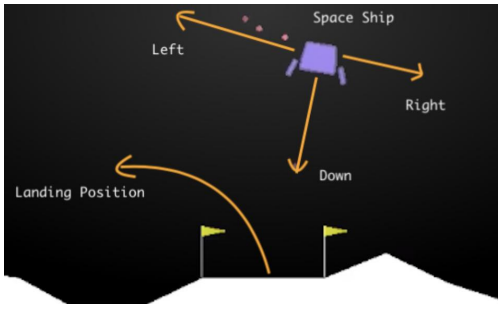
Задание 1.

Я выбрал модель обучение агента для среды “LunarLander-v2”



Входные данные среды:

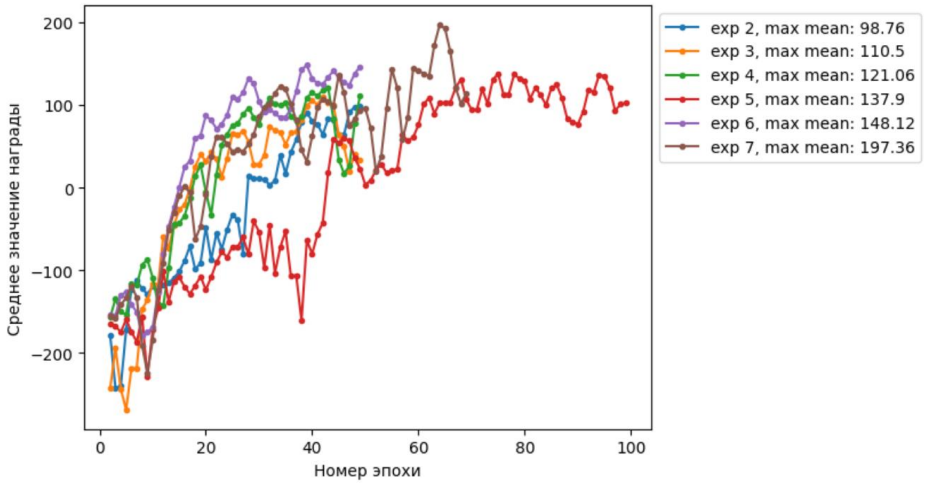
action_n = 4
state_n = 8.

Для возможности фиксирования и чистоты эксперементов был выбран и применен random_state = 1771.

Основная задача была посадить корабль. Было много проблем с посадкой и принятием решением. Так как при «жетской посадке» снималось много очков, многие нейронные сети решили просто парить над площадкой тем самым не теряя эти очки как можно больше времени.

Результаты моих эксперементов:

exp	Q_param	Trajectory_n	Itern_n	Max_len	lr	Hidden_layer(1)	Hidden_layer(2)	Max mean (total_rewards)
1	0.6	50	50	400	0.01	208, 4	-	47.19
2	0.6	50	50	400	0.01	208, 104	104, 4	98.758
3	0.9	100	50	200	0.01	208, 104	104, 4	110.49
4	0.7	100	50	250	0.01	64, 104	104, 4	121
5	0.7	25	100	250	0.01	64,32	32,4	136.43
6	0.8	300	50	300	0.01	128, 64	64,4	148
7	0.9	400	70	500	0.01	128, 64	64,4	197



Я проверил множество гипотез. Изначально я провел эксперментты с моделью с одним скрытым слоем, но результат был плохой. Тогда я начал добавил второй внутренний слой, и результаты стали гораздо лучше. Очевидно, что наиболее лучшие результаты показала модель с наиболее «плотными внутренними слоями» и с наиболее широкой свободой выбора траекторий. Сложно было проводить опыты с расчетами на CPU процессоре. Для второй части будем использовать куду. Наилучшее среднее значеине было достигнуто в 197.86, что уже неплохо. По графику видно, что многие модели осталиь недообученные и добавив 100-200 эпох с скедулом на lr мы могли бы получить гораздо лучшие результаты. Возможно, для более качественной регулязации нейронной сети, также нужно было добавить пару drop-out слоев.

Задание 2.

Я выбрал задачу “Mountain Car Continuous“. Первое что, я сделал, это подгрузил torch.CUDA, так как на задание 1 я устал от ожидания результата.

Входные данные state_dim: [2] два значения.
Выходное значение action_dim [1] - одно значение
Максимальное кол-во действий в эпизоде по услвоию - 999.

Это задание отличается от всех заданий которые были раньше. У нас нет определенного действия с номером как раньше, а мы можешь выбрать любое значения от (-1 , 1). При значение от (-inf, 0) тележка движется влево, при (0, +inf) тележка будет двигаться вправо (от базового положения), при значение 0 тележка не делает ничего.

Мы можем передавать любое значение, но по условию
”The action is clipped in the range [-1,1] and multiplied by a power of 0.0015”.

Я задался вопросом, есть ли смысл ставить в конце нейронной сети функцию активации для приведения результирующего значения в промежуток от (-1, 1). Да есть смысл, посмотрев как устроен работа данной среды «под капотом» действительно идет сравнение значения действия action с (-1,1)

```
def step(self, action: np.ndarray):  self: <Continuous_MountainCarEnv<Mountain

    position = self.state[0]  position: -0.5966944728942941
    velocity = self.state[1]  velocity: 0.0
    force = min(max(action[0], self.min_action), self.max_action)  force: 1.0
```

Но награда начисляется по входному значению функции:

```
reward = 0  reward: -1000.0
if done:
    reward = 100.0
reward -= math.pow(action[0], _y: 2) * 0.1
```

Таким образом «штраф» за действие зависит от входящего значения, например

action: [0.74] toward: -0.0548
action: [0.598] toward: -0.0358
action: [0.956] toward: -0.0913
action: [0.129] toward: -0.0017
action: [-0.738] toward: -0.0545
action: [-0.275] toward: -0.0076
action: [0.708] toward: -0.0502
action: [-0.451] toward: -0.0204
action: [0.526] toward: -0.0277

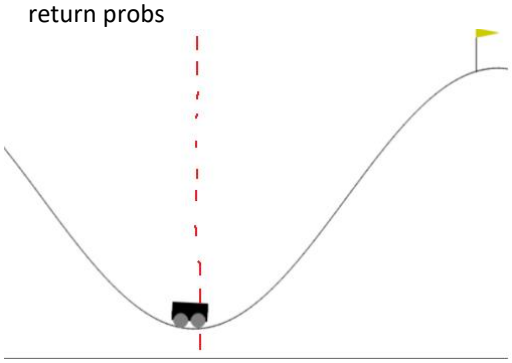
Я чуть чуть изменил нейронную сеть. Теперь она принимает 2 значения, и на выходе одно значение.

Так же изменена функция активации - на Tanh функцию, и функция ошибки loss = MSELoss()

Task 2_1

Логика обучения, и логика выбора действия была выбрана только на основе прошлого опыта, нет никакой теории исследования и случайности. Из-за такой методологии обучения модель просто катается с наименьшей амплитудой около положения равновесия и не пытается даже выбраться из ямы.

```
‘’’
def fit(self, elit_trajectories):
    elite_states = []
    elit_actions = []
    for trajectory in elit_trajectories:
        for state, action in zip(trajectory['states'], trajectory['actions']):
            elite_states.append(state)
            elit_actions.append(action)

    elite_states = np.array(elite_states)
    elite_states = torch.FloatTensor(elite_states).to(self.device)

    pred_actions = self.forward(elite_states)
    elit_actions = torch.Tensor(np.array(elit_actions)).to(self.device)
    loss = self.loss(pred_actions, elit_actions)

    loss.backward()
    self.optim.step()
    self.optim.zero_grad()
‘’’

def get_action(self, state):
    state = torch.FloatTensor(state).to(self.device)
    logit = self.forward(state)
    probs = self.Tanh(logit).detach().to('cpu').numpy()

    # action = np.random.choice(self.action_n, p=probs)

    return probs
```



Task 2_2

Я почитал в интернете, что нужно использовать метод актера и критика, но так как мы еще не обсуждали такой способ решения, я отказался от этой идеи. Я решил создать шумом кучу разных траекторий:

```
if iter < 10:
    random_mean = np.random.uniform(-1, high: 1)
    random_std_dev = np.random.uniform( low: 0,  high: 1)
    noise = np.random.normal(random_mean, random_std_dev, size=(self.max_steps,))
    noise = np.clip(noise, -1,  a_max: 1)
    TrainLoop.noise = noise
```

В результате на 500 случайных траекторий получаются 5-10 траекторий по которым шум его действительно выталкивает навверх. (См файл visualization.py) Я пытался обучить агента по таким траекториям, но все равно, что-то шло не так.