

Санкт-Петербургский государственный университет

Кафедра прикладной кибернетики

Группа 24.Б82-мм

Алгоритмы решения задачи стабилизации стержня на подвижном основании

Черняков Евгений Олегович

Отчёт по учебной практике
в форме «Сравнение»

Научный руководитель:
профессор кафедры прикладной кибернетики Мокаев Р. Н.

Санкт-Петербург
2025

Оглавление

Введение	3
1. Постановка задачи	5
2. Обзор литературы и методов	6
2.1. Обзор существующих решений	6
2.2. Обзор используемых технологий и метрик	8
2.3. Выводы	12
3. Описание решения	13
3.1. Моделирование системы	13
3.2. Программная реализация модели	21
3.3. Реализация алгоритмов обучения	24
4. Эксперимент	37
4.1. Условия эксперимента и подбор гиперпараметров	37
4.2. Сравнительный анализ метрик	41
4.3. Результаты	43
4.4. Выводы и оценка целесообразности использования градиентных методов	48
5. Заключение	50
Список литературы	52

Введение

Задача стабилизации стержня на подвижной тележке (Cart-Pole) является классическим эталоном в теории автоматического управления и обучении с подкреплением. Она моделирует неустойчивые нелинейные системы, принципы управления которыми лежат в основе работы балансирующих роботов и ракет-носителей на старте. Традиционные методы решения, такие как ПИД-регуляторы или LQR-алгоритмы, требуют точного знания математической модели объекта. Однако с развитием искусственного интеллекта всё большую популярность приобретают методы, способные обучаться управлению через взаимодействие со средой, не требуя явного задания уравнений динамики. Существующие исследования широко освещают применение глубокого обучения с подкреплением (Deep RL) для этой задачи, демонстрируя возможности сложных нейросетевых архитектур.

В то же время, в учебной и технической литературе часто наблюдается пробел в сравнительном анализе эффективности базовых стохастических методов оптимизации по сравнению с градиентными методами начального уровня. Нередко сложные алгоритмы применяются там, где достаточно простейшего случайного поиска. Остается открытый вопрос: оправдано ли усложнение алгоритма (переход от случайногопоиска к градиентной политике) для системы с малым количеством степеней свободы.

Целью данной работы является выявление сравнительной эффективности алгоритмов различной природы — базового случайного поиска (Basic Random Search), метода восхождения на холм (Hill Climbing) и градиентной политики (Actor-Critic) — в задаче балансировки стержня. Необходимо определить, какой из подходов обеспечивает наилучший баланс между скоростью обучения и устойчивостью удержания стержня. Работа призвана продемонстрировать, что для определенных классов задач простые безградиентные методы могут составлять конкуренцию более сложным подходам RL.

Для реализации поставленной задачи в работе разрабатывается про-

граммная модель динамической системы, основанная на численном решении дифференциальных уравнений движения стержня и тележки. В данную среду интегрируются и настраиваются алгоритмы Basic Random Search, Hill Climbing и Actor-Critic. В ходе экспериментов проводится обучение агентов с фиксацией ключевых метрик: На основе полученных данных будет проведен сравнительный анализ и сформулированы выводы о применимости каждого из методов для задач стабилизации неустойчивых динамических систем.

1. Постановка задачи

Целью работы является проведение сравнительного анализа эффективности алгоритмов стохастической оптимизации и методов обучения с подкреплением при решении задачи стабилизации стержня на подвижном основании. Для её выполнения были поставлены следующие задачи:

1. математически описать динамику системы «тележка-стержень», составив дифференциальные уравнения движения;
2. создать программную среду, эмулирующую поведение системы во времени на основе разработанной математической модели, без использования сторонних физических движков;
3. программно реализовать алгоритмы: Basic Random Search, Hill Climbing, Actor-Critic;
4. провести серию экспериментов по обучению агентов каждым из алгоритмов, подобрав гиперпараметры и зафиксировав метрики;
5. сравнить полученные данные, выявить преимущества и недостатки каждого алгоритма и оценить целесообразность использования градиентных методов для задач данной размерности.

2. Обзор литературы и методов

Целью данного раздела является формирование необходимого теоретического фундамента для понимания работы и обоснование методологического выбора, сделанного в рамках исследования. Обзор систематизирует известную информацию о механической системе «тележка–стержень», существующих подходах к её стабилизации и принципах работы выбранных алгоритмов обучения. Обзор был выполнен с использованием следующих методов:

- **Систематический поиск:** использовались научные базы данных (например, Google Scholar, Scopus, ResearchGate) по ключевым запросам, охватывающим как классическую теорию управления, так и современное обучение с подкреплением (например, «Cart-Pole stabilization», «Inverted Pendulum control», «Policy Gradient vs Random Search»).
- **Оценка релевантности:** в фокусе внимания находились статьи и монографии, описывающие математическую модель системы со стержнем с распределённой массой [11], а также работы, напрямую сравнивающие эффективность стохастической оптимизации (**Random Search, Hill Climbing**) и градиентных методов [9, 5].
- **Идентификация технологий:** изучались принципы работы выбранных алгоритмов (**AC, HC, BRS**) и современные методы их реализации на языке **Python** [2].

2.1. Обзор существующих решений

Обзор существующих решений направлен на систематизацию научного и инженерного опыта, накопленного в области стабилизации неустойчивых динамических систем и, в частности, механической системы «тележка–стержень». Это позволяет избежать дублирования уже известных подходов и чётко определить место данной работы среди предыдущих исследований.

2.1.1. Математическая модель «Тележка–Стержень»

Система «тележка–стержень» является классическим объектом исследования в теории автоматического управления (ТАУ) и представляет собой пример нелинейной, существенно неустойчивой системы [4]. Математическое описание обычно строится на законах Ньютона или уравнениях Лагранжа [11]. В ранних работах массу стержня часто аппроксимируют как точечную, однако для более точной симуляции применяется модель с учётом распределённой массы стержня, что усложняет расчёты, но повышает реалистичность моделирования.

2.1.2. Классические подходы к стабилизации

Исторически задача решалась с помощью методов, основанных на знании математической модели:

- 1. Линеаризация и LQR (Linear Quadratic Regulator).** Преимущество LQR — оптимальность управления для линеаризованной системы при минимизации квадратичного функционала [11]. Недостаток — сниженная эффективность при больших начальных отклонениях.
- 2. ПИД-регулирование (PID).** Простота и универсальность; сложность — подбор коэффициентов K_p, K_i, K_d для широкого диапазона состояний [3].
- 3. Управление на основе фазового пространства.** Используется переключение между различными линейными стратегиями в зависимости от состояния; метод трудоёмок и плохо масштабируется.

2.1.3. Современные подходы на основе обучения с подкреплением (RL)

Методы RL позволяют агенту обучаться оптимальному управлению, взаимодействуя со средой **без явного знания уравнений динамики** системы [12].

1. **Value-based** методы (DQN, Q-learning). Эффективны для дискретных действий, но при большой размерности требуют сложной аппроксимации функции ценности [6].
2. **Policy Gradient** методы. Непосредственно оптимизируют политику управления и предпочтительны для непрерывного пространства действий [13].
3. **Actor–Critic** (AC). Гибридный подход: актор выбирает действия, критик оценивает их, что уменьшает дисперсию градиента [8, 1].

2.1.4. Обоснование выбора методологии

Данное исследование фокусируется на сравнении трёх фундаментально разных классов оптимизации, применённых к одной и той же задаче: **наивного случайного поиска** (BRS), **локального поиска** (HC) и **градиентного метода** (Actor–Critic). Выбор **Actor–Critic с линейной аппроксимацией** обусловлен желанием проверить целесообразность применения градиентных методов: оправдывает ли их теоретическая мощь вычислительные затраты на вычисление градиентов по сравнению с простыми и быстрыми эвристиками (HC, BRS) в задаче с малым числом параметров состояния [9]. Использование линейных моделей позволяет также **снизить вычислительные требования** по сравнению с глубоким обучением. В литературе обычно сравнивают сложные современные алгоритмы (например, [1, 5]), однако в настоящей работе намеренно проводится прямое сравнение **простых** методов (BRS, HC) с **более сложными** градиентными подходами (AC) на одной и той же самописной физической модели.

2.2. Обзор используемых технологий и метрик

Данный подраздел посвящен обзору и обоснованию выбора тех технологических решений и алгоритмов, которые будут использованы для

программной реализации задачи стабилизации стержня. Цель — показать, почему выбранные алгоритмы (**BRS**, **HC**, **AC**) являются наилучшим набором для достижения целей исследования, а выбранные инструменты — оптимальными для реализации.

2.2.1. Сравнение и выбор алгоритмов оптимизации

В рамках работы проводится сравнительный анализ трёх методов, представляющих разные уровни сложности и подходы к оптимизации функций. Обзорная литература [12, 13] показывает, что эти методы формируют логическую иерархию, позволяющую провести методологическое сравнение:

1. **Basic Random Search (BRS):** Простейший безградиентный метод, основы которого заложены в работе [10].
2. **Hill Climbing (HC):** Эвристический метод, использующий локальный случайный поиск для ускорения сходимости BRS.
3. **Actor–Critic (AC):** Градиентный метод обучения с подкреплением, который теоретически должен превосходить безградиентные методы по скорости сходимости в задачах оптимизации [13, 8].

Для оценки алгоритмов использовались следующие критерии, релевантные целям работы: вычислительная сложность, скорость сходимости и риск застревания в локальном оптимуме.

Таблица 1: Сравнение алгоритмов оптимизации

Алгоритм	Класс	Вычисл. сложность	Риск локального оптимума
Basic Random Search	Безградиентный	Низкая	Низкий
Hill Climbing	Безградиентный	Низкая	Высокий
Actor–Critic	Градиентный (RL)	Средняя	Средний

Обоснование выбора: данный набор выбран потому, что он позволяет ответить на ключевой вопрос исследования: оправдывает ли рост вычислительной сложности (переход от **HC** к **AC**) пропорциональный

рост скорости сходимости в задаче стабилизации низкой размерности. Как показано в работе [9], простые методы часто недооцениваются. **BRS** и **HC** выступают в роли надёжных, но простых контрольных групп для оценки эффективности градиентного подхода.

2.2.2. Метрики оценки эффективности

Для всесторонней оценки качества управления и скорости обучения был выбран следующий набор метрик. Каждая метрика освещает определённый аспект работы алгоритма:

1. **Среднее вознаграждение** (*Average Reward*). Основной показатель **качества обучения**. Вычисляется как среднее арифметическое наград, полученных агентом за эпизод (или серию эпизодов).
2. **Кумулятивное вознаграждение** (*Cumulative Reward*). Сумма всех наград, полученных агентом в рамках одного эпизода. В задаче Cart–Pole эта метрика прямо пропорциональна **времени удержания стержня** до падения.
3. **Средняя длина эпизодов** (*Episode Length*). Показывает **устойчивость** системы. Максимальная длина эпизода (например, 1000 шагов) свидетельствует о найденном оптимальном и стабильном решении.
4. **RMSE по углу θ** (*Root Mean Square Error of θ*). Среднеквадратичная ошибка угла отклонения стержня от вертикали ($\theta = 0$). Характеризует **точность** стабилизации: низкий RMSE означает, что стержень удерживается почти неподвижно.
5. **Время удержания вертикали** (*Upright Steps*). Количество шагов, в течение которых угол отклонения стержня по модулю не превышал заданного порогового значения (например, $|\theta| < 0.087$ рад $\approx 5^\circ$). Метрика показывает способность алгоритма удерживать систему в **безопасной зоне равновесия**.

2.2.3. Выбор платформы реализации

Для реализации физической модели и алгоритмов необходимо выбрать подходящую программную среду и язык. Для работ в области машинного обучения стандартным выбором является язык **Python**, несмотря на то, что **C/C++** обеспечивает более высокую производительность для чистого численного интегрирования дифференциальных уравнений.

Таблица 2: Сравнение платформ для реализации

Платформа	Производительность	Удобство ML/RL	Скорость разработки
C/C++	Высокая	Низкое	Низкая
Python	Средняя	Высокое	Высокая

Обоснование выбора: для задачи стабилизации стержня, которая является системой низкой размерности, высокая производительность **C/C++** не является критичной. В то же время, основная цель работы — сравнение алгоритмов обучения. Использование языка **Python** значительно ускоряет разработку и обеспечивает лучшую читаемость кода.

2.2.4. Используемые библиотеки и инструменты

Для программной реализации среды и алгоритмов будет использован следующий стек технологий:

- **Язык программирования:** Python.
- **Численные расчёты и симуляция:** библиотека **NumPy** [2].
Используется для эффективного хранения и оперирования матрицами весовых коэффициентов агентов.
- **Визуализация и анализ данных:** библиотека **Matplotlib** [7].
Будет применяться для визуализации экспериментов, построения графиков сходимости, визуализации метрик и сравнения результатов экспериментов.

2.3. Выводы

Обзор существующих решений (раздел 2.1) показал, что, хотя задача стабилизации стержня решена классическими и современными методами, отсутствует прямое сравнение выбранных алгоритмов (**BRS**, **HC**, **AC**) на самописной физической модели. Обзор технологий (раздел 2.2) подтвердил, что язык **Python** и его библиотеки оптимальны для реализации, поскольку приоритет отдается удобству и скорости реализации сложных алгоритмов машинного обучения перед микрооптимизацией физической симуляции. Таким образом, теоретическая база для выполнения поставленных задач полностью сформирована.

3. Описание решения

В данной главе описывается последовательный процесс реализации системы автоматической стабилизации стержня. Решение задачи декомпозировано на три взаимосвязанных этапа, отражающих переход от теоретической модели к исполняемому программному коду.

В рамках **первого этапа** (подраздел 3.1) выполняется математическое моделирование. На основе законов классической механики выводятся нелинейные дифференциальные уравнения движения, которые ложатся в основу физической симуляции.

На **втором этапе** (подраздел 3.2) разрабатывается программная среда. На базе выведенных уравнений реализован класс `Model`, который выполняет численное интегрирование динамики системы методом Рунге-Кутты. Данный модуль выступает в роли «Среды» (`Environment`) в терминологии обучения с подкреплением.

Третий этап (подраздел 3.3) посвящен алгоритмической реализации агентов управления. Были реализованы и интегрированы в среду три метода различной природы:

- **Basic Random Search (BRS)** — в качестве базового метода глобальной стохастической оптимизации.
- **Hill Climbing (HC)** — как улучшенная эвристика локального поиска.
- **Actor-Critic (AC)** — как представитель градиентных методов RL, способный обучаться online.

Весь программный комплекс реализован на языке **Python** с активным использованием библиотеки **NumPy** для ускорения векторных вычислений.

3.1. Моделирование системы

Для описания динамики системы «стержень на тележке» воспользуемся вторым законом Ньютона. Система состоит из тележки массой

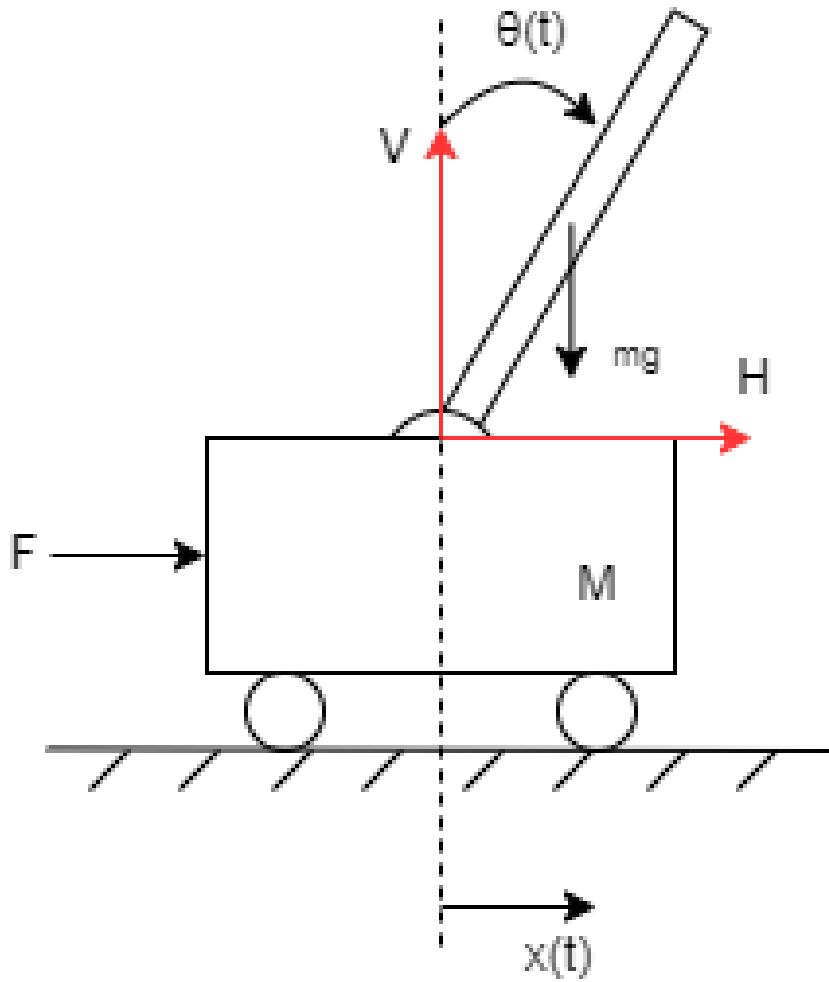


Рис. 1: Схема системы «Стержень на тележке» с действующими силами.

M , движущейся горизонтально, и стержня массой m и длиной l , закрепленного на тележке через шарнир.

Введем следующие обозначения:

- $x(t)$ — горизонтальная координата тележки;
- $\theta(t)$ — угол отклонения стержня от вертикали (по часовой стрелке);
- F — управляющая сила, приложенная к тележке;
- H, V — горизонтальная и вертикальная компоненты силы реакции в шарнире;

- $a = l/2$ — расстояние от оси шарнира до центра масс стержня;
- I — момент инерции стержня относительно точки крепления.

3.1.1. Кинематика системы

Координаты центра масс стержня (x_c, y_c) выражаются через обобщенные координаты системы x и θ :

$$x_c = x + a \sin \theta, \quad y_c = a \cos \theta. \quad (1)$$

Дважды продифференцировав эти выражения по времени, получим проекции ускорения центра масс стержня:

$$\begin{cases} \ddot{x}_c = \ddot{x} + a \cos \theta \ddot{\theta} - a \sin \theta \dot{\theta}^2, \\ \ddot{y}_c = -a \sin \theta \ddot{\theta} - a \cos \theta \dot{\theta}^2. \end{cases} \quad (2)$$

3.1.2. Уравнения динамики

Рассмотрим силы, действующие на тела системы.

1. Движение тележки. На тележку действуют управляющая сила F и горизонтальная реакция стержня H (по третьему закону Ньютона, направленная противоположно силе, действующей на стержень):

$$M\ddot{x} = F - H. \quad (3)$$

2. Движение стержня. Запишем уравнения движения центра масс стержня в проекциях на оси координат:

$$\begin{cases} m\ddot{x}_c = H, \\ m\ddot{y}_c = V - mg. \end{cases} \quad (4)$$

Подставим выражения для ускорений (2) в систему (4):

$$\begin{cases} m(\ddot{x} + a \cos \theta \ddot{\theta} - a \sin \theta \dot{\theta}^2) = H, \\ m(-a \sin \theta \ddot{\theta} - a \cos \theta \dot{\theta}^2) = V - mg. \end{cases} \quad (5)$$

Исключим силу реакции H . Сложим уравнение тележки (3) и первое уравнение стержня:

$$M\ddot{x} + m(\ddot{x} + a \cos \theta \ddot{\theta} - a \sin \theta \dot{\theta}^2) = F.$$

После группировки получаем первое уравнение движения системы:

$$(M + m)\ddot{x} + ma \cos \theta \ddot{\theta} = F + ma \sin \theta \dot{\theta}^2. \quad (6)$$

Для получения второго уравнения рассмотрим вращательное движение стержня. Запишем уравнение моментов относительно точки крепления (оси шарнира). Момент инерции стержня относительно конца равен $I = \frac{1}{3}ml^2 = \frac{4}{3}ma^2$. Уравнение вращательного движения:

$$ma \cos \theta \ddot{x} + I\ddot{\theta} = -mga \sin \theta. \quad (7)$$

3.1.3. Разрешение относительно старших производных

Объединим уравнения (6) и (7) в матричную форму относительно вектора угловых и линейных ускорений $[\ddot{x}, \ddot{\theta}]^T$:

$$\begin{pmatrix} M+m & ma \cos \theta \\ ma \cos \theta & I \end{pmatrix} \begin{pmatrix} \ddot{x} \\ \ddot{\theta} \end{pmatrix} = \begin{pmatrix} F + ma \sin \theta \dot{\theta}^2 \\ -mga \sin \theta \end{pmatrix}. \quad (8)$$

Определитель матрицы системы равен:

$$\Delta = (M + m)I - (ma \cos \theta)^2. \quad (9)$$

Решая систему методом Крамера или путем обратной матрицы, получаем явные выражения для ускорений, необходимые для численного моделирования:

$$\ddot{x} = \frac{I(F + ma \sin \theta \dot{\theta}^2) + m^2ga^2 \cos \theta \sin \theta}{\Delta}, \quad (10)$$

$$\ddot{\theta} = \frac{-(ma \cos \theta)(F + ma \sin \theta \dot{\theta}^2) - (M + m)mga \sin \theta}{\Delta}. \quad (11)$$

Полученная система уравнений (10)–(11) позволяет сформировать

вектор состояния системы $s = [x, \dot{x}, \theta, \dot{\theta}]^T$, который будет использоваться для реализации среды обучения с подкреплением.

3.1.4. Преобразование динамической системы в форму первого порядка

Исходная математическая модель, основанная на законах Ньютона, представляет собой систему обыкновенных дифференциальных уравнений (ОДУ) **второго порядка**, поскольку она выражается через вторые производные координат (ускорения \ddot{x} и $\ddot{\theta}$).

Большинство численных методов интегрирования (например, метод Рунге-Кутты) разработаны для работы с системами ОДУ **первого порядка**. Для приведения системы к требуемой форме мы используем **вектор состояния** s , который включает все обобщенные координаты и их первые производные (скорости):

$$s = \begin{pmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{pmatrix}. \quad (12)$$

Производная этого вектора \dot{s} представляет собой систему первого порядка:

$$\dot{s} = \frac{d}{dt}s = \begin{pmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{pmatrix}. \quad (13)$$

Таким образом, динамическая система принимает форму $\dot{s} = g(s, F)$, где функция $g(s, F)$ определяется:

$$g(s, F) = \begin{pmatrix} \dot{x} \\ \ddot{x}(s, F) \\ \dot{\theta} \\ \ddot{\theta}(s, F) \end{pmatrix}, \quad (14)$$

где $\ddot{x}(s, F)$ и $\ddot{\theta}(s, F)$ вычисляются по явным формулам (10) и (11).

3.1.5. Достаточность вектора состояния для динамики и RL

Для обучения с подкреплением агент оперирует **вектором наблюдения** (`observation_space`), который в данном случае совпадает с вектором состояния системы:

$$\mathbf{s}_{RL} = [x, \dot{x}, \theta, \dot{\theta}]^T. \quad (15)$$

1. Достаточность для динамики (Марковское свойство): Система описывается обыкновенными дифференциальными уравнениями (ОДУ) второго порядка. Для однозначного прогнозирования траектории системы в будущем необходимо знать текущее положение и скорость по каждой степени свободы. Вектор \mathbf{s}_{RL} включает все обобщенные координаты (x, θ) и их первые производные $(\dot{x}, \dot{\theta})$, что обеспечивает выполнение **Марковского свойства**: будущее состояние системы полностью определяется текущим состоянием s_t и управляющим воздействием F_t .

2. Пригодность для RL: Вектор \mathbf{s}_{RL} является **вектором наблюдения** (`Observation`), который предоставляет агенту всю информацию, необходимую для принятия оптимальных решений:

- x, \dot{x} : Необходимы для удержания тележки в пределах рабочей зоны.
- $\theta, \dot{\theta}$: Являются критическими параметрами. Агент использует θ для оценки отклонения от вертикали и $\dot{\theta}$ для прогнозирования будущей траектории стержня.

3.1.6. Численное интегрирование методом Рунге-Кутты 4-го порядка

Для симуляции движения системы (переход $s_t \rightarrow s_{t+\Delta t}$) используется **Метод Рунге-Кутты 4-го порядка (RK4)**. Этот метод обеспечивает

высокий порядок точности ($O(\Delta t^4)$ на шаге) и хорошую устойчивость при моделировании нелинейных динамических систем.

RK4 является одношаговым методом, использующим четыре оценки наклона (производной $g(s, F)$) для вычисления приращения вектора состояния на временном интервале Δt .

Пусть s_t — состояние в момент времени t , F — управляющая сила, и $g(s, F)$ — функция правой части системы \dot{s} . Шаг интегрирования Δt :

1. K_1 (**Наклон в начале интервала**):

$$\mathbf{K}_1 = \Delta t \cdot g(s_t, F)$$

2. K_2 (**Наклон в середине, с использованием K_1**):

$$\mathbf{K}_2 = \Delta t \cdot g(s_t + \mathbf{K}_1/2, F)$$

3. K_3 (**Уточненный наклон в середине, с использованием K_2**):

$$\mathbf{K}_3 = \Delta t \cdot g(s_t + \mathbf{K}_2/2, F)$$

4. K_4 (**Наклон в конце интервала, с использованием K_3**):

$$\mathbf{K}_4 = \Delta t \cdot g(s_t + \mathbf{K}_3, F)$$

5. **Итоговое состояние $s_{t+\Delta t}$:**

$$s_{t+\Delta t} = s_t + \frac{1}{6}(\mathbf{K}_1 + 2\mathbf{K}_2 + 2\mathbf{K}_3 + \mathbf{K}_4). \quad (16)$$

Этот численный метод реализуется в функции `step` симуляционной среды, обеспечивая эволюцию состояния системы под действием управляющей силы F и внутренних физических законов.

3.1.7. Функция вознаграждения

Функция вознаграждения r_t (Reward Function) определяет цель для RL-агента, поощряя желательное поведение (стабилизация) и наказывая нежелательное (падение или выход за пределы). На каждом времен-

ном шаге t вознаграждение рассчитывается по следующей формуле:

$$r = 1 - \left(C_1\theta^2 + C_2x^2 + C_3\dot{\theta}^2 + C_4\dot{x}^2 \right), \quad (17)$$

В данной реализации используются следующие весовые коэффициенты (константы): $C_1 = 0.1$, $C_2 = 0.1$, $C_3 = 0.01$, $C_4 = 0.01$.

Базовое вознаграждение r стремится к максимальному значению $r = 1$ при идеальном состоянии ($\theta = 0, x = 0, \dot{\theta} = 0, \dot{x} = 0$). Каждый компонент в скобках представляет собой "штраф", который вычитается из единицы:

1. Штраф за положение маятника и тележки ($0.1\theta^2 + 0.1x^2$):

- Наибольший вес (0.1) имеют штрафы, зависящие от **положения** — угла θ и координаты x . Это заставляет агента фокусироваться на удержании маятника как можно ближе к вертикали ($\theta = 0$) и тележки как можно ближе к центру ($x = 0$).

2. Штраф за скорости ($0.01\dot{\theta}^2 + 0.01\dot{x}^2$):

- Меньший вес (0.01) назначен штрафам, зависящим от **скоростей** ($\dot{\theta}, \dot{x}$). Это поощряет агента к стабилизации системы, но не делает это приоритетом перед удержанием положений.

Если система выходит за установленные физические границы, налагается крупный единовременный штраф, и эпизод завершается:

$$\text{Если } (|x| > x_{\text{fail}} \text{ или } |\theta| > \theta_{\text{fail}}) : \quad r_{\text{fail}} = r - 10. \quad (18)$$

- x_{fail} и θ_{fail} — это предельные значения для координаты тележки и угла маятника, при превышении которых симуляция считается неудачной.
- Штраф в -10 является значительным отрицательным вознаграждением, которое эффективно обучает агента избегать состояний, приводящих к сбою.

3.2. Программная реализация модели

Для моделирования динамики системы «Стержень на тележке» и создания среды для обучения с подкреплением (RL) разработан класс `Model` на языке Python.

3.2.1. Структура и параметры модели

Класс инициализируется с основными физическими и симуляционными параметрами:

- Массы: $M = 1.2$ кг (тележка), $m = 0.5$ кг (маятник).
- Длина маятника: $l = 1.0$ м.
- Шаг интегрирования: $dt = 0.01$ с.
- Максимальная управляющая сила: $F_{\max} = 10$ Н.
- Границы сбоя: $\theta_{\text{fail}} = \pi/2$ рад., $x_{\text{fail}} = 10.0$ м.

3.2.2. Функция динамики и интегрирование

Ключевыми компонентами являются методы, реализующие динамические уравнения и численный переход.

Функция динамики $g(s, F)$: Метод `g(self, state, action)` вычисляет производную вектора состояния \dot{s} , используя явные формулы для ускорений \ddot{x} и $\ddot{\theta}$ (уравнения (10) и (11)). Хотя аналитически определитель системы Δ строго положителен для любых $M, m > 0$, в программной реализации добавлено ограничение снизу $\Delta \geq 10^{-4}$. Это исключает возникновение ошибок деления на ноль или переполнения в случаях некорректной инициализации параметров масс или предельных погрешностей формата floating point.

Численное интегрирование: Метод `apply_action(self, state, action, dt)` использует **Метод Рунге-Кутты 4-го порядка (RK4)** для перехода от состояния s_t к s_{t+dt} .

3.2.3. Интерфейс симуляции (Метод `step`)

Метод `step(self, state, action)` является основным интерфейсом взаимодействия RL-агента со средой. Он принимает текущее состояние и выбранную агентом силу F (`action`), после чего возвращает новое состояние, вознаграждение и флаг завершения эпизода (`done`).

Ниже представлен код, реализующий динамическую модель:

Листинг 1: Реализация динамической модели «стержень на тележке»

```
1 import numpy as np
2
3 class Model():
4     def __init__(self, episode_length=1000, Fmax=10, dt=0.01):
5         self.l = 1.0
6         self.m = 0.5
7         self.M = 1.2
8         self.tetta_fail = np.pi/2
9         self.x_fail = 10.0
10        self.Fmax = Fmax
11        self.dt = dt
12        self.episode_length = episode_length
13
14    def step(self, state, action):
15        action = np.clip(action, -self.Fmax, self.Fmax)
16        new_state = self.apply_action(state, action, self.dt)
17        reward = self.reward(new_state)
18        done = abs(new_state[0]) > self.x_fail or abs(new_state[2]) > self.tetta_fail
19        return new_state, reward, done
20
21    def g(self, state, action):
22        x, x_der, tetta, tetta_der = state
23        g = 9.8
24        I = self.m * self.l**2 / 3
25        a = self.l / 2
26        # Вычисление определятеля Delta (знаменатель)
27        delta = I * (self.M + self.m) - (self.m * a * np.cos(tetta))**2
28        delta = max(delta, 1e-4) # Численная стабилизация
29        x_2der = (I * (action + self.m * a * tetta_der**2 * np.sin(tetta)) +
30                  self.m**2 * g * a**2 * np.cos(tetta) * np.sin(tetta)) / delta
31        tetta_2der = (-self.m * a * np.cos(tetta) * (action + self.m * a * tetta_der**2 *
32                           np.sin(tetta)) -
33                           (self.M + self.m) * self.m * g * a * np.sin(tetta)) / delta
34        return np.array([x_der, x_2der, tetta_der, tetta_2der])
35
36    def apply_action(self, state, action, dt):
37        # Метод Рунге-Куммы 4-го порядка (RK4)
38        k1 = self.g(state, action)
39        k2 = self.g(state + dt * 0.5 * k1, action)
40        k3 = self.g(state + dt * 0.5 * k2, action)
41        k4 = self.g(state + dt * k3, action)
42        new_state = state + dt * (1/6 * (k1 + 2*k2 + 2*k3 + k4))
```

```

42         return new_state
43
44     def reward(self, state):
45         x, x_der, tetta, tetta_der = state
46         # Базовое вознаграждение
47         r = 1 - (0.1 * tetta ** 2 + 0.1 * x ** 2 + 0.01 * tetta_der ** 2 + 0.01 * x_der
48             ** 2)
49         # Дополнительный штраф за сбой
50         if abs(x) > self.x_fail or abs(tetta) > self.tetta_fail:
51             r -= 10
52     return r

```

3.3. Реализация алгоритмов обучения

В данном разделе представлена программная реализация трех выбранных алгоритмов обучения, предназначенных для поиска оптимальной политики стабилизации стержня на тележке.

3.3.1. Реализация Basic Random Search (BRS)

Basic Random Search (BRS) — это простейший представитель безградиентных методов оптимизации политики. В контексте задачи CartPole, целью BRS является поиск оптимального вектора весов \mathbf{w} для **линейной политики**, которая связывает вектор состояния $\mathbf{s} = [x, \dot{x}, \theta, \dot{\theta}]^T$ с управляющей силой F :

$$F = \mathbf{w}^T \mathbf{s} = w_1 x + w_2 \dot{x} + w_3 \theta + w_4 \dot{\theta}$$

Принцип работы BRS основан на глобальном случайному поиске:

- **Генерация:** На каждой итерации (эпизоде) генерируется совершенно новый вектор весов \mathbf{w} (в данном случае из нормального распределения с математическим ожиданием -1 и стандартным отклонением 1).
- **Оценка:** Производится прогон модели (метод `rollout`) с использованием новой политики \mathbf{w} , и вычисляется суммарная награда r .

- **Обновление:** Если полученная награда r превышает текущую наилучшую награду (r_{best}), то \mathbf{w} сохраняется как новая лучшая политика \mathbf{w}_{best} .

Этот метод обеспечивает высокую вероятность выхода из локальных оптимумов, но его сходимость крайне низка, что делает его идеальным **базовым алгоритмом** для сравнения.

Класс `BRSagent` инкапсулирует логику поиска.

- Метод `learn` реализует основной цикл, где ключевая строка `self.w = np.random.normal(loc=-1, scale=1, size=4)` обеспечивает **полностью случайную генерацию** нового кандидата на каждой итерации.
- Метод `rollout` симулирует эпизод, используя линейную политику `action = np.clip(np.dot(w, state), ...)` и гарантирует, что управляющая сила F не превышает физического ограничения F_{max} .
- Метод `reset_state` инициализирует начальное состояние маятника \mathbf{s}_0 в случайной области.

Исходный код Basic Random Search (BRS)

Листинг 2: Реализация Basic Random Search

```
1 import numpy as np
2 from src.Model import Model
3
4 class BRSagent():
5     def __init__(self, model=Model(), n_episodes=1000):
6         # Инициализация случайными весами (линейная политика)
7         self.w = np.random.normal(loc=-1, scale=1, size=4)
8         self.n_episodes = n_episodes
9         self.model = model
10        self.best_w = self.w.copy()
11        self.best_reward = -np.inf
12        self.states = []
13        self.actions = []
14        self.rewards = []
15
16    def learn(self):
17        for episode_ind in range(self.n_episodes):
18            # Полнотью новый случайный вектор весов
19            self.w = np.random.normal(loc=-1, scale=1, size=4)
20            r = self.rollout(self.w)
21            # Сохранение лучшей политики
22            if r > self.best_reward:
23                self.best_reward = r
24                self.best_w = self.w.copy()
25
26    def rollout(self, w):
27        state = self.reset_state()
28        total_reward = 0
29        done = False
30        steps = 0
31        while not done and steps < self.model.episode_length:
32            # Линейная политика с ограничением F_max
33            action = np.clip(np.dot(w, state), -self.model.Fmax, self.model.Fmax)
34            new_state, reward, done = self.model.step(state, action)
35            self.states.append(state)
36            self.actions.append(action)
37            self.rewards.append(reward)
38            state = new_state
39            total_reward += reward
40            steps += 1
41        return total_reward
42
43    def reset_state(self):
```

```

44     # Инициализация состояния в случайном диапазоне
45     self.states = []
46     self.actions = []
47     self.rewards = []
48     state = np.array([
49         np.random.uniform(-1.0, 1.0),          # x
50         np.random.uniform(-0.1, 0.1),          # x_dot
51         np.random.uniform(-np.pi/6, np.pi/6), # tetta
52         np.random.uniform(-0.1, 0.1)          # tetta_dot
53     ], dtype=float)
54     return state

```

3.3.2. Реализация Hill Climbing (HC)

Hill Climbing (HC) (Восхождение на вершину) является итеративным алгоритмом локального поиска, который представляет собой усовершенствование BRS за счет целенаправленной эксплуатации окрестности уже найденного лучшего решения. В отличие от BRS, HC не генерирует полностью новую политику на каждой итерации, а вводит небольшое случайное **возмущение** (δ) к текущему лучшему вектору весов \mathbf{w}_{best} , формируя политику-кандидат \mathbf{w}' :

$$\mathbf{w}' = \mathbf{w}_{best} + \delta, \quad \text{где } \delta \sim \mathcal{N}(\mathbf{0}, \sigma_{dist}^2 \mathbf{I})$$

Здесь δ — вектор случайного шума, распределенного по Гауссу со стандартным отклонением σ_{dist} (параметр, контролирующий радиус локального поиска).

Принцип работы HC:

- **Генерация кандидата:** Политика-кандидат \mathbf{w}' генерируется путем добавления гауссовского шума к текущему лучшему вектору весов \mathbf{w}_{best} .
- **Оценка:** Производится прогон модели (`rollout`) с использованием \mathbf{w}' , и вычисляется суммарная награда r .
- **Обновление:** Если полученная награда r превышает текущую наилучшую награду (r_{best}), то \mathbf{w}' становится новой \mathbf{w}_{best} . Если улучшения нет, \mathbf{w}_{best} остается неизменным.

Данная реализация включает механизм **затухания** σ_{decay} , который позволяет уменьшать радиус поиска с течением времени. Это способствует переходу от широкого исследования пространства в начале обучения к точной настройке (эксплуатации) найденного оптимума на более поздних стадиях. НС, как правило, сходится быстрее, чем BRS, но более подвержен риску застревания в локальных оптимумах.

Обоснование коэффициента затухания σ_{decay}

Для эффективного обучения НС критически важно правильно настроить **скорость затухания** радиуса поиска (σ_{dist}). Чрезмерно быстрое затухание приводит к застреванию в локальном оптимуме, а слишком медленное — к неэффективному исследованию.

Мы стремимся, чтобы радиус поиска σ уменьшился до определенной доли ($\sigma_{final}/\sigma_{init}$) от начального значения σ_{init} за заданное число итераций N_{steps} .

Уменьшение σ происходит по геометрической прогрессии:

$$\sigma_N = \sigma_{init} \cdot (\sigma_{decay})^N$$

Была поставлена цель: **уменьшить радиус поиска до 10% от начального значения за $N = 1000$ итераций**. Это обеспечит быстрый переход от глобального исследования к точной настройке оптимума в начале обучения.

Тогда желаемое отношение $\sigma_N/\sigma_{init} = 0.1$. Для нахождения σ_{decay} решаем уравнение относительно желаемого отношения 0.1 при $N = 1000$:

$$0.1 = (\sigma_{decay})^{1000} \implies \ln(\sigma_{decay}) = \frac{\ln(0.1)}{1000} \approx -0.0023026$$

Вычисляя экспоненту, получаем: $\sigma_{decay} \approx 0.997699\dots$ Округляя полученное значение до трех знаков после запятой (для удобства программной реализации), мы выбираем коэффициент затухания:

$$\sigma_{decay} = 0.998 \tag{19}$$

Этот коэффициент обеспечивает контролируемое сжатие радиуса по-

иска, достигая 10% от начального значения ровно через 1000 итераций. Класс `HCagent` включает два новых ключевых параметра: `sigma` и `sigma_decay`.

- В методе `learn` генерируется возмущение: `delta = np.random.normal(0, self.sigma, size=4)`, а затем вычисляется кандидат `new_w = self.w + delta`.
- Обновление весов происходит только при улучшении, что соответствует классическому алгоритму HC.
- В конце каждой итерации `sigma` уменьшается: `self.sigma = max(self.sigma * self.sigma_decay, 0.01)`.

Исходный код Hill Climbing (HC)

Листинг 3: Реализация Hill Climbing (HC) с затуханием шума

```
1 import numpy as np
2 from src.Model import Model
3
4 class HCagent():
5     def __init__(self, model=Model(), sigma=1, sigma_decay=0.998, n_episodes=1000):
6         self.w = np.random.uniform(-0.4, 0.4, size=4)
7         self.best_w = self.w.copy()
8         self.sigma = sigma                      # Параметр шума (радиус поиска)
9         self.sigma_decay = sigma_decay          # Коэффициент затухания шума
10        self.n_episodes = n_episodes
11        self.model = model
12        self.best_reward = -np.inf
13        self.states = []
14        self.actions = []
15        self.rewards = []
16
17    def learn(self):
18        for episode_ind in range(self.n_episodes):
19            # Генерация гауссского шума (дельта)
20            delta = np.random.normal(0, self.sigma, size=4)
21            # Новый кандидат w'
22            new_w = self.w + delta
23            reward = self.rollout(new_w)
24            # Обновление w только при улучшении (восхождение)
25            if reward > self.best_reward:
26                self.w = new_w
27                self.best_reward = reward
28                self.best_w = new_w.copy()
29            # Затухание радиуса поиска
30            self.sigma = max(self.sigma * self.sigma_decay, 0.01)
31
32    def rollout(self, w):
33        state = self.reset_state()
34        total_reward = 0
35        done = False
36        steps = 0
37        while not done and steps < self.model.episode_length:
38            # Линейная политика с ограничением F_max
39            action = np.clip(np.dot(w, state), -self.model.Fmax, self.model.Fmax)
40            new_state, reward, done = self.model.step(state, action)
41            self.states.append(state)
42            self.actions.append(action)
```

```

43         self.rewards.append(reward)
44         state = new_state
45         total_reward += reward
46         steps += 1
47
48     return total_reward
49
50
51     def reset_state(self):
52         # Инициализация состояния в случайному диапазоне
53         self.states = []
54         self.actions = []
55         self.rewards = []
56         state = np.array([
57             np.random.uniform(-1.0, 1.0),           # x
58             np.random.uniform(-0.1, 0.1),           # x_dot
59             np.random.uniform(-np.pi/6, np.pi/6),   # tetta
60             np.random.uniform(-0.1, 0.1)            # tetta_dot
61         ], dtype=float)
62
63     return state

```

3.3.3. Реализация архитектуры Actor-Critic (AC)

В отличие от BRS и HC, метод **Actor-Critic** относится к классу алгоритмов обучения с подкреплением, использующих градиентную оптимизацию. Это гибридная архитектура, сочетающая в себе преимущества методов на основе ценности (Value-based) и методов на основе политики (Policy-based).

Архитектура состоит из двух аппроксиматоров:

1. **Actor (Актер):** Определяет стохастическую политику $\pi_\theta(a|s)$, выбирая действие a в состоянии s . Его цель — максимизировать ожидаемую награду $J(\theta)$.
2. **Critic (Критик):** Оценивает функцию ценности состояния $V_w(s)$, предсказывая ожидаемую сумму будущих наград. Его цель — минимизировать ошибку предсказания (TD-ошибку).

Математическая модель

В данной реализации используется параметризация политики Гауссовским распределением: $a \sim \mathcal{N}(\mu(s), \sigma(s))$. Параметры распределения (μ

и σ) и функция ценности $V(s)$ аппроксимируются линейными моделями от вектора состояния \mathbf{s} :

$$\mu(s) = \theta_\mu^T \mathbf{s}, \quad \sigma(s) = \exp(\theta_\sigma^T \mathbf{s}), \quad V(s) = \mathbf{w}^T \mathbf{s}. \quad (20)$$

Использование экспоненты для $\sigma(s)$ необходимо для гарантии положительности стандартного отклонения.

Теорема о градиенте политики и вывод правил обновления
Согласно Теореме о градиенте политики (Policy Gradient Theorem), градиент целевой функции $J(\theta)$ по параметрам политики θ определяется как:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi [\nabla_\theta \ln \pi(a|s) \cdot Q(s, a)]. \quad (21)$$

В архитектуре Actor-Critic вместо функции $Q(s, a)$ используется TD-ошибка δ_t , которая служит несмещенной оценкой преимущества действия:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t). \quad (22)$$

Для реализации градиентного подъема необходимо вычислить градиент $\nabla_\theta \ln \pi(a|s)$. Плотность вероятности нормального распределения задается формулой:

$$\pi(a|s) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(a-\mu)^2}{2\sigma^2}\right).$$

Логарифм плотности вероятности:

$$\ln \pi(a|s) = -\frac{1}{2} \ln(2\pi) - \ln \sigma - \frac{(a-\mu)^2}{2\sigma^2}.$$

Найдем частные производные по параметрам векторов θ_μ и θ_σ :

1. Для среднего значения μ : Используя цепное правило $\nabla_{\theta_\mu} = \frac{\partial \ln \pi}{\partial \mu} \frac{\partial \mu}{\partial \theta_\mu}$ и учитывая, что $\frac{\partial \mu}{\partial \theta_\mu} = \mathbf{s}$:

$$\nabla_{\theta_\mu} \ln \pi(a|s) = \frac{a-\mu}{\sigma^2} \mathbf{s}. \quad (23)$$

Это выражение реализовано в методе `acceptance_mu`.

2. Для стандартного отклонения σ : Учитывая параметризацию $\sigma = \exp(\theta_\sigma^T \mathbf{s})$, производная внутренней функции равна $\frac{\partial \sigma}{\partial \theta_\sigma} = \sigma \mathbf{s}$. Производная логарифма правдоподобия по σ :

$$\frac{\partial \ln \pi}{\partial \sigma} = -\frac{1}{\sigma} + \frac{(a - \mu)^2}{\sigma^3}.$$

Итоговый градиент:

$$\nabla_{\theta_\sigma} \ln \pi(a|s) = \left(\frac{(a - \mu)^2}{\sigma^3} - \frac{1}{\sigma} \right) \sigma \mathbf{s} = \left(\frac{(a - \mu)^2}{\sigma^2} - 1 \right) \mathbf{s}. \quad (24)$$

Это выражение реализовано в методе `acceptance_sigma`.

Класс `ACagent` реализует описанную математику, содержа параметры обучения ($\alpha_{actor}, \alpha_{critic}, \gamma$) и веса: `w`, `tetta_mu`, `tetta_sigma`.

Ключевые методы реализации:

- **Расчет TD-ошибки:** В методе `learn` вычисляется `TD_error`, которая выступает «учителем» для Актера.
- **Обновление Критика:** Веса `w` обновляются методом стохастического градиентного спуска: $\mathbf{w} \leftarrow \mathbf{w} + \alpha_{critic} \delta_t \mathbf{s}$.
- **Обновление Актера:** Используются выведенные выше градиенты. Если действие привело к положительной ошибке δ_t , вероятность такого действия увеличивается.
- **Клиппинг (Clipping):** Для обеспечения стабильности обучения и предотвращения «взрыва градиентов», параметры σ и сами весовые коэффициенты принудительно ограничиваются в заданных диапазонах (`np.clip`).

Исходный код Actor-Critic

Листинг 4: Реализация алгоритма Actor-Critic

```
1 import numpy as np
2 from src.Model import Model
3
4 class ACagent():
5     def __init__(self, lr_actor=5e-4, lr_critic=1e-3, gamma=0.95, model=Model(),
6                  n_episodes=1000):
7         self.lr_actor = lr_actor      # Скорость обучения Актера
8         self.lr_critic = lr_critic    # Скорость обучения Критика
9         self.gamma = gamma           # Коэффициент дисконтирования
10        self.model = model
11        self.n_episodes = n_episodes
12        # Инициализация весов
13        self.w = np.random.uniform(-1, 1, size=4)          # Веса Критика (Value func)
14        self.tetta_mu = np.zeros(4)                         # Веса Актера (Мат. ожидание)
15        self.tetta_sigma = np.ones(4) * np.log(0.5)          # Веса Актера (Дисперсия)
16        # Параметры ограничения (clipping) для стабильности
17        self.sigma_clip_min = 0.05
18        self.sigma_clip_max = 2.0
19        self.tetta_sigma_clip = 5.0
20        self.tetta_mu_clip = 10
21        self.states = []
22        self.actions = []
23        self.rewards = []
24
25    def learn(self):
26        for episode_ind in range(self.n_episodes):
27            self.rollout()
28            # Пошаговое обновление весов (Online learning)
29            for i in range(len(self.actions)):
30                action, state, reward = self.actions[i], self.states[i], self.rewards[i]
31                # Оценка ценности следующего состояния  $V(s')$ 
32                next_value = self.value_func(self.states[i+1]) if i+1 < len(self.states)
33                else 0
34                # Вычисление ошибки временной разности:  $\delta = r + \gamma V(s') - V(s)$ 
35                TD_error = reward + self.gamma * next_value - self.value_func(state)
36                # Обновление весов Актера (Policy Update)
37                # acceptance_... реализуют градиент логарифма правдоподобия
38                self.tetta_mu += self.lr_actor * TD_error * self.acceptance_mu(action,
→ state)
                self.tetta_sigma += self.lr_actor * TD_error *
→ self.acceptance_sigma(action, state)
                # Ограничение весов для предотвращения расходимости
```

```

39         self.tetta_sigma = np.clip(self.tetta_sigma, -self.tetta_sigma_clip,
40             ↵ self.tetta_sigma_clip)
41         self.tetta_mu = np.clip(self.tetta_mu, -self.tetta_mu_clip,
42             ↵ self.tetta_mu_clip)
43         # Обновление весов Критика (Value Update)
44         # Градиент  $V(s)$  по  $w$  равен самому вектору состояния  $state$  (так как  $V$ 
45             ↵ линейна)
46         self.w += self.lr_critic * TD_error * state
47
48
49     def rollout(self):
50         state = self.reset_state()
51         total_reward = 0
52         done = False
53         steps = 0
54         while not done and steps < self.model.episode_length:
55             # Выбор действия стochастической политикой
56             mu_val = self.mu(state)
57             sigma_val = self.sigma(state)
58             action = self.policy(mu_val, sigma_val)
59             new_state, reward, done = self.model.step(state, action)
60             self.states.append(state)
61             self.actions.append(action)
62             self.rewards.append(reward)
63             state = new_state
64             total_reward += reward
65             steps += 1
66
67     return total_reward
68
69
70     def value_func(self, state):
71         # Линейная аппроксимация  $V(s) = w^T * s$ 
72         return np.dot(state, self.w)
73
74
75     def policy(self, mu, sigma):
76         # Сэмплирование действия из нормального распределения
77         a = np.random.normal(mu, sigma)
78         return np.clip(a, -self.model.Fmax, self.model.Fmax)
79
80
81     def sigma(self, s):
82         # Вычисление стандартного отклонения:  $\sigma = \exp(\theta_\sigma \cdot T * s)$ 
83         z = np.dot(s, self.tetta_sigma)
84         z = np.clip(z, -10, 10) # Защита от переполнения экспоненты
85         sigma = np.clip(np.exp(z), self.sigma_clip_min, self.sigma_clip_max)
86         return sigma
87
88
89     def mu(self, s):
90         # Вычисление мат. ожидания:  $\mu = \theta_\mu \cdot T * s$ 
91         mu = np.dot(s, self.tetta_mu)

```

```

83     return mu
84
85     def acceptance_mu(self, action, state):
86         # Градиент log-probability по параметрам mu
87         # d(ln pi)/d(mu) * d(mu)/d(theta_mu)
88         acceptance = state * (action - self.mu(state)) / (self.sigma(state) ** 2 + 1e-8)
89         return acceptance
90
91     def acceptance_sigma(self, action, state):
92         # Градиент log-probability по параметрам sigma
93         # Учитываем, что sigma параметризована через экспоненту
94         acceptance = state * ((action - self.mu(state)) ** 2 / (self.sigma(state) ** 2 +
95             ↪ 1e-8) - 1)
96         return acceptance
97
98     def reset_state(self):
99         self.states = []
100        self.actions = []
101        self.rewards = []
102        state = np.array([
103            np.random.uniform(-1.0, 1.0),           # x
104            np.random.uniform(-0.1, 0.1),           # x_dot
105            np.random.uniform(-np.pi/6, np.pi/6), # tetta
106            np.random.uniform(-0.1, 0.1)           # tetta_dot
107        ], dtype=float)
108        return state

```

4. Эксперимент

Данный раздел посвящен описанию и анализу результатов численного эксперимента.

Эксперимент включает два основных этапа:

1. **Настройка гиперпараметров:** Определение оптимальных конфигураций для каждого из алгоритмов с целью обеспечения максимальной производительности и стабильности.
2. **Сравнительный анализ:** Итоговое сравнение отобранных алгоритмов по ключевым метрикам и оценка целесообразности использования градиентных подходов для задач данной размерности.

4.1. Условия эксперимента и подбор гиперпараметров

Перед проведением сравнительного анализа алгоритмов был выполнен этап настройки гиперпараметров (Hyperparameter Tuning). Для каждого алгоритма была проведена серия запусков с различными конфигурациями для выявления параметров, обеспечивающих наилучшую сходимость и стабильность. Для усреднения результатов стохастических процессов каждый эксперимент запускался несколько раз (5 для AC, 10 для BRS и HC), а в качестве метрики качества использовалось среднее значение максимального полного вознаграждения, полученного агентом (*Best Reward*).

Ниже представлены результаты подбора для каждого из исследуемых методов.

4.1.1. Настройка Basic Random Search (BRS)

Для BRS критическим фактором является способ инициализации весовых коэффициентов, так как алгоритм не имеет механизма направленного улучшения, а полагается на удачную генерацию. Исследовались

два типа распределений: равномерное (в диапазоне $[-w, w]$) и нормальное ($\mathcal{N}(\mu, \sigma^2)$).

Результаты экспериментов по подбору диапазона равномерного распределения представлены в таблице 3.

Таблица 3: BRS: Влияние диапазона инициализации (Равномерное распределение)

Диапазон весов (w_{range})	Средняя награда
0.01	998.91
0.1	998.78
0.2	999.14
0.4	999.30
1.0	999.01

Также было исследовано влияние параметров нормального распределения (Таблица 4).

Таблица 4: BRS: Влияние параметров нормального распределения

Мат. ожидание (μ)	Стд. отклонение (σ)	Средняя награда
-1	0.01	999.02
-1	0.1	999.24
-1	1.0	999.45
0	0.1	999.23
0	1.0	999.20
1	1.0	999.42

Выбор: Для финальных экспериментов было выбрано нормальное распределение с $\mu = -1$, $\sigma = 1$. Количество эпизодов было зафиксировано на уровне 1000, так как дальнейшее увеличение до 2000 дает минимальный прирост (с 999.73 до 999.83).

4.1.2. Настройка Hill Climbing (HC)

Для алгоритма HC, помимо инициализации, критически важны параметры шума: амплитуда возмущения (σ_{noise}) и коэффициент её затухания.

В отличие от BRS, алгоритм НС оказался чувствителен к начальной точке: нормальное распределение с $\mu = 1$ показало крайне низкие результаты, в то время как равномерное распределение обеспечило стабильную сходимость.

Таблица 5: НС: Влияние диапазона инициализации (Равномерное распределение)

Диапазон весов (w_{range})	Средняя награда
0.01	999.85
0.1	999.87
0.2	999.82
0.4	999.91
1.0	999.88

Таблица 6: НС: Влияние параметров нормального распределения

Мат. ожидание (μ)	Стд. отклонение (σ)	Средняя награда
-1	0.01	999.88
-1	0.1	999.88
-1	1.0	999.91
0	0.01	953.04
0	0.1	999.65
0	1.0	898.58
1	0.01	611.45
1	0.1	807.46
1	1.0	581.22

Параметры шума (Таблица 7) определяют способность алгоритма выходить из локальных минимумов.

Таблица 7: НС: Влияние параметров шума (σ_{noise})

Дисперсия шума	Средняя награда
0.01	936.27
0.1	869.73
0.5	967.40
1.0	999.87

Выбор: Для НС выбрана инициализация Uniform[-0.4, 0.4] и начальная дисперсия шума $\sigma = 1.0$.

4.1.3. Настройка Actor-Critic (AC)

Для градиентного метода Actor-Critic ключевыми параметрами являются скорости обучения (Learning Rate) для сетей Актера и Критика, а также начальная дисперсия действий. Был проведен поиск по сетке (Grid Search) для пары $(\alpha_{actor}, \alpha_{critic})$.

Таблица 8: Actor-Critic: Подбор скоростей обучения (LR)

LR Actor	LR Critic	Средняя награда
$1 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	997.22
$1 \cdot 10^{-4}$	$3 \cdot 10^{-3}$	997.81
$3 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	999.04
$3 \cdot 10^{-4}$	$3 \cdot 10^{-3}$	998.45
$5 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	998.80
$5 \cdot 10^{-4}$	$1 \cdot 10^{-3}$	999.52

Также было установлено, что более высокая начальная энтропия (начальная σ_{init}) способствует лучшему исследованию среды.

Таблица 9: Actor-Critic: Влияние начальной дисперсии и дисконтирования

Нач. Sigma (σ_{init})	Награда	Gamma (γ)	Награда
0.2	998.94	0.95	999.68
0.3	999.50	0.98	998.73
0.5	999.61	0.99	999.64

Выбор: Для финальных экспериментов Actor-Critic используются: $\alpha_{actor} = 5 \cdot 10^{-4}$, $\alpha_{critic} = 1 \cdot 10^{-3}$, $\sigma_{init} = 0.5$, $\gamma = 0.95$.

4.1.4. Итоговая конфигурация

На основе проведенных экспериментов сформирована итоговая конфигурация гиперпараметров для сравнительного анализа (Таблица 10).

Таблица 10: Итоговые параметры алгоритмов для сравнительного эксперимента

Алгоритм	Параметры
BRS	$w \sim N(-1, 1)$
HC	$w \sim U[-0.4, 0.4]$, $\sigma_{noise} = 1.0$
Actor-Critic	$\alpha_{act} = 5e^{-4}$, $\alpha_{crit} = 1e^{-3}$, $\gamma = 0.95$, $\sigma_{init} = 0.5$

4.2. Сравнительный анализ метрик

Для проведения объективного сравнительного анализа эффективности алгоритмов Basic Random Search (BRS), Hill Climbing (HC) и Actor-Critic (AC) была выбрана система из пяти ключевых метрик:

1. **Среднее вознаграждение (Average Reward):** Среднее вознаграждение по всем шагам, отражающее общее качество управления.
2. **Кумулятивное вознаграждение (Cumulative Reward):** Суммарная награда, достигнутая агентом к концу эпизода (максимум 1000).
3. **Средняя длина эпизодов (Episode Length):** Среднее количество шагов, которое агент смог удержать стержень до падения (максимум 1000).
4. **RMSE θ (Root Mean Square Error of θ):** Среднеквадратичное отклонение угла θ от вертикали (0 рад), показывающее точность стабилизации.
5. **Время удержания вертикали (Upright Steps):** Среднее количество шагов, в течение которых угол стержня $|\theta|$ не превышал порогового значения 0.087 рад (5°).

В анализ был дополнительно включен агент **BestBRS**. Этот агент не является обучаемым, а представляет собой агента с **фиксированными весами**, которые были выбраны как лучшая политика из 10

независимых прогонов алгоритма BRS. BestBRS служит эталоном для оценки максимальной производительности, достижимой с помощью градиентно-свободных методов.

Сводные результаты метрик Сводные результаты, полученные для четырех агентов (HC, BRS, BestBRS, AC) после проведения экспериментов, представлены в Таблице 11.

Таблица 11: Сводные результаты алгоритмов по пяти основным метрикам

Метрика	HC	BRS	BestBRS	AC
Ср. Кум. Награда (Max 1000)	911.48	471.67	992.87	975.06
Ср. Вознаграждение/Шаг	0.911	0.472	0.993	0.975
Ср. Длина Эпизода (шагов)	969.0	571.9	1000.0	994.4
Ср. RMSE θ (рад)	0.1435	0.3470	0.1078	0.1319
Ср. Шагов в вертикали	776023	631930	735413	693627

1. Производительность (Кумулятивная Награда и Длина Эпизода): Агент **BestBRS** является безусловным лидером, достигая почти идеального среднего кумулятивного вознаграждения (992.87) и максимальной средней длины эпизода (1000.0 шагов). Это подтверждает, что для задачи CartPole существует высококачественный фиксированный набор весов, и что обучение может быть сведено к поиску этого набора.

Агент **Actor-Critic (AC)** показал второй лучший результат (975.06), демонстрируя высокую эффективность градиентного обучения, близкую к эталону. Агент **Hill Climbing (HC)** показал средний результат (911.48), уступая лидерам. Агент **BRS** оказался наименее эффективным (471.67), что ожидаемо, поскольку он не использует накопленный опыт.

2. Точность и Стабильность (RMSE θ): Наименьший средний показатель ошибки стабилизации (RMSE θ) принадлежит **BestBRS** (0.1078 рад), что логично, так как он представляет лучшую найденную политику. **Actor-Critic (AC)** показал третье место по средней ошибке (0.1319 рад), но его минимальный RMSE θ (0.0841 рад) оказался наименьшим среди всех агентов, что говорит о его потенциале к достижению очень

точной стабилизации в лучших прогонах. **BRS** показал худшую точность (0.3470 рад).

3. Время удержания вертикали (Upright Steps): По этой метрике, количество шагов, когда угол $|\theta| \leq 0.087$ рад, лидирует агент **HC** (≈ 776 тыс. шагов). Это указывает на то, что, хотя его общая награда ниже, его политика склонна удерживать стержень в очень узком диапазоне в течение длительного времени, компенсируя это снижением награды, когда стержень уходит от вертикали.

4.3. Результаты

Для более глубокого понимания поведения алгоритмов, помимо сводных табличных данных, был проведен анализ динамики обучения и распределения показателей качества. Ниже представлены графики, полученные в ходе экспериментов, с соответствующими комментариями.

4.3.1. Динамика вознаграждения

На рисунке 2 представлена динамика усредненного вознаграждения, получаемого агентами на каждом шаге обучения.

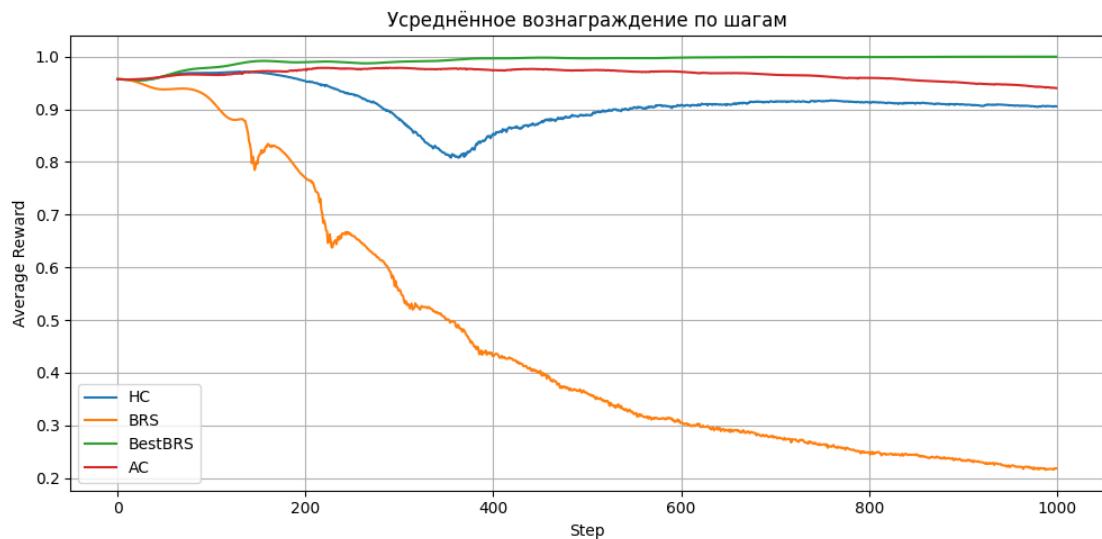


Рис. 2: Усреднённое вознаграждение по шагам для всех агентов

Из графика видно следующее:

- **BestBRS** и **Actor-Critic (AC)** демонстрируют стабильно высокие показатели, удерживая планку среднего вознаграждения близко к максимальной на протяжении всего времени.
- **BRS** показывает стабильно низкий результат, что подтверждает неэффективность простого случайного поиска без механизма отбора.
- **Hill Climbing (HC)** демонстрирует характерное поведение: после начального роста наблюдается постепенное *снижение* среднего вознаграждения.

Анализ деградации Hill Climbing: Снижение эффективности НС объясняется чувствительностью алгоритма к стохастической природе начального состояния среды. В задаче CartPole каждый эпизод начинается со случайного угла отклонения шеста $\theta \in [-0.05, 0.05]$. Алгоритм работает следующим образом:

1. Агент находит набор весов w , который показывает отличный результат при малых начальных отклонениях.
2. В следующем эпизоде среда инициализируется с *большим* углом отклонения. Текущие веса w неправляются с управлением, агент быстро роняет шест и получает низкую награду.
3. Алгоритм НС воспринимает это как сигнал о том, что веса w "плохие", и пытается перейти в соседнюю точку пространства весов. Однако из-за уменьшения параметра шума (σ) со временем (для обеспечения сходимости), агент теряет способность совершать большие скачки и не может найти глобально устойчивые веса.
4. В результате агент "сползает" в локальный оптимум или менее эффективную область, что на графике выглядит как постепенное уменьшение среднего вознаграждения.

Рисунок 3 подтверждает эти выводы, отображая кумулятивное вознаграждение. BestBRS и AC достигают максимума быстрее всех, в то время как HC отстает из-за описанной выше нестабильности.

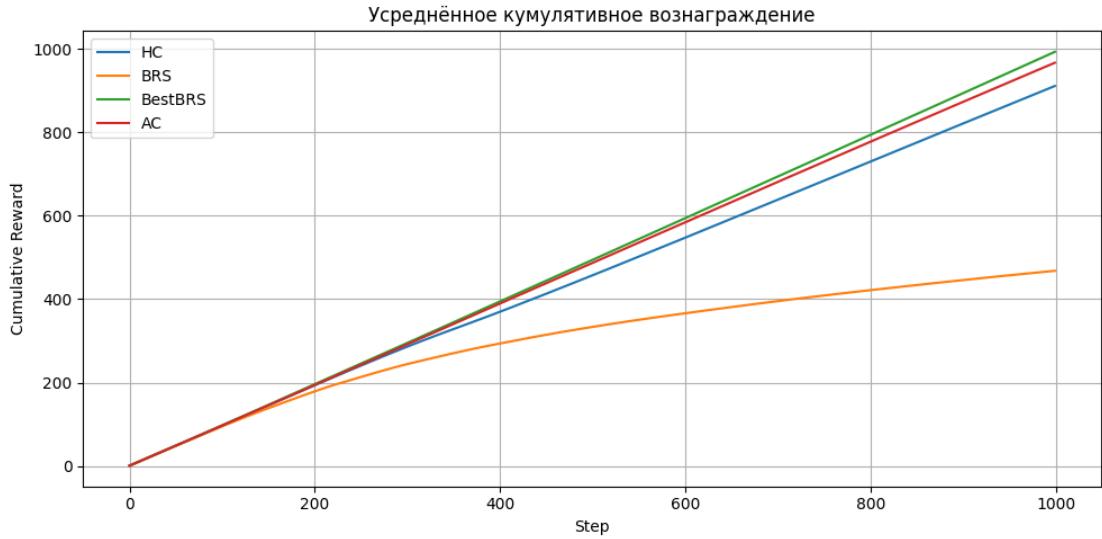


Рис. 3: Усреднённое кумулятивное вознаграждение

4.3.2. Стабильность длины эпизодов

На рисунке 4 показано распределение длин эпизодов. Красные точки обозначают среднее значение, синие — разброс по отдельным запускам.

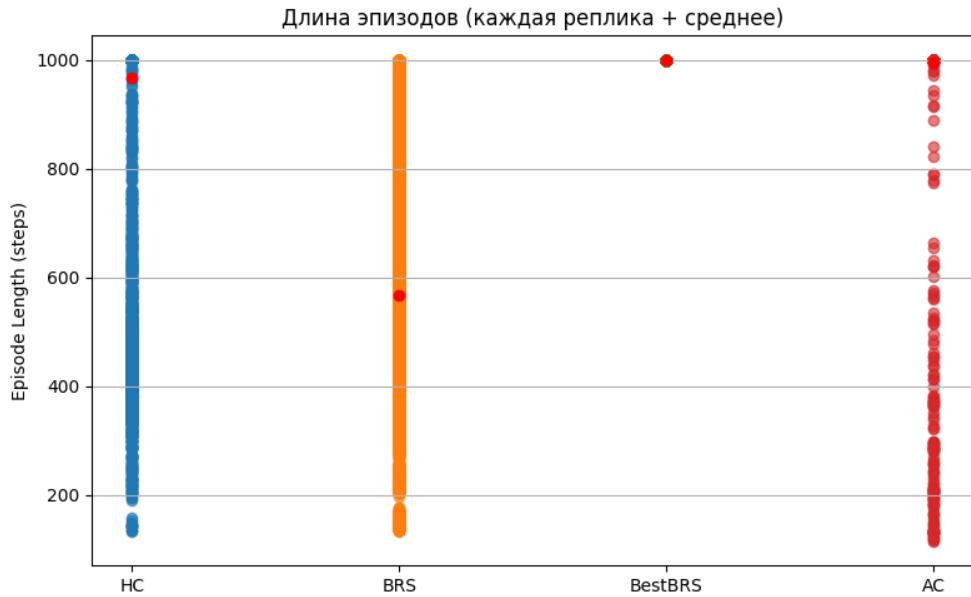


Рис. 4: Распределение длины эпизодов

- **BestBRS** демонстрирует идеальную стабильность: все эпизоды (плотное скопление точек) достигают лимита в 1000 шагов.
- **AC** имеет высокую среднюю длину, но наблюдаются отдельные выбросы (точки ниже 1000), что говорит о редких сбоях в процессе обучения.
- **HC** имеет значительный разброс, варьируясь от 73 до 1000 шагов, что подтверждает его зависимость от начальных условий.

4.3.3. Качество стабилизации (RMSE и Upright Steps)

Для детального анализа качества управления были построены графики среднеквадратичной ошибки угла (RMSE θ) и времени удержания в вертикали для каждого агента.

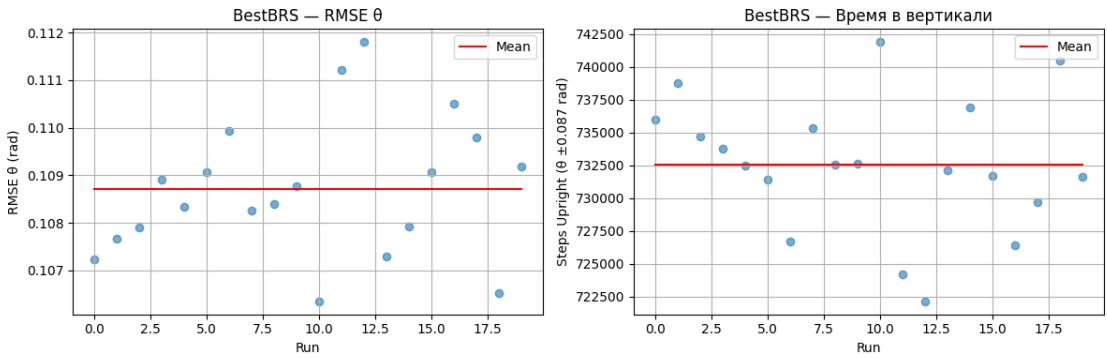


Рис. 5: Показатели BestBRS

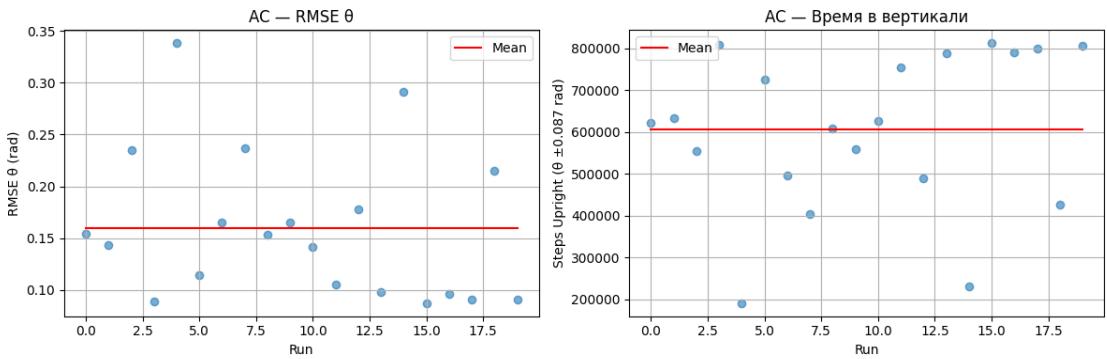


Рис. 6: Показатели Actor-Critic

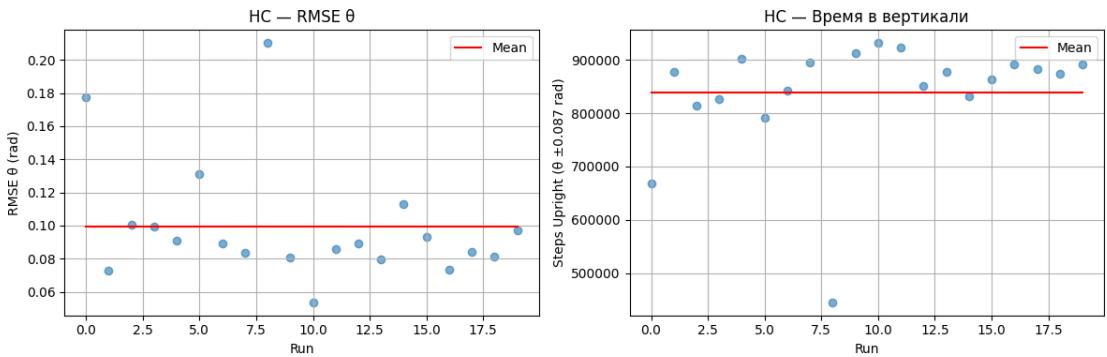


Рис. 7: Показатели HC

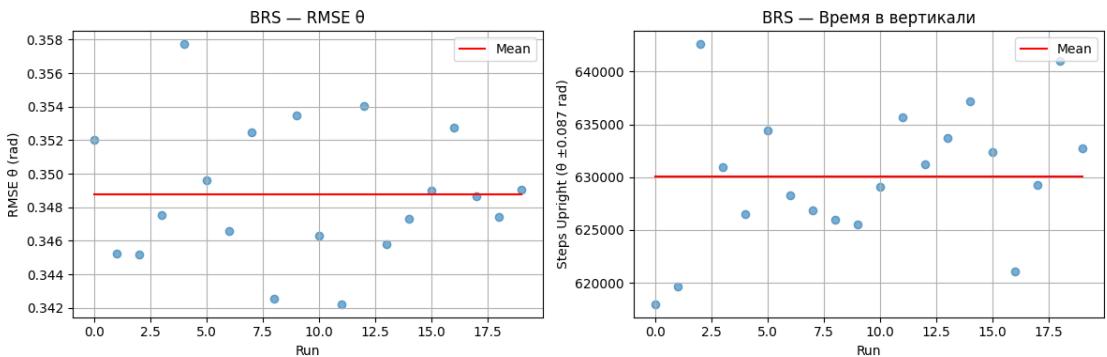


Рис. 8: Показатели BRS

- **BestBRS (Рис. 5):** Показывает минимальный разброс значений RMSE и стабильно высокое время удержания. Это эталонное поведение.
- **AC (Рис. 6):** Также демонстрирует хорошие результаты, но с чуть большим разбросом значений RMSE, чем у BestBRS.
- **HC (Рис. 7):** График наглядно демонстрирует высокую дисперсию. Примечательно, что несмотря на сбои, общее время удержания в вертикали у HC очень велико. Это говорит о том, что в удачных эпизодах HC удерживает шест очень жестко и точно, но не обладает робастностью.
- **BRS (Рис. 8):** Высокий уровень ошибки и низкое время удержания.

4.4. Выводы и оценка целесообразности использования градиентных методов

На основе полученных данных можно провести итоговое сравнение алгоритмов и ответить на вопрос о целесообразности применения градиентных методов (таких как Actor-Critic) для задач данной размерности.

4.4.1. Преимущества и недостатки алгоритмов

1. BestBRS (Random Search с памятью):

- *Преимущества:* Абсолютный лидер по стабильности и качеству управления (награда ≈ 993). Предельно прост в реализации.
- *Недостатки:* Не является обучаемым в классическом смысле (не адаптируется онлайн), требует предварительного этапа отбора весов.

2. Actor-Critic (Градиентный метод):

- *Преимущества:* Высокое качество управления (≈ 975), способность обучаться ”с нуля” и адаптироваться. Теоретически масштабируется на сложные задачи.
- *Недостатки:* Высокая вычислительная сложность (расчет градиентов), большое количество гиперпараметров (learning rates, gamma, sigma), сложность в отладке.

3. Hill Climbing:

- *Преимущества:* Простота реализации, хорошее удержание вертикали в удачных эпизодах.
- *Недостатки:* Неустойчивость к стохастике среды, склонность к застреванию в локальных оптимумах, деградация производительности со временем.

4.4.2. Оценка целесообразности градиентных методов

Задача CartPole- характеризуется крайне низкой размерностью пространства поиска: вектор состояния имеет размерность 4, а количество обучаемых весов линейной модели составляет всего $4 \times 2 = 8$ параметров (или меньше, в зависимости от архитектуры).

Сравнивая результаты **BestBRS** (неградиентный метод) и **Actor-Critic** (градиентный метод), можно сделать вывод:

Использование сложных градиентных методов для задач малой размерности (как CartPole) является нецелесообразным.

Таким образом, для простых задач управления с малым числом параметров методы нулевого порядка (эволюционные стратегии, случайный поиск) обеспечивают лучшее соотношение сложности и качества, чем методы градиентного спуска. Градиентные методы раскрывают свой потенциал в задачах с высокой размерностью (например, обучение по изображениям), где случайный поиск становится бессильным из-за ”проклятия размерности”.

5. Заключение

В рамках учебной практики был успешно решен комплекс задач, направленных на сравнительный анализ эффективности градиентных и неградиентных методов обучения с подкреплением применительно к классической задаче стабилизации неустойчивой механической системы «тележка-стержень» (Cart-Pole).

Все поставленные цели и задачи были достигнуты, что подтверждается следующими результатами:

- **Математически описана динамика системы «тележка-стержень»:** Составлены нелинейные дифференциальные уравнения движения для модели с распределенной массой стержня, что позволило создать высокоточную программную симуляцию.
- **Создана программная среда эмуляции:** Разработан модульный симулятор среды Cart-Pole на языке **Python**, использующий метод численного интегрирования для точного расчета эволюции системы во времени без привлечения сторонних физических движков.
- **Программно реализованы алгоритмы:** Реализованы три агента с линейной аппроксимацией политики: **Basic Random Search (BRS)**, **Hill Climbing (HC)** и **Actor-Critic (AC)**.
- **Проведена серия экспериментов и зафиксированы метрики:** Проведен этап подбора гиперпараметров для каждого алгоритма, определены оптимальные конфигурации, и выполнена серия сравнительных экспериментов. Полученные данные (среднее вознаграждение, кумулятивное вознаграждение, RMSE θ , средняя длина эпизода, время удержания вертикали) зафиксированы и использованы для анализа.
- **Сравнены данные и оценена целесообразность градиентных методов:**

1. Установлено, что неградиентный метод **BestBRS** (Random Search с памятью) показал наилучшее соотношение стабильности и качества управления (≈ 993 кумулятивной награды), достигая максимальной длины эпизода в 1000 шагов.
2. Градиентный метод **Actor-Critic (AC)** показал второй лучший результат (≈ 975), подтвердив свою высокую эффективность и потенциал к точной стабилизации, но уступив BestBRS по стабильности.
3. Сделан вывод, что **использование сложных градиентных методов для задач малой размерности** (с небольшим числом обучаемых параметров, как Cart-Pole) **является нецелесообразным**, поскольку методы нулевого порядка (Random Search) обеспечивают сравнимое или превосходящее качество при значительно меньшей вычислительной и программной сложности.

Исходный код разработанных моделей, алгоритмов и скрипты для проведения экспериментов доступны в публичном репозитории на GitHub:

<https://github.com/ChernyackovEugeny/rod-stabilization>

Данные, использованные для построения графиков и метрик, доступны на Google Диске:

https://drive.google.com/drive/folders/1iTSAHL9I19hiVFMesW_ZycqNTcKhcpIB?usp=sharing

Список литературы

- [1] A2C is a special case of PPO / Shengyi Huang, Anssi Kanervisto, Antonin Raffin et al. // arXiv preprint arXiv:2205.09123. — 2022.
- [2] Array programming with NumPy / Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt et al. // Nature. — 2020. — Vol. 585, no. 7825. — P. 357–362.
- [3] Astrom Karl J., Hagglund Tore. PID Controllers: Theory, Design, and Tuning. — Instrument Society of America, 1995.
- [4] Barto Andrew G., Sutton Richard S., Anderson Charles W. Neuron-like adaptive elements that can solve difficult learning control problems // IEEE Transactions on Systems, Man, and Cybernetics. — 1983. — no. 5. — P. 834–846.
- [5] Evolution Strategies as a Scalable Alternative to Reinforcement Learning / Tim Salimans, Jonathan Ho, Xi Chen et al. // arXiv preprint arXiv:1703.03864. — 2017.
- [6] Human-level control through deep reinforcement learning / Volodymyr Mnih, Koray Kavukcuoglu, David Silver et al. // Nature. — 2015. — Vol. 518, no. 7540. — P. 529–533.
- [7] Hunter J. D. Matplotlib: A 2D graphics environment // Computing in Science & Engineering. — 2007. — Vol. 9, no. 3. — P. 90–95.
- [8] Konda Vijay R., Tsitsiklis John N. Actor-Critic Algorithms // Advances in Neural Information Processing Systems. — 2000. — P. 1008–1014.
- [9] Mania Horia, Guy Aurelia, Recht Benjamin. Simple random search provides a competitive approach to reinforcement learning // Advances in Neural Information Processing Systems. — 2018.
- [10] Matyas J. Random optimization // Automation and Remote Control. — 1965. — Vol. 26. — P. 246–253.

- [11] Ogata Katsuhiko. Modern Control Engineering. — 5th edition. — Prentice Hall, 2010.
- [12] Sutton Richard S., Barto Andrew G. Reinforcement Learning: An Introduction. — Second edition. — MIT Press, 2018.
- [13] Williams Ronald J. Simple statistical gradient-following algorithms for connectionist reinforcement learning // Machine Learning. — 1992. — Vol. 8, no. 3-4. — P. 229–256.