, 1]")
         pre_policy[state] /= sum(pre_policy[state])
         self.policy[state] = (pre_policy[state] * policy_lambda) +
(self.policy[state] * (1 - policy_lambda))
```

Изначально нужно проверить является ли policy_lamda в промежутку (0, 1] и вызвать ValueError, если нет. после этого нужно сделать так, чтобы сумма pre_policy[state] была равна нулю. после этого просто умножаем согласно формуле. Данная модификация так же не произвела существенного эффекта на обучение агента.



Метод целиком выглядит как представлено ниже и так же представлен во втором .py файле

```
 def update_policy(self, elite_trajectories, laplace_smoothing=False,
policy_lambda=False):
   '''Updates current policy based on elite trajectories by modifying
action probs'''

   pre_policy = np.zeros((self.state_n, self.action_n))

   # extract actions from elite trajectories
   for trajectory in elite_trajectories:
     for state, action in zip(trajectory['states'],
trajectory['actions']):
       pre_policy[state][action] += 1

   # update policy
   for state in range(self.state_n):

     # change probs if action in elite trajectories
     if sum(pre_policy[state]) != 0:
       if laplace_smoothing:
         self.policy[state] = (pre_policy[state] + self.policy[state])
/ (sum(pre_policy[state]) + 1)

       elif policy_lambda:
         if policy_lambda > 1 or policy_lambda <= 0:
```

```
            raise ValueError("Policy lambda must be in range (0, 1]")
          pre_policy[state] /= sum(pre_policy[state])
          self.policy[state] = (pre_policy[state] * policy_lambda) +
(self.policy[state] * (1 - policy_lambda))

        else:
          self.policy[state] = pre_policy[state] /
sum(pre_policy[state])
```

Чем отличается метод кросс энтропии для стохастических сред от обычного понять не удалось
То что я смог понять, это то что нужно сгенерировать детерминированные матрицы
Например, в случае если первые пять состояний представлены вот так:

```
0 [1. 0. 1. 0. 1. 0.]
1 [1. 0. 1. 0. 1. 0.]
2 [1. 0. 1. 0. 1. 0.]
3 [1. 0. 1. 0. 1. 0.]
4 [1. 0. 1. 0. 0. 0.]
```

То детерминированная матрица будет выглядеть следующим образом:

```
 0
 [1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0.]

 1
 [1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0.]

 2
 [1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0.]
```

Однако, чем подобный подход отличается от обычного набора траекторий, остается непонятым.