Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«Национальный исследовательский университет ИТМО»
(Университет ИТМО)

Факультет **СУ и Р**

Образовательная программа Робототехника и искусственный интеллект

О Т Ч Е Т

о производственной практике, проектно-конструкторская

Тема задания: Построение траектории движения шестиосевого манипулятора при помощи методов машинного обучения

Обучающийся Возжаева Дарья Антоновна, гр. R3235

Руководитель практики от профильной организации: **Фурзикова Светлана Сергеевна, методист**

Руководитель практики от университета: **Хитров Егор Германович, доцент**

Санкт-Петербург
2024

Содержание

ВВЕДЕНИЕ

Сегодня в повседневной жизни нас окружает все больше и больше бытовой техники, высокоточных устройств, машин и механизмов. Изготовление всех этих утсройст требует колоссальных затрат времени работника, причем чем сложнее устройство, тем более высококвалифицированный сотрудник необходим для его изготовления. Однако возможности человека, даже самого профессионально обученнного сильно ограничены. В этом случае на помощь приходит автоматизация производства: бригады заменяются роботизированными конвейерами, на смену рабочим за станками приходят роботизированные манипуляторы. Именно о таких манипуляторах пойдет речь в этой работе.

На первый взгляд может показаться, что в автоматическом управлении манипулятором нет ничего сложного, оданко это не так. Даже построения траектории движения манипулятора с 4-мя степенями свободы необходимо разрабатывать сложный алгоритм, и с каждым прибавлением степени свободы сложность растет с невероятной скоростью. Именно поэтому нам на помощь приходят методы машинного обучения. Они позволяют диллегировать расчет сложной траектории компьютеру, причем они не требуют предъявить конкртеный алгоритм построении траектории, а сами разрабатывают его.

Также отдельной задачей станет симуляция работы манипулятора. Конечно, никакие алгоритмы не совершенны, поэтому необходима среда, которая позволит отловить все неточности полученного решения, позволит не портить дорогостоящего точного робота. При этом сочленение трех звеньев с 6-ю степенями свободы требует не простой математической модели, стоит уделить этому достаточное количество внимания.

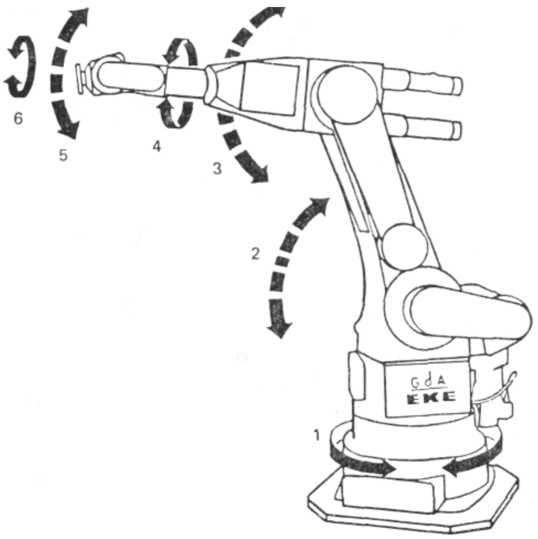

Рис. 1: Пример 6-ти осевого манипулятора

РАЗДЕЛ ПЕРВЫЙ

В машинном обучении выделяют три основыне категории по обучающим данным:

- Надзорное обучение - заранее известен правильный ответ на поставленную задачу, модель учится массиве готовых решений
- Ненадзорное обучение - заранее неизвестен правильный ответ, модель обучается на большом массиве данных и выявляет в них зависимости
- Подкрепляющее обучение - заранее нет ни правльных ответов, ни массива данных, модель пробует различные стратегии и выбирает наиболее выгодные

В нашем очень сложно добыть готовый массив данных, и тем более иметь уже готовый пул правильных решений, поэтому обратимся к обучению с подкреплением.

Итак, алгоритмы обучения с подкреплением требуют двух трех основный составляющих: среда, агент и вознаграждение, зададимся ими. В качестве среды у нас выступает трехмерное пространство, в качестве агента модель 6-ти осевого манипулятора, а вознаграждение зависит от близости манипулятора к необходимому положению.

Начнем по порядку: агент обладает двумя основными характеристиками, а именно $V \in \mathbb{R}^n$ - состояние и $A \in \mathbb{R}^m$ - действие. Также нам необходима функция вознаграждения $R : \mathbb{R}^m \to \mathbb{R}$, то есть награда за некоторое действие $r_n = R(A_n)$, тогда награда за сессию из N действий равна $R(\{A_i\}_{i=0}^N) = \sum_{i=0}^N R(A_i)$.

В таком случае можем заадать функцию политики агента $\pi : \mathbb{R}^n \to \mathbb{R}^m$, то етсь $\pi(V) = A$, и задачей обучения станет поиск $\arg\max\limits_{\pi} \left(R^\pi(\{A_i\}_{i=0}^N)\right)$

Метод Кросс-энтропии

Метод Кросс-энтропии заключается в двух этапах:

- Policy evaluation - оценка эффективности политики.
  По нынешней политике генерируется K сессий по N действий в каждом и считается эффективность политики R^π
- Policy improvement - корректировка политики. При помощи q-квантили отбирается набор элитных траекторий и по ним происходит корректировка политики

В слуачае с конечным числом состояний и действий, π могла бы быть просто матрицей, но в нашем случае мы имеем дело с конечномерными пространствами состояний и действий, поэтому для определения политики применим нейросеть.

Структура нейросети

Принцип работы нейросети заключается в применении функции активации к взвешенному набору признаков с некоторым смещением. То есть один нейрон является некоторой $f : \mathbb{R}^n \to \mathbb{R} : f(x) = \gamma(b + \sum_{i=0}^n \omega_i x_i)$, причем γ - функция активации. Тогда дним слоем является $F = (f^1, f^2, ..., f^m)^T : \mathbb{R}^n \to \mathbb{R}^m$. Мнлжество нейронных слоев создают сеть.

Для того, что бы сконструировать нейроннуб сеть, в первую очередь надо задаться функцией активации, как правило на внутренних слоя используется одна и та же. Перечислим основные их виды:

- $f(x) = x$
- $f(x) = \frac{1}{1+e^{-x}}$
- $f(x) = \tan^{-1}(x)$
- $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Однако мы выберем в качестве функции активации функцию линейного выпрямителя:

$$f(x) = \begin{cases} x, x \geq 0 \\ 0, x < 0 \end{cases}$$

У такой функции ест ряд преимуществ, она монотонна дифференцируема, и основной из них - ее производная так же монотонная. Заметим, что дифференцируемость функции чрезвычайно важна. Перед нами стоит задачи максимизации функции награды, а значит для поиска решения мы воспользуемся методами из класса градиентных спусков. Таким образом дифференцируемость поможет нам вычислять градиент аналитически, не затрачивая вычислительные ресурсы на подсчет численной производной.

Градиентный спуск

Определим функцию потерь. Пусть $E = (V^1, A_1, V^2, A_2, ..., A^n)$ некоторая элитная траектория. Тогда функцией потерь назовем $Loss(\pi) = \frac{1}{n}\sum_{i=1}^{n} \|F^{\pi}(V^i) - A^i\|^2$

Итак, получаем следующий алгоритм:

1. Строим n траекторий по нынешней политике $\{E_i^{\pi}\}_{i=1}^{n}$
2. Оцениваем эффективность политики $R(\{A_i\}_{i=1}^{n})$
3. При помощи q-квантили отбираем набор элитных траекторий $\{E_i^{\pi}\}_{i \in I}$
4. Делаем один шаг градинетного спуска, в сторону градинта функции $Loss(\pi)$

Реализацию данного алгоритма можно увидеть в Приложении 1

РАЗДЕЛ ВТОРОЙ

Итак, мы разобрались с политикой действий агента, однако необходимо задаться еще и средой, для этого построим математическую модель манипулятора. Начнем с того, что изобразим его схематично:

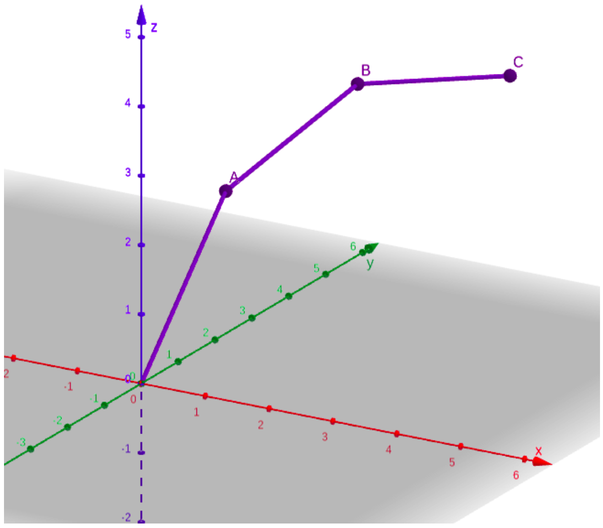

Рис. 2: Схематичное изображение манипулятора

Состояние манипулятора задается тремя точками A, B, и C, а также точкой T - целевой точкой, которая должна по итогу совпасть с C. Таким образом для задания состояния робота нам необходим 12-мерный вектор, то есть $V \in \mathbb{R}^{12}$. Пространство действий шестимерно: каждое из звеньев может поворачиваться вокруг своей оси на угол θ и наклоняться на угол α. Таким образом мы определили пространство состояний и пространство действий:

$$\begin{aligned} V =&(A_x, A_y, A_z, B_x, B_y, B_z, C_x, C_y, C_z, T_x, T_y, T_z) \in \mathbb{R}^{12} \\ A =&(\alpha_1, \alpha_2, \alpha_3, \theta_1, \theta_2, \theta_3) \in \mathbb{R}^6 \end{aligned}$$

Для того, что бы задать среду, нам необходимо определить функцию $step(V^n, A^n) = (V^{n+1}, R(A^n))$

Наклон звена

Объяснимся, как на состояние подействует каждое из действий, начнем с наклона звена.
Для того, что бы осуществить поворот, перейдем в влокальную систему координат, связанную с началом поворачивамого звена, повернем его на необходимый градус, и затем вернемся в глобальную систему координат. Рассмотрим на примере поворота второго звена:

$$basis = \begin{pmatrix} | & | & | \\ \frac{[\vec{OA}, \vec{OB}]}{\|[\vec{OA}, \vec{OB}]\|} & \frac{\vec{OB} - \vec{OA}}{\|\vec{OB} - \vec{OA}\|} & \left[\frac{[\vec{OA}, \vec{OB}]}{\|[\vec{OA}, \vec{OB}]\|}, \frac{\vec{OB} - \vec{OA}}{\|\vec{OB} - \vec{OA}\|}\right] \\ | & | & | \end{pmatrix}$$

Таким образом в качестве первого базисного орт мы выбираем вектор нормали к плоскости, образованной векторами $\vec{OA}$ и $\vec{OB}$, в качестве второго нормированный вектор $\vec{AB}$, а в качестве третьего базисного вектора выступает векторное произведение первых двух. Таким образом мы обесвечиваем себя ортонормированным базисом, а значит для поворота нам понадобиться классическая мтрица поворота вокруг первого базисного вектора:

$$rotate(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

Таким образом оператор поворота это композиция преобразований:

$$\vec{OA}_{\alpha_2} = \vec{OA}$$
$$\vec{OB}_{\alpha_2} = basis * (rotate(\alpha_2) * (basis^{-1} * (\vec{OB} - \vec{OA}))) + \vec{OA}$$
$$\vec{OC}_{\alpha_2} = basis * (rotate(\alpha_2) * (basis^{-1} * (\vec{OC} - \vec{OA}))) + \vec{OA}$$

Аналогичным образом определим пнаклон первого звена:

$$\vec{OA}_{\alpha_1} = basis * (rotate(\alpha_1) * (basis^{-1} * (\vec{OA})))$$
$$\vec{OB}_{\alpha_1} = basis * (rotate(\alpha_1) * (basis^{-1} * (\vec{OB})))$$
$$\vec{OC}_{\alpha_1} = basis * (rotate(\alpha_1) * (basis^{-1} * (\vec{OC})))$$

И третьего звена:

$$\vec{OA}_{\alpha_3} = \vec{OA}$$
$$\vec{OB}_{\alpha_3} = \vec{OB}$$
$$\vec{OC}_{\alpha_3} = basis * (rotate(\alpha_2) * (basis^{-1} * (\vec{OC} - \vec{OB}))) + \vec{OB}$$

Заметим, что матрица базисных векторов basis при наклоне разных звеньев отличается, и определяется отдельного для каждого звена.

Вращение звена

Теперь определим вращение звена вокруг собственной оси. Подметим, что хоть и кажется, что это простое преобразование, поскольку оно не влияет на само звено, но при этом оно поворачивает все последующие вокруг повернуой в ропстранстве оси.

Внось восполльзуемся сменой базиса и матрицей поворота, только на этот раз первый базисный вектор будет связан не с перпендикуляром к наклоняемому звену, а с самим вращаемым звеном. Вновь разберем на примере поворота второго звена:

$$basis = \begin{pmatrix} | & | & | \\ \frac{\vec{OA}}{\|\vec{OA}\|} & \frac{[\vec{OA},\vec{OB}]}{\|[\vec{OA},\vec{OB}]\|} & \left[\frac{[\vec{OA},\vec{OB}]}{\|[\vec{OA},\vec{OB}]\|}, \frac{\vec{OA}}{\|\vec{OA}\|}\right] \\ | & | & | \end{pmatrix}$$

Матрица поворота:

$$rotate(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix}$$

Посмотрим, как повлияют поворот второго звена вокург своей оси на состояние системы:

$$\vec{OA}_{\theta_2} = \vec{OA}$$
$$\vec{OB}_{\theta_2} = basis * (rotate(\alpha_2) * (basis^{-1} * (\vec{OB} - \vec{OA}))) + \vec{OA}$$
$$\vec{OC}_{\theta_2} = basis * (rotate(\alpha_2) * (basis^{-1} * (\vec{OC} - \vec{OA}))) + \vec{OA}$$

Аналогичным образом определяется поворот первого звена:

$$\vec{OA}_{\theta_1} = basis * (rotate(\alpha_2) * (basis^{-1} * (\vec{OA})))$$
$$\vec{OB}_{\theta_1} = basis * (rotate(\alpha_2) * (basis^{-1} * (\vec{OB})))$$
$$\vec{OC}_{\theta_1} = basis * (rotate(\alpha_2) * (basis^{-1} * (\vec{OC})))$$

И поворот третьего звена: И третьего звена:

$$\vec{OA}_{\theta_3} = \vec{OA}$$
$$\vec{OB}_{\theta_3} = \vec{OB}$$
$$\vec{OC}_{\theta_3} = basis * (rotate(\alpha_2) * (basis^{-1} * (\vec{OC} - \vec{OB}))) + \vec{OB}$$

Симуляция работы манипулятора

Посмотрим на то, как физически выглядят наклоны и повороты отдельных звеньев. Для это обратимся к результатам работы кода из Приложений 2 и 3:

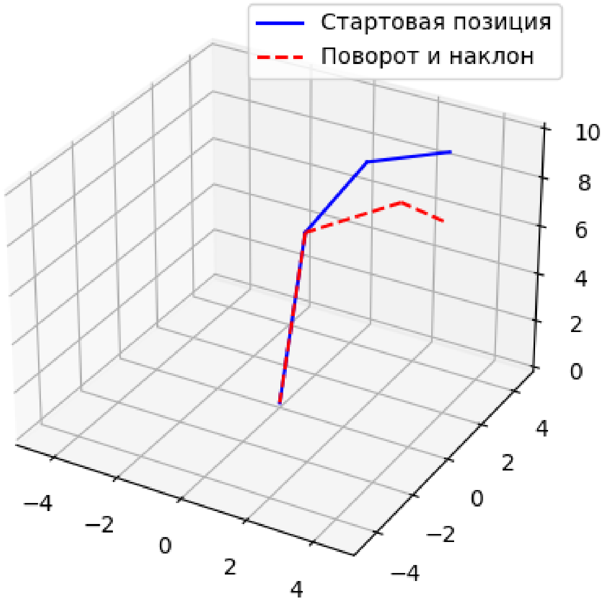

Рис. 3: Поворот 2-го звена на 60 градусов

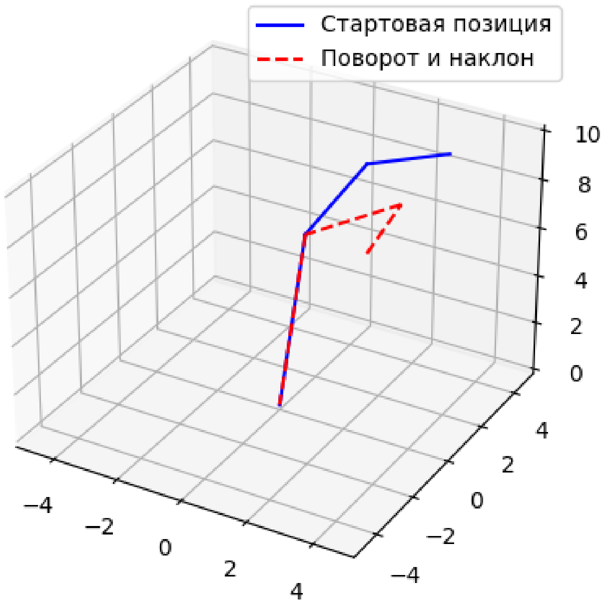

Рис. 4: Поворот 2-го и 3-го звеньев на 60 градусов

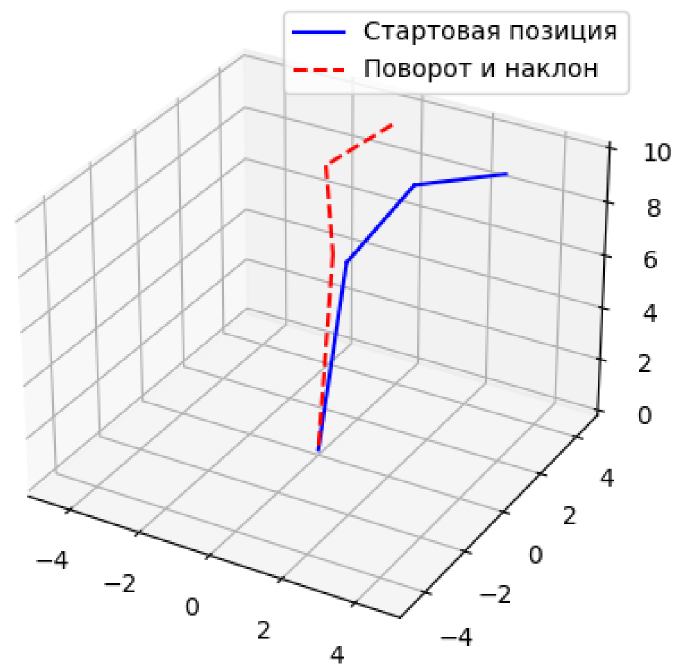

Рис. 5: Поворот 1-го звена на 45 градусов

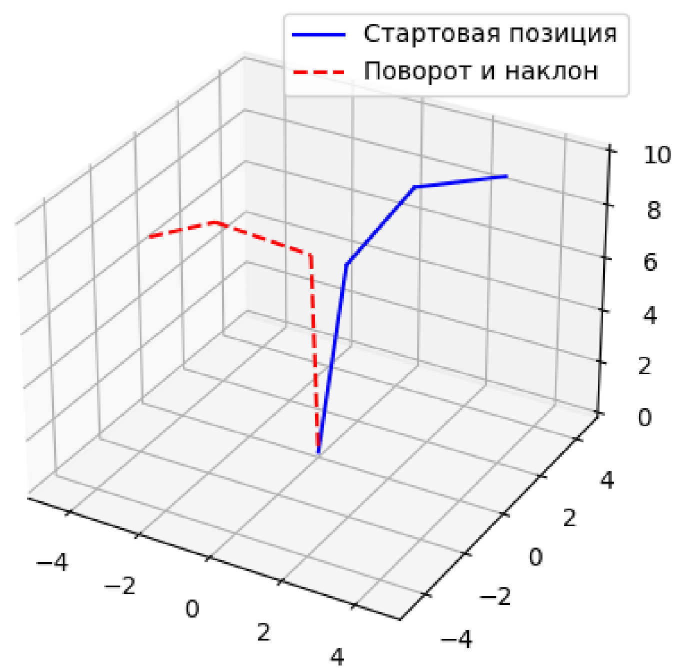

Рис. 6: Поворот 1-го звена на 90 градусов, 2-го на 60 и 3-го на 90

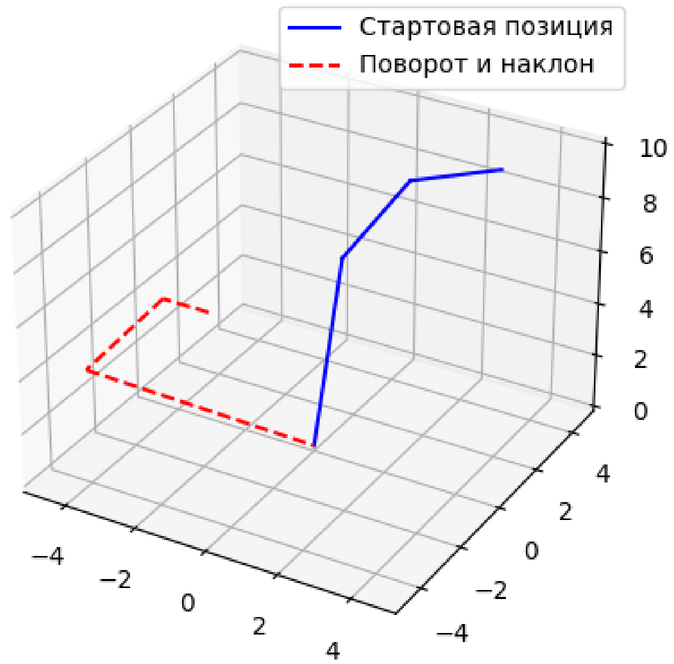

Рис. 7: Наклон 1-го и 2-го звеньев на 60 градусов и 3-го на 45

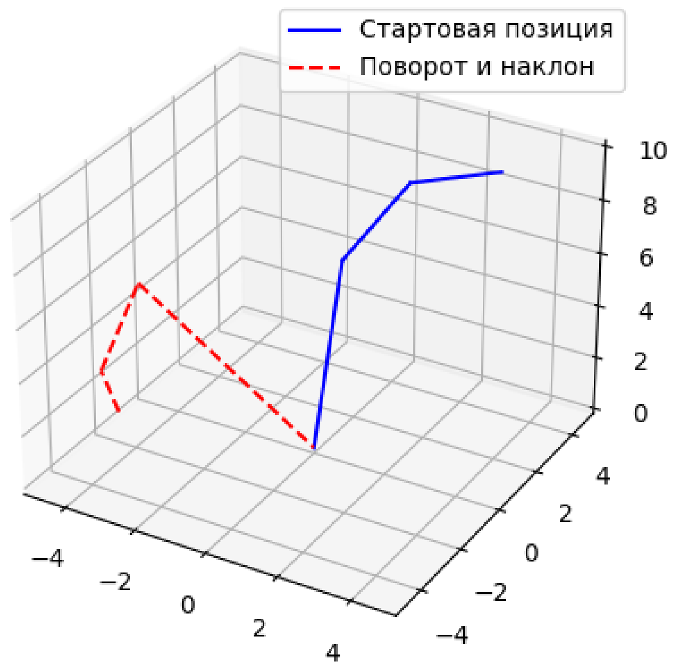

Рис. 8: Поворот 1-го звена на 120 градусов и наклон 1-го и 2-го звеньев на 60

РАЗДЕЛ ТРЕТИЙ

Анализ гиперпараметров модели

При обучении, на изменение политики в основном влияет параматр q, он отвечает за выбор элитных траекторий. Изучим его влияние на обучение при разных параметрах, для этого построим графики зависимости средней ошибки от номера эпизода:

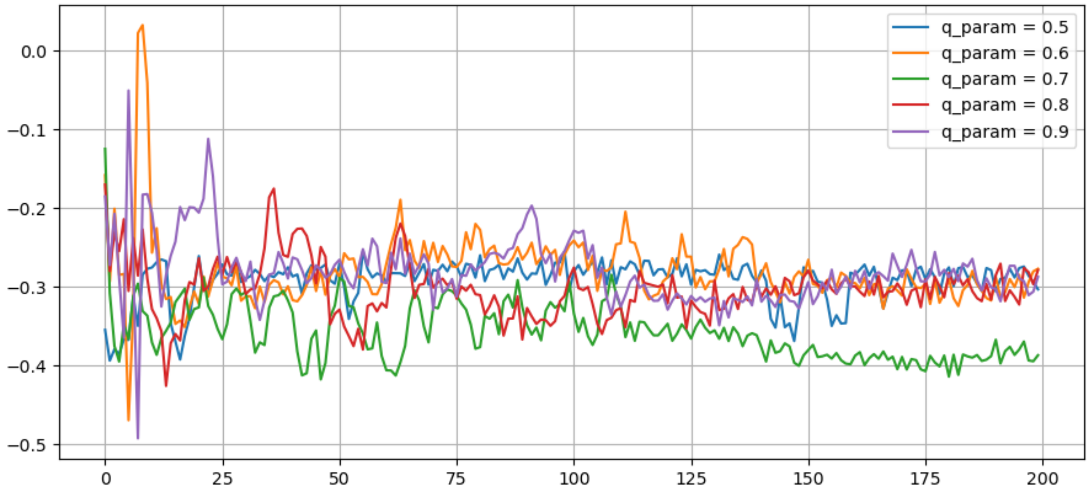

Рис. 9: Значение средней награды при количестве эпизодов 200, количестве сессий 200 и длинне сессии 100

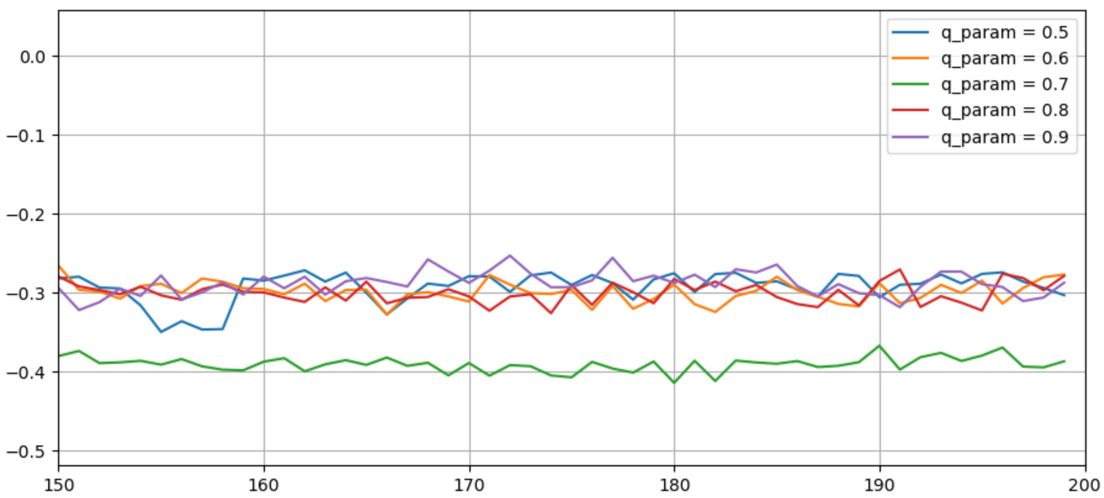

Рис. 10: Последние 50 значений средней награды при количестве эпизодов 200, количестве сессий 200 и длинне сессии 100

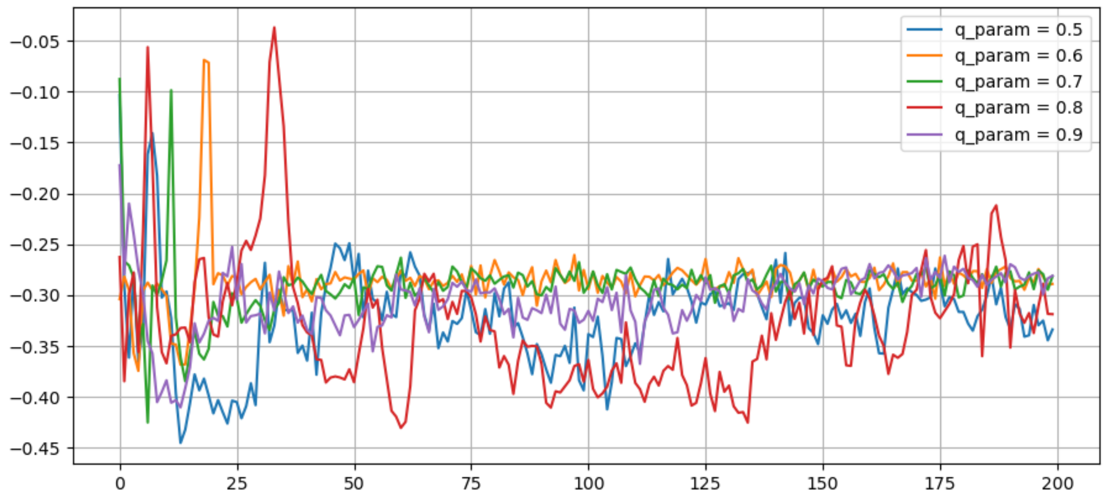

Рис. 11: Значение средней награды при количестве эпизодов 200, количестве сессий 200 и длинне сессии 300

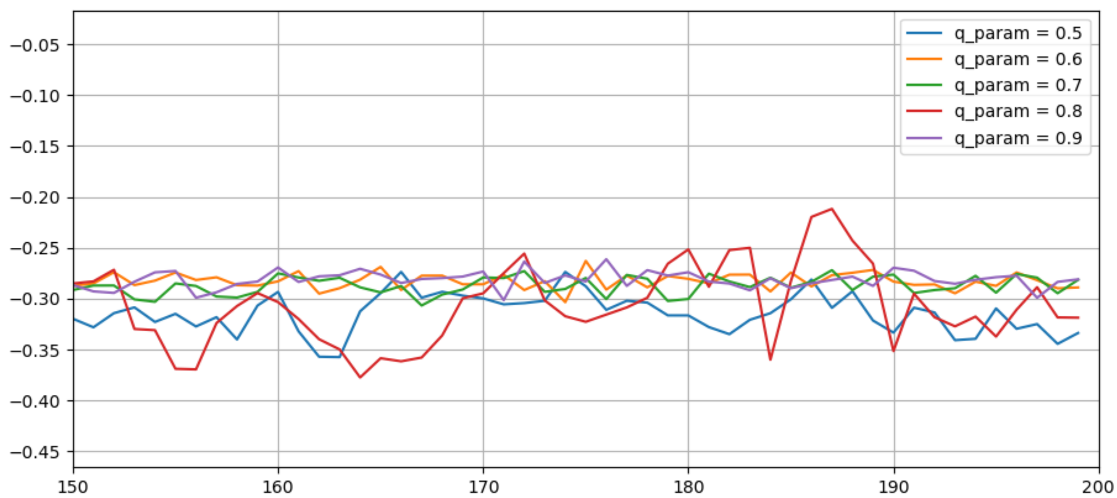

Рис. 12: Последние 50 значений средней награды при количестве эпизодов 200, количестве сессий 200 и длинне сессии 300

Как можно увидеть на графиках, наилучшим варинатом во всех случаях является значение параметра q = 0.9. Почему же так происходит? Дело в том, что мы как бы выделяем из всех траекторий долю лучших равную q. Если мы выбираем q слишком маленьким, при нашей постановке задачи, действительно удачные траектории маловероятно попадают в выборку, а значит и обучение происходит малоэффективно. Эта особенность связана с тем, что манипулятор фактически случайными блужданиями должен в буквальном смысле наткнуться на точку назначения, чего шанс крайне небольшой, а значит нам нельзя рисковать и ставить коэффициент q слишком маленьким.

Теперь посмотрим, как влияет количество сессий в эпизоде на качество обучения модели:

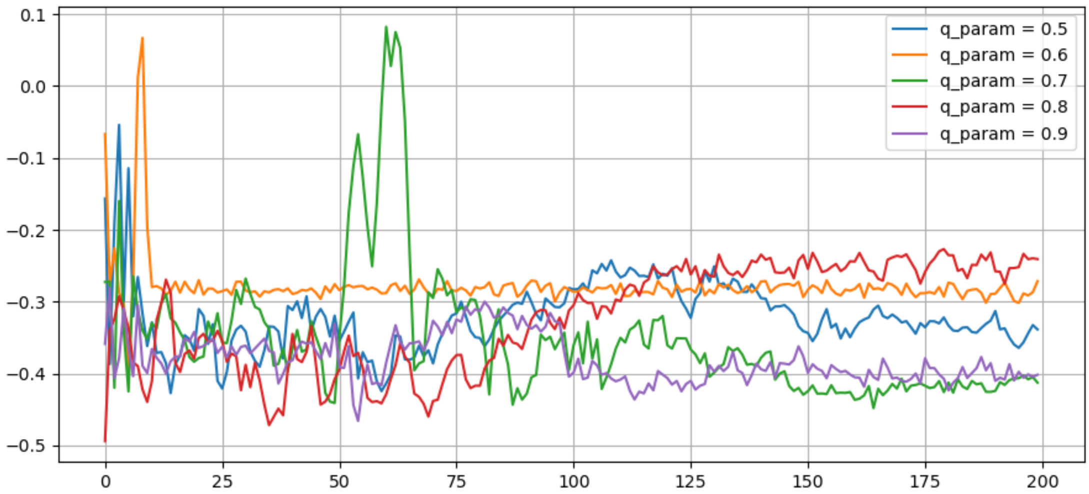

Рис. 13: Значение средней награды при количестве эпизодов 200, количестве сессий 500 и длинне сессии 300

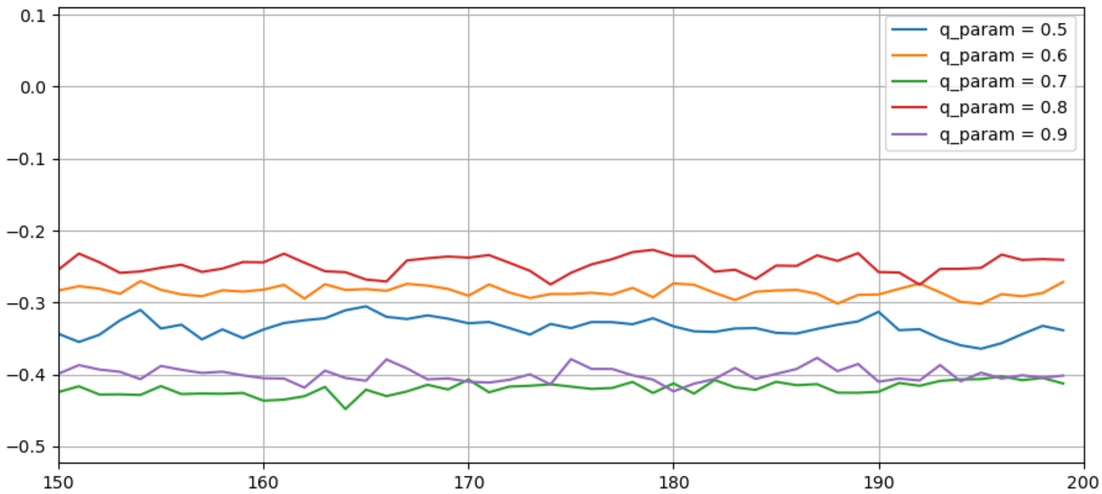

Рис. 14: Последние 50 значений средней награды при количестве эпизодов 200, количестве сессий 500 и длинне сессии 300

Итак, на первый взгляд может показаться, что ничего не поменялось, но это не так: увеличилось значение средней награды. Если в случае длинны эпизода равной 200 сессий средняя награда к концу обчения принадлежала промежутку (-0.3, -0.25) (при наилучшем q), то при увеличении длинны эпизода она выросла до (-0.25, 0.2). Таким образом стоит увеличение количества сессий в эпизоде приводит к улучшению показателей модели.

Вновь проанализируем параметр q. Как и в прошлый раз, наиболее выгодным стало одно из больших значение q, то есть $q = 0.8$. Однако неожиданным оказался результат при $q = 0.9$: он оказался вообще одним из наихудших. Почему же так происходит? Дело в том, что скорость схождения модели при градиентном спуске сильно зависит от того, насколько удачно были выбраны начальные веса. Поскольку веса в начале выбираются случайным образом пир помощи нормального распределения, можно сделать предположение, что именно при этом запуске обучения они вышли сильно далекими от экстремума функции потерь.

ВЫВОДЫ

Итак, в заключение, хочется сказать, что мне удалось выполнить комплексную задачу. Во-первых, конечно, одним из результатов работы стала среда, позволяющая полностью симулировать работу манипулятора. Построенный мною симулятор позволяет управлять каждой из осей робота, и что важно, хранит информацию о его положении в декартовой системе координат. Удобным бонусом стала визуализация траектории на упрощенной 3D схеме манипулятора. Хочется отметить, что программа позводяет задать собственные параметры манипулятора: его размеры, начальное положение, ограничения на диапозон вращений каждой из осей. Таким образом ее можно применить для большого количества разных манипуляторов.

Во-вторых, я изучила большое количество информации о машинном обучении, что позволило мне создать и обучить модель, основанную на обучении с подкреплением и нейросети. Для построенной модели я подобрала гиперпараметры, обеспечивающие ей хорошие показатели оценки работы. Хотя модель и достигла неплохих показателей, хочется отметить, что рациональность применения именно таких алгоритмов остается под вопросом, поскольку требует огромных затрат на вычисления при обучении модели.

Также можно отметить и положительную сторону такого подхода. Дело в том, что в популярных алгоритмах построения траектории есть существенный недочет, робот не может с состоянием сингулярности. Так называется положение робота,к когда каждый из приводов находится на одной оси, что приводит к некоторой случайной смене положения манипулятором только для того, что бы выфти из этого положения. В случае с алгоритмом, раситанным моделью, подобного не случится, поскольку она планирует ввсю траекторию сразу же, оппираясь только на нчальную и конечную точки.

ПРИЛОЖЕНИЕ 1

```
import gym
import time
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch import nn

class CrossEnthropyNNAgent(nn.Module):
    def __init__(self, state_dim, action_n, action_max, grad_step=0.01):
        super().__init__()
        hidden1 = 200
        hidden2 = 100
        hidden3 = 200
        self.network = nn.Sequential(
            nn.Linear(state_dim, hidden1),
            nn.ReLU(),
            nn.Linear(hidden1, hidden2),
            nn.ReLU(),
            nn.Linear(hidden2, hidden3),
            nn.ReLU(),
            nn.Linear(hidden3, action_n)
        )
        self.softmax = nn.Softmax()
        self.loss = Loss_func
        self.optimizer = torch.optim.Adam(self.parameters(), lr=grad_step)
        self.action_max = action_max




    def forward(self, input):
        return self.network(input)

    def get_action(self, state):
        state = torch.FloatTensor(state)
        logits = self.network(state)
        action = torch.atan(logits).detach().numpy() * self.action_max
        return action

    def update_policy(self, elite_sessions):
        elite_states, elite_actions = [], []
        for session in elite_sessions:
            elite_states.extend(session['states'])
            elite_actions.extend(session['actions'])

        elite_states = torch.FloatTensor(elite_states)
        elite_actions = torch.FloatTensor(elite_actions)

        loss = self.loss(self.network(elite_states), elite_actions)
        loss.backward()
        self.optimizer.step()
        self.optimizer.zero_grad()
        return None

def Loss_func(calculated_actions, elite_actions):
    res = 0
    for i in range(len(elite_actions)):
```

```python
        res += (calculated_actions[i] - elite_actions[i]).norm() ** 2
    return res / len(elite_actions)


def get_session(env, agent, session_len, visual=False):
    session = {}
    states, actions = [], []
    total_reward = 0

    state = env.reset()
    for _ in range(session_len):
        states.append(state)
        action = agent.get_action(state)
        actions.append(action)

        #if visual:
        #    env.render()
        state, reward, done= env.step(action)
        total_reward += reward

        if done:
            break

    session['states'] = states
    session['actions'] = actions
    session['total_reward'] = total_reward
    return session

def get_elite_sessions(sessions, q_param):
    total_rewards = np.array([session['total_reward'] for session in sessions])
    quantile = np.quantile(total_rewards, q_param)

    elite_sessions = []
    for session in sessions:
        if session['total_reward'] > quantile:
            elite_sessions.append(session)

    return elite_sessions
```

ПРИЛОЖЕНИЕ 2

```
import gym
import time
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch import nn

class Robot:
    def __init__(self, end_1, end_2, end_3):
        self.end_1 = end_1
        self.end_2 = end_2
        self.end_3 = end_3

        self.slope_1, self.slope_2, self.slope_3 = 0, 0, 0
        self.rot_1, self.rot_2, self.rot_3 = 0, 0, 0

    def draw_line(self, a, b, ax, col, style):
        ax.plot([a[0], b[0]], [a[1], b[1]], [a[2], b[2]], color=col, linestyle=style)
        return None

    def show(self, ax, col='b', style='solid'):
        self.draw_line(self.end_1, [0, 0, 0], ax, col, style)
        self.draw_line(self.end_2, self.end_1, ax, col, style)
        self.draw_line(self.end_3, self.end_2, ax, col, style)
        return None

    def rotate(self, rot_ax, vec, alpha):
        base1 = rot_ax
        base2 = np.cross(rot_ax, vec)
        base3 = np.cross(base1, base2)

        base1 = base1 / np.linalg.norm(base1)
        base2 = base2 / np.linalg.norm(base2)
        base3 = base3 / np.linalg.norm(base3)

        base_mat = np.transpose(np.matrix([base1, base2, base3]))
        base_inv = np.linalg.inv(base_mat)

        rot_mat = np.matrix([[1, 0, 0], [0, np.cos(alpha), -np.sin(alpha)], [0, np.sin(a

        new_vec = base_mat * (rot_mat * (base_inv * np.transpose(np.matrix(vec - rot_ax)

        rot_vec = np.array(np.transpose(new_vec))[0] + rot_ax

        return rot_vec

    def rotate_first_el(self, alpha):
        if self.rot_1 + alpha > np. pi:
            alpha = np.pi - self.rot_1
        elif self.rot_1 + alpha < - np.pi:
            alpha = - np.pi - self.rot_1

        base_vec = np.array([0, 0, 1])
        rot_end_1 = self.rotate(base_vec, self.end_1, alpha)
        rot_end_2 = self.rotate(base_vec, self.end_2, alpha)
        rot_end_3 = self.rotate(base_vec, self.end_3, alpha)
```

```
        self.end_1 = rot_end_1
        self.end_2 = rot_end_2
        self.end_3 = rot_end_3
        self.rot_1 += alpha

    def rotate_second_el(self, alpha):
        if self.rot_2 + alpha > np. pi:
            alpha = np.pi - self.rot_2
        elif self.rot_2 + alpha < - np.pi:
            alpha = - np.pi - self.rot_2

        rot_end_2 = self.rotate(self.end_1, self.end_2, alpha)
        rot_end_3 = self.rotate(self.end_1, self.end_3, alpha)

        self.end_2 = rot_end_2
        self.end_3 = rot_end_3
        self.rot_2 += alpha
        return None

    def rotate_third_el(self, alpha):
        if self.rot_3 + alpha > np. pi:
            alpha = np.pi - self.rot_3
        elif self.rot_3 + alpha < - np.pi:
            alpha = - np.pi - self.rot_3

        rot_end_3 = self.rotate(self.end_2, self.end_3, alpha)

        self.end_3 = rot_end_3
        self.rot_3 += alpha
        return None

    def slope_first_el(self, alpha):
        if self.slope_1 + alpha > np. pi:
            alpha = np.pi - self.slope_1
        elif self.slope_1 + alpha < - np.pi:
            alpha = - np.pi - self.slope_1

        base1 = np.array([self.end_1[0], self.end_1[1], 0])
        base2 = np.array([0, 0, 1])
        base3 = np.cross(base1, base2)


        base1 = base1 / np.linalg.norm(base1)
        base2 = base2 / np.linalg.norm(base2)
        base3 = base3 / np.linalg.norm(base3)

        base_mat = np.transpose(np.matrix([base3, base2, base1]))
        base_inv = np.linalg.inv(base_mat)

        rot_mat = np.matrix([[1, 0, 0], [0, np.cos(alpha), -np.sin(alpha)], [0, np.sin(a
.matrix(self.end_1)))))
        new_end_1 = np.array(np.transpose(new_end_1))[0]
        new_end_2 = base_mat * (rot_mat * (base_inv * (np.transpose(np.matrix(self.end_2
        new_end_2 = np.array(np.transpose(new_end_2))[0]
        new_end_3 = base_mat * (rot_mat * (base_inv * (np.transpose(np.matrix(self.end_3
        new_end_3 = np.array(np.transpose(new_end_3))[0]

        self.end_1 = new_end_1
        self.end_2 = new_end_2
```

```
        self.end_3 = new_end_3
        self.slope_1 += alpha

        return None

    def slope_second_el(self, alpha):
        if self.slope_2 + alpha > np. pi:
            alpha = np.pi - self.slope_2
        elif self.slope_2 + alpha < - np.pi:
            alpha = - np.pi - self.slope_2

        base1 = np.cross(self.end_1, self.end_2)
        base2 = self.end_1
        base3 = np.cross(base1, base2)

        base1 = base1 / np.linalg.norm(base1)
        base2 = base2 / np.linalg.norm(base2)
        base3 = base3 / np.linalg.norm(base3)

        base_mat = np.transpose(np.matrix([base1, base2, base3]))
        base_inv = np.linalg.inv(base_mat)

        rot_mat = np.matrix([[1, 0, 0], [0, np.cos(alpha), -np.sin(alpha)], [0, np.sin(a

        new_end_2 = base_mat * (rot_mat * (base_inv * (np.transpose(np.matrix(self.end_2
        new_end_2 = np.array(np.transpose(new_end_2))[0] + self.end_1

        new_end_3 = base_mat * (rot_mat * (base_inv * (np.transpose(np.matrix(self.end_3

        new_end_3 = np.array(np.transpose(new_end_3))[0] + self.end_1

        self.end_2 = new_end_2
        self.end_3 = new_end_3
        self.slope_2 += alpha

        return None

    def slope_third_el(self, alpha):
        base1 = np.cross(self.end_2 - self.end_1, self.end_3 - self.end_1)
        base2 = self.end_2 - self.end_1
        base3 = np.cross(base1, base2)

        base1 = base1 / np.linalg.norm(base1)
        base2 = base2 / np.linalg.norm(base2)
        base3 = base3 / np.linalg.norm(base3)

        base_mat = np.transpose(np.matrix([base1, base2, base3]))
        base_inv = np.linalg.inv(base_mat)

        rot_mat = np.matrix([[1, 0, 0], [0, np.cos(alpha), -np.sin(alpha)], [0, np.sin(a

        new_end_3 = base_mat * (rot_mat * (base_inv * (np.transpose(np.matrix(self.end_3
        new_end_3 = np.array(np.transpose(new_end_3))[0] + self.end_2
        self.slope_3 += alpha

        self.end_3 = new_end_3

        return None
```

ПРИЛОЖЕНИЕ 3

```
import gym
import time
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch import nn


class Environment():
    def __init__(self):
        self.robot = None
        self.start_pos = None
        self.target = None
        self.state = np.zeros(12)
        self.max_len = 0
        return None

    def robot_On(self, a, b, c):
        self.robot = Robot(a, b, c)
        self.start_pos = [a, b, c]
        self.state[0:3] = a
        self.state[3:6] = b
        self.state[6:9] = c
        self.max_len = np.linalg.norm(a) + np.linalg.norm(b) + np.linalg.norm(c)
        return None

    def set_target(self):
        max_len = np.linalg.norm(self.robot.end_1) + np.linalg.norm(self.robot.end_2)
        x = np.random.rand() * max_len
        max_len -= np.linalg.norm(x)
        y = np.random.rand() * max_len
        max_len -= np.linalg.norm(y)
        z = np.random.rand() * max_len

        self.target = np.array([x, y, z])
        self.state[9:12] = self.target
        return None

    def reset(self):
        self.robot_On(self.start_pos[0], self.start_pos[1], self.start_pos[2])
        self.set_target()
        return self.state

    def step(self, action):
        prev_dist = np.linalg.norm(self.state[9:12] - self.state[6:9])

        self.robot.slope_first_el(action[0])
        self.robot.slope_second_el(action[1])
        self.robot.slope_third_el(action[2])

        self.robot.rotate_first_el(action[3])
        self.robot.rotate_second_el(action[4])
        self.robot.rotate_third_el(action[5])

        self.state[0:3] = self.robot.end_1
        self.state[3:6] = self.robot.end_2
        self.state[6:9] = self.robot.end_3
```

```python
dist = np.linalg.norm(self.state[9:12] - self.state[6:9])

if dist < 0.1:
    reward = 100
    done = True
    return self.state, reward, done

reward = (prev_dist - dist) / self.max_len
done = False
return self.state, reward, done
```