Санкт-Петербургский государственный университет

Кафедра прикладной кибернетики

Группа 24.Б82-мм

Алгоритмы mikmik

Черняков Евгений Олегович

Отчёт по учебной практике
в форме «Сравнение»

Научный руководитель:
профессор, научный сотрудник кафедры прикладной кибернетики Мокаев Р. Н.

Санкт-Петербург
2025

Оглавление

Введение **3**

1. Постановка задачи **5**

2. Обзор **6**

2.1. Обзор существующих решений 6

2.2. Обзор используемых технологий 9

2.3. Выводы 11

3. Описание решения **13**

3.1. Первая задача 13

3.2. Вторая задача 20

3.3. Третья задача 23

4. Эксперимент **35**

4.1. Условия эксперимента и подбор гиперпараметров 35

4.2. Сравнительный анализ метрик 38

4.3. Результаты 40

Список литературы **41**

Введение

Задача стабилизации перевернутого маятника 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 открытым вопрос: оправдано ли усложнение алгоритма (переход от случайного поиска к градиентной политике) для системы с малым количеством степеней свободы, и как именно соотносятся затраты на вычисления и качество стабилизации при реализации физической модели «с нуля», без использования готовых библиотек симуляции.

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 (Policy Gradient) — в задаче балансировки стержня. Необходимо определить, какой из подходов обеспечивает наилучший баланс между скоростью обучения и устойчивостью удержания маятника. Работа призвана продемонстрировать, что для определенных классов задач простые безградиентные методы могут составлять

конкуренцию более сложным подходам RL, а также показать влияние точности физической симуляции на процесс обучения агентов.

Для реализации поставленной задачи в работе разрабатывается программная 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 Policy Gradient. В ходе экспериментов проводится обучение агентов с фиксацией ключевых метрик: времени сходимости алгоритма и длительности удержания стержня в вертикальном положении. 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, Policy Gradient;
4. провести серию экспериментов по обучению агентов каждым из алгоритмов, зафиксировав метрики скорости сходимости и длительности удержания стержня в вертикальном положении;
5. сравнить полученные данные, выявить преимущества и недостатки каждого алгоритма и оценить целесообразность использования градиентных методов для задач данной размерности.

2. Обзор

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 ее 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 с распределенной массой (стержень), а также работы, напрямую сравнивающие эффективность стохастической оптимизации (Random Search, Hill Climbing) с градиентными методами.

- Идентификация технологий: Изучались принципы работы выбранных алгоритмов (PG, HC, BRS) и современные методы их реализации на языке Python.

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 четко 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 идеальный пример нелинейной, существенно неустойчивой системы. Математическое описание основывается на законах Ньютона или уравнениях Лагранжа. В большинстве ранних работ используется модель, где масса стержня считается точечной, однако для точной симуляции требуется модель с учетом распределенной массы (стержня), что усложняет расчеты, но повышает точность реализации.

2.1.2. Классические подходы к стабилизации

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

1. **Линеаризация и LQR-регулирование (Linear Quadratic Regulator).** Этот подход является золотым стандартом в ТАУ. Преимущество LQR заключается в том, что он гарантирует оптимальность управления (минимизацию заданного квадратичного функционала) для линеаризованной системы, обеспечивая высокую скорость реакции вблизи точки равновесия. Недостаток метода в том, что он требует точной линеаризации уравнений вокруг вертикального положения и может стать неэффективным при больших начальных отклонениях стержня.

2. **ПИД-регулирование (PID Control).** Преимуществом ПИД-регулятора является его простота и универсальность. Он не требует полного знания уравнений движения. Недостаток заключается в сложности подбора трех коэффициентов (K_p, K_i, K_d) для достижения устойчивости в широком диапазоне состояний, особенно в нелинейных зонах.

3. **Управление на основе фазового пространства.** Некоторые решения используют переключение между различными линейными стратегиями в зависимости от текущего состояния (угла и уг-

ловой скорости), но их разработка трудоемка и слабо масштабируется на более сложные робототехнические системы.

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

С развитием вычислительных мощностей и нейронных сетей методы RL стали доминирующими в задачах, требующих адаптивного управления.

1. **Value-Based методы (DQN, Q-learning).** Эти алгоритмы эффективны для дискретного пространства действий, что применимо к задаче, если управляющую силу $F(t)$ дискретизировать (например, $F \in \{-10, 0, 10\}$). Однако их сложно масштабировать, и они требуют аппроксимации функции ценности, что часто приводит к нестабильности обучения.

2. **Policy-Based методы (Policy Gradient, Actor-Critic, PPO).** Эти методы непосредственно оптимизируют политику π, что делает их идеальными для задач с непрерывным пространством действий и состояний. Преимущество Policy Gradient (PG), используемого в данной работе, заключается в его фундаментальности, позволяя агенту обучаться, лишь наблюдая за результатами своих действий (наградой), без явного моделирования среды. Недостаток PG — высокая дисперсия градиента, требующая большого числа выборок для стабилизации.

2.1.4. Обоснование выбора методологии

Данное исследование не ставит целью изобретение нового алгоритма стабилизации. Вместо этого, работа фокусируется на методологическом сравнении трех фундаментально разных классов оптимизации — наивного поиска (BRS), локального поиска (HC) и градиентной оптимизации (PG). В существующей литературе, как правило, сравниваются только передовые алгоритмы RL (A2C vs. PPO [1]) , тогда как прямое

сравнение их эффективности с простыми стохастическими методами на базовой, но точно реализованной физической модели встречается редко. Этот анализ позволит определить, насколько сложность PG оправдана по сравнению с его простейшими аналогами для систем низкой размерности.

2.2. Обзор используемых технологий

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, PG) 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 трех методов, представляющих разные уровни сложности и подходы к оптимизации функций. Обзорная литература [3, 4] показывает, что эти методы формируют логическую иерархию, позволяющую провести глубокое методологическое сравнение:

1. **Basic Random Search (BRS):** Простейший безградиентный метод.

2. **Hill Climbing (HC):** Эвристический метод, использующий локальный случайный поиск для ускорения сходимости BRS.

3. **Policy Gradient (PG):** Градиентный метод обучения с подкреплением, который теоретически должен превосходить безградиентные методы по скорости сходимости в задачах оптимизации [4].

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
Policy Gradient	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 PG) пропорциональный рост скорости сходимости в задаче стабилизации низкой размерности. BRS и HC выступают в роли надежных, но простых контрольных групп для оценки эффективности градиентного подхода.

2.2.2. 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 [2].

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**

в связке с фреймворком **PyTorch** значительно ускоряет разработку, упрощает реализацию Policy Gradient за счет автоматического дифференцирования и обеспечивает лучшую читаемость кода, что критически важно для курсовой работы.

2.2.3. Используемые библиотеки и инструменты

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

- **Язык программирования: Python**.
- **Численные расчеты и симуляция:** Библиотека **NumPy**. Используется для эффективного хранения и оперирования матрицами весовых коэффициентов агентов.
- **Реализация алгоритма Policy Gradient:** Фреймворк **PyTorch**. Выбран для удобства работы со стохастическими распределениями (семплирование действий агента), необходимых для алгоритма Policy Gradient.
- **Визуализация и анализ данных:** Библиотека **Matplotlib**. 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, PG) 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 /quoteСреды (Environment) в терминологии обучения с подкреплением.

Третий этап (подраздел 3.3) посвящен алгоритмической реализации агентов управления. Были запрограммированы 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. Первая задача

Для описания динамики системы «перевернутый маятник на тележке» воспользуемся вторым законом Ньютона. Система состоит из те-

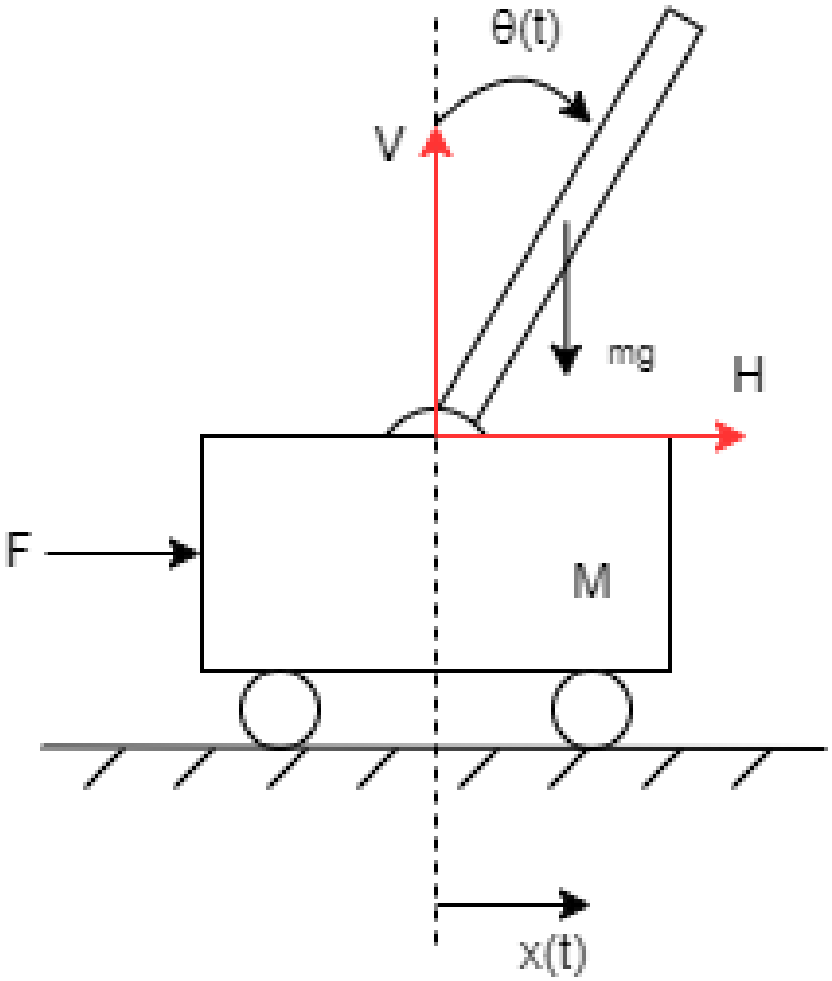

Рис. 1: Схема системы «Перевернутый маятник на тележке» с действующими силами.

лежки массой 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:

$$x_c = x + a\sin\theta, \quad y_c = a\cos\theta. \tag{1}$$

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

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

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):

$$M\ddot{x} = F - H. \tag{3}$$

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

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

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

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

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:

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

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:

$$ma\cos\theta\,\ddot{x} + I\ddot{\theta} = -mga\sin\theta. \tag{7}$$

3.1.3. Разрешение относительно старших производных

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

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

Определитель матрицы системы равен:

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

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

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

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

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):

$$s = \begin{pmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{pmatrix}. \tag{12}$$

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

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

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

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

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:

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

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 (гашение колебаний и стабилизация).

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

Для симуляции движения системы (переход $s_t \to s_{t+\Delta t}$) 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** $s_{t+\Delta t}$**:**

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

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:

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

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 (гашению скоростей), но не делает это приоритетом перед удержанием положений.

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

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

* 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. Вторая задача

Для моделирования динамики системы «Перевернутый маятник на тележке» и создания среды для обучения с подкреплением (RL) разработан класс `Model` на языке Python. Этот класс инкапсулирует математическую модель, численный решатель и логику взаимодействия,

соответствующую стандарту `Gym`.

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 $\mathbf{g}(s, F)$: 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)). Для повышения устойчивости расчетов в знаменателе (Δ) реализована **численная стабилизация** с ограничением $\max(\Delta, 10^{-4})$, что предотвращает ошибку деления на ноль.

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»

```python
import numpy as np

class Model():
    def __init__(self, episode_length=1000, Fmax=10, dt=0.01):
        self.l = 1.0
        self.m = 0.5
        self.M = 1.2
        self.tetta_fail = np.pi/2
        self.x_fail = 10.0
        self.Fmax = Fmax
        self.dt = dt
        self.episode_length = episode_length

    def step(self, state, action):
        action = np.clip(action, -self.Fmax, self.Fmax)
        new_state = self.apply_action(state, action, self.dt)
        reward = self.reward(new_state)
        done = abs(new_state[0]) > self.x_fail or abs(new_state[2]) > self.tetta_fail
        return new_state, reward, done

    def g(self, state, action):
        x, x_der, tetta, tetta_der = state
        g = 9.8
        I = self.m * self.l**2 / 3
        a = self.l / 2
        # Вычисление определителя Delta (знаменатель)
        delta = I * (self.M + self.m) - (self.m * a * np.cos(tetta))**2
        delta = max(delta, 1e-4)  # Численная стабилизация
        x_2der = (I * (action + self.m * a * tetta_der**2 * np.sin(tetta)) +
                  self.m**2 * g * a**2 * np.cos(tetta) * np.sin(tetta)) / delta
        tetta_2der = (-self.m * a * np.cos(tetta) * (action + self.m * a * tetta_der**2 *
        ↪  np.sin(tetta)) -
                      (self.M + self.m) * self.m * g * a * np.sin(tetta)) / delta
        return np.array([x_der, x_2der, tetta_der, tetta_2der])

    def apply_action(self, state, action, dt):
        # Метод Рунге-Кутты 4-го порядка (RK4)
        k1 = self.g(state, action)
        k2 = self.g(state + dt * 0.5 * k1, action)
        k3 = self.g(state + dt * 0.5 * k2, action)
        k4 = self.g(state + dt * k3, action)
        new_state = state + dt * (1/6 * (k1 + 2*k2 + 2*k3 + k4))
```

```
        return new_state

    def reward(self, state):
        x, x_der, tetta, tetta_der = state
        # Базовое вознаграждение
        r = 1 - (0.1 * tetta ** 2 + 0.1 * x ** 2 + 0.01 * tetta_der ** 2 + 0.01 * x_der
        ↪  ** 2)
        # Дополнительный штраф за сбой
        if abs(x) > self.x_fail or abs(tetta) > self.tetta_fail:
            r -= 10
        return r
```

3.3. Третья задача

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

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 $\mathbf{s} = [x, \dot{x}, \theta, \dot{\theta}]^{\mathrm{T}}$ с управляющей силой F:

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

Принцип работы BRS основан на глобальном случайном поиске:

- **Генерация:** На каждой итерации (эпизоде) генерируется совершенно новый вектор весов $\mathbf{w}$ из равномерного распределения в заданном диапазоне.
- **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 $\mathbf{w}_{\text{best}}$.

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

Детали программной реализации Класс `BRSagent` инкапсулирует логику поиска.

- Метод `learn` реализует основной цикл, где ключевая строка `self.w = np.random.uniform(-1, 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

```python
import numpy as np
from src.Model import Model

class BRSagent():
    def __init__(self, model=Model(), n_episodes=100):
        # Инициализация случайными весами (линейная политика)
        self.w = np.random.uniform(-1, 1, size=4)
        self.n_episodes = n_episodes
        self.model = model
        self.best_w = self.w.copy()
        self.best_reward = -np.inf
        self.states = []
        self.actions = []
        self.rewards = []

    def learn(self):
        for episode_ind in range(self.n_episodes):
            # Полностью новый случайный вектор весов
            self.w = np.random.uniform(-1, 1, size=4)
            r = self.rollout(self.w)
            # Сохранение лучшей политики
            if r > self.best_reward:
                self.best_reward = r
                self.best_w = self.w.copy()

    def rollout(self, w):
        state = self.reset_state()
        total_reward = 0
        done = False
        steps = 0
        while not done and steps < self.model.episode_length:
            # Линейная политика с ограничением F_max
            action = np.clip(np.dot(w, state), -self.model.Fmax, self.model.Fmax)
            new_state, reward, done = self.model.step(state, action)
            self.states.append(state)
            self.actions.append(action)
            self.rewards.append(reward)
            state = new_state
            total_reward += reward
            steps += 1
        return total_reward

    def reset_state(self):
```

```
        # Инициализация состояния в случайном диапазоне
        self.states = []
        self.actions = []
        self.rewards = []
        state = np.array([
            np.random.uniform(-1.0, 1.0),          # x
            np.random.uniform(-0.1, 0.1),          # x_dot
            np.random.uniform(-np.pi/6, np.pi/6), # tetta
            np.random.uniform(-0.1, 0.1)           # tetta_dot
        ], dtype=float)
        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 $\mathbf{w}_{\text{best}}$, формируя политику-кандидат $\mathbf{w}'$:

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

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

Принцип работы HC:

- **Генерация кандидата:** Политика-кандидат $\mathbf{w}'$ генерируется путем добавления гауссовского шума к текущему лучшему вектору весов $\mathbf{w}_{\text{best}}$.
- **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 $\mathbf{w}_{\text{best}}$. Если улучшения нет, $\mathbf{w}_{\text{best}}$ остается неизменным.

Данная реализация включает механизм **затухания** σ (`sigma_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. HC, как правило, сходится быстрее, чем BRS, но более подвержен риску застревания в локальных оптимумах.

Детали программной реализации 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 σ 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

```python
import numpy as np
from src.Model import Model

class HCagent():
    def __init__(self, model=Model(), sigma=0.5, sigma_decay=0.995, n_episodes=100):
        self.w = np.random.uniform(-1, 1, size=4)
        self.best_w = self.w.copy()
        self.sigma = sigma              # Параметр шума (радиус поиска)
        self.sigma_decay = sigma_decay  # Коэффициент затухания шума
        self.n_episodes = n_episodes
        self.model = model
        self.best_reward = -np.inf
        self.states = []
        self.actions = []
        self.rewards = []

    def learn(self):
        for episode_ind in range(self.n_episodes):
            # Генерация гауссовского шума (дельта)
            delta = np.random.normal(0, self.sigma, size=4)
            # Новый кандидат w'
            new_w = self.w + delta
            reward = self.rollout(new_w)
            # Обновление w только при улучшении (восхождение)
            if reward > self.best_reward:
                self.w = new_w
                self.best_reward = reward
                self.best_w = new_w.copy()
            # Затухание радиуса поиска
            self.sigma = max(self.sigma * self.sigma_decay, 0.01)

    def rollout(self, w):
        state = self.reset_state()
        total_reward = 0
        done = False
        steps = 0
        while not done and steps < self.model.episode_length:
            # Линейная политика с ограничением F_max
            action = np.clip(np.dot(w, state), -self.model.Fmax, self.model.Fmax)
            new_state, reward, done = self.model.step(state, action)
            self.states.append(state)
            self.actions.append(action)
```

```
            self.rewards.append(reward)
            state = new_state
            total_reward += reward
            steps += 1

        return total_reward

    def reset_state(self):
        # Инициализация состояния в случайном диапазоне
        self.states = []
        self.actions = []
        self.rewards = []
        state = np.array([
            np.random.uniform(-1.0, 1.0),          # x
            np.random.uniform(-0.1, 0.1),          # x_dot
            np.random.uniform(-np.pi/6, np.pi/6), # tetta
            np.random.uniform(-0.1, 0.1)           # tetta_dot
        ], dtype=float)
        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 линейных аппроксиматоров:

1. **Actor (Актер):** Определяет стохастическую политику $\pi_\theta(a|s)$, выбирая действие a в состоянии s. Его цель — максимизировать ожидаемую награду.

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 $\mathbf{s}$:

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

Обновление весов происходит на каждом шаге (online-обучение) на основе ошибки временной разности (TD-error), которая показывает, насколько действие оказалось лучше или хуже ожидаемого:

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

Градиенты для обновления параметров Актера выводятся через теорему о градиенте политики (Policy Gradient Theorem) и логарифмическую производную функции плотности вероятности нормального распределения.

Детали программной реализации Класс `ACagent` содержит параметры для обучения ($\alpha_{actor}, \alpha_{critic}, \gamma$) и весовые коэффициенты: `w` (для Критика), `tetta_mu` и `tetta_sigma` (для Актера).
Ключевые методы реализации:

- **Расчет TD-ошибки:** В методе `learn` вычисляется `TD_error`, которая выступает «учителем» для Актера. Если ошибка положительна, вероятность выбранного действия увеличивается, если отрицательна — уменьшается.
- **Обновление Критика:** Веса `w` обновляются методом стохастического градиентного спуска, чтобы $V(s)$ приближалась к реальной полученной награде.
- **Обновление Актера:** Используются функции `acceptance_mu` и `acceptance_sigma`, которые вычисляют градиенты $\nabla_\theta \ln \pi(a|s)$. Для μ градиент пропорционален $\frac{a-\mu}{\sigma^2}$, а для σ (учитывая экспоненциальную параметризацию) — $(\frac{(a-\mu)^2}{\sigma^2} - 1)$.
- **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

```python
import numpy as np
from src.Model import Model

class PGagent():
    def __init__(self, lr_actor=1e-3, lr_critic=1e-2, gamma=0.99, model=Model(), n_episodes=100):
        self.lr_actor = lr_actor    # Скорость обучения Актера
        self.lr_critic = lr_critic  # Скорость обучения Критика
        self.gamma = gamma          # Коэффициент дисконтирования
        self.model = model
        self.n_episodes = n_episodes
        # Инициализация весов
        self.w = np.random.uniform(-1, 1, size=4)       # Веса Критика (Value func)
        self.tetta_mu = np.zeros(4)                     # Веса Актера (Мат. ожидание)
        self.tetta_sigma = np.ones(4) * 0.1             # Веса Актера (Дисперсия)
        # Параметры ограничения (clipping) для стабильности
        self.sigma_clip_min = 0.05
        self.sigma_clip_max = 2.0
        self.tetta_sigma_clip = 5.0
        self.tetta_mu_clip = 10
        self.states = []
        self.actions = []
        self.rewards = []

    def learn(self):
        for episode_ind in range(self.n_episodes):
            self.rollout()
            # Пошаговое обновление весов (Online learning)
            for i in range(len(self.actions)):
                action, state, reward = self.actions[i], self.states[i], self.rewards[i]
                # Оценка ценности следующего состояния V(s')
                next_value = self.value_func(self.states[i+1]) if i+1 < len(self.states) else 0
                # Вычисление ошибки временной разности: delta = r + gamma*V(s') - V(s)
                TD_error = reward + self.gamma * next_value - self.value_func(state)
                # Обновление весов Актера (Policy Update)
                # acceptance_... реализуют градиент логарифма правдоподобия
                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,
                   ↪  self.tetta_sigma_clip)
40                 self.tetta_mu = np.clip(self.tetta_mu, -self.tetta_mu_clip,
                   ↪  self.tetta_mu_clip)
41                 # Обновление весов Критика (Value Update)
42                 # Градиент V(s) по w равен самому вектору состояния state (так как V
                   ↪  линейна)
43                 self.w += self.lr_critic * TD_error * state
44
45     def rollout(self):
46         state = self.reset_state()
47         total_reward = 0
48         done = False
49         steps = 0
50         while not done and steps < self.model.episode_length:
51             # Выбор действия стохастической политикой
52             mu_val = self.mu(state)
53             sigma_val = self.sigma(state)
54             action = self.policy(mu_val, sigma_val)
55             new_state, reward, done = self.model.step(state, action)
56             self.states.append(state)
57             self.actions.append(action)
58             self.rewards.append(reward)
59             state = new_state
60             total_reward += reward
61             steps += 1
62         return total_reward
63
64     def value_func(self, state):
65         # Линейная аппроксимация V(s) = w^T * s
66         return np.dot(state, self.w)
67
68     def policy(self, mu, sigma):
69         # Семплирование действия из нормального распределения
70         a = np.random.normal(mu, sigma)
71         return np.clip(a, -self.model.Fmax, self.model.Fmax)
72
73     def sigma(self, s):
74         # Вычисление стандартного отклонения: sigma = exp(theta_sigma^T * s)
75         z = np.dot(s, self.tetta_sigma)
76         z = np.clip(z, -10, 10) # Защита от переполнения экспоненты
77         sigma = np.clip(np.exp(z), self.sigma_clip_min, self.sigma_clip_max)
78         return sigma
79
80     def mu(self, s):
81         # Вычисление мат. ожидания: mu = theta_mu^T * s
82         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          # Учитывает, что sigma параметризована через экспоненту
94          acceptance = state * ((action - self.mu(state)) ** 2 / (self.sigma(state) ** 2 +
            ↪  1e-8) - 1)
95          return acceptance
96
97      def reset_state(self):
98          self.states = []
99          self.actions = []
100         self.rewards = []
101         state = np.array([
102             np.random.uniform(-1.0, 1.0),          # x
103             np.random.uniform(-0.1, 0.1),          # x_dot
104             np.random.uniform(-np.pi/6, np.pi/6), # tetta
105             np.random.uniform(-0.1, 0.1)           # tetta_dot
106         ], dtype=float)
107         return state
```

4. Эксперимент

Как мы проверяем, что всё удачно получилось. Если работа рассчитана на несколько семестров и в текущем до эксперимента дело не дошло, опишите максимально подробно, как он будет проводиться и на чём (то, что называется *дизайн эксперимента* — от того, что и как Вы будете проверять, очень сильно зависит, что Вы будете делать, так что это важно и делается *не* после реализации).

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.
Также было исследовано влияние параметров нормального распределе-

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

ния (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 таблице 5 показано влияние начальной инициализации. В отличие от BRS, алгоритм HC оказался чувствителен к начальной точке: нормальное распределение с $\mu = 1$ показало крайне низкие результаты, в то время как равномерное распределение обеспечило стабильную сходимость.

Параметры шума (Таблица 6) определяют способность алгоритма выхо-

Таблица 5: HC: Сравнение способов инициализации

Параметры инициализации	Средняя награда
Uniform [-0.01, 0.01]	999.85
Uniform [-0.4, 0.4]	**999.91**
Normal ($\mu = 0, \sigma = 1$)	898.58
Normal ($\mu = 1, \sigma = 1$)	581.22

дить из локальных минимумов. Эксперименты показали, что медленное затухание шума (высокий коэффициент) критически важно для точной настройки весов.

Таблица 6: HC: Влияние параметров шума (σ_{noise}) и затухания

Дисперсия шума	Награда	Коэф. затухания	Награда
0.01	936.27	0.9	840.36
0.5	967.40	0.99	947.28
1.0	**999.87**	**0.999**	**999.88**
-	-	1.0	999.85

Выбор: Для HC выбрана инициализация `Uniform[-0.4, 0.4]`, начальная дисперсия шума $\sigma = 1.0$ и коэффициент затухания **0.999**, обеспечивший наивысшую среднюю награду.

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 $(\alpha_{actor}, \alpha_{critic})$.

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

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

Таблица 7: 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
$\mathbf{5 \cdot 10^{-4}}$	$\mathbf{1 \cdot 10^{-3}}$	**999.52**

Таблица 8: 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

4.1.4. Итоговая конфигурация

На основе проведенных экспериментов сформирована итоговая конфигурация гиперпараметров для сравнительного анализа (Таблица 9).

Таблица 9: Итоговые параметры алгоритмов для сравнительного эксперимента

Алгоритм	Параметры
BRS	$w \sim N(-1, 1)$
HC	$w \sim U[-0.4, 0.4]$, $\sigma_{noise} = 1.0$, decay= 0.999
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 10.

Таблица 10: 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

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] Brooks Jr. Frederick P. The Mythical Man-month (Anniversary Ed.). — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. — ISBN: 0-201-83595-9.

[3] Sutton Richard S., Barto Andrew G. Reinforcement Learning: An Introduction. — MIT Press, 1998. — ISBN: 0262193981.

[4] Williams Ronald J. Simple statistical gradient-following algorithms for connectionist reinforcement learning // Machine Learning. — 1992. — Vol. 8, no. 3-4. — P. 229–256.