



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра высшей математики

КУРСОВАЯ РАБОТА
по дисциплине
«Объектно-ориентированное программирование»

Тема курсовой работы
«Обучение с подкреплением на примере воздушного хоккея»

Студент группы КМБО-03-22


Лыков Д.С.

Руководитель курсовой работы
доцент кафедры Высшей математики
к.ф.-м.н.

Петрусевиц Д.А.

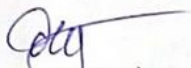
Работа представлена к
защите

«1» фев 20 23 г.


(подпись студента)

«Допущен к защите»

«1» фев 20 23 г.


(подпись руководителя)



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра высшей математики

Утверждаю

Исполняющий обязанности заведующего
кафедрой М.М.Шатин А.В.Шатина

«22» сентября 2023 г.

ЗАДАНИЕ
на выполнение курсовой работы
по дисциплине «Объектно-ориентированное программирование»

Студент *Лыков Д.С.*

Группа *КМБО-03-22*

1. Тема: «Обучение с подкреплением на примере воздушного хоккея»

2. Исходные данные:

Построить класс для модели реализации обучения с подкреплением (метод UCS, как минимум)
На игровом поле присутствует два слайдера, защищающие ворота, и шайба. Агент может наносить удар с двумя характеристиками: направление и импульс шайбе. Она испытывает упругое соударение со слайдером или бортами. За пропуск шайбы даётся максимальный штраф, большое поощрение за гол.
Реализовать переход между обучением и стационарным состоянием агента в виде эпсилон-жадной стратегии

3. Перечень вопросов, подлежащих разработке, и обязательного графического материала:


Продemonстрировать изменение распределения вероятностей выбора действия (уровень импульса и соотношение между направлениями, по которому прилетела до и улетела после удара шайба)

Продemonстрировать изменение выигрыша агента со временем

4. Срок представления к защите курсовой работы: до «22» декабря 2023 г.

Задание на курсовую
работу выдал

«22» сентября 2023 г.

 (Петрусеви́ч Д.А.)

Задание на курсовую
работу получил

«22» сентября 2023 г.

 (Лыков Д.С.)

Оглавление

Глава I. Теоретическая часть	3
Обучение с подкреплением. Общие сведения.....	3
Частный случай обучения с подкреплением. Задача о многоруком бандите.....	3
Алгоритмы решения задачи о многоруких бандитах. Greedy Epsilon-Greedy, Softmax, Upper Confidence Bound (UCB).....	5
Среды с состояниями. Общая постановка задачи. Алгоритмы решения.....	9
Глава II. Практическая часть.....	14
Класс GameRenderer. Создание объектов и физики игры Air hockey	14
Класс Agent. Создание агента для задачи обучения с подкреплением. Методы обучения агента....	17
Класс Game. Логика игры агента в Air Hockey	18
Класс AgentStatistics. Сравнение стратегий для обучения агента в воздушном хоккее. Эффективность стратегии с различным значением параметра	20
Заключение.....	24
Список литературы	25
Приложения	26
Приложение 1	26
Приложение 2	44

Глава I. Теоретическая часть

Обучение с подкреплением. Общие сведения

Определение. Обучение с подкреплением (Reinforcement Learning, RL) — это раздел машинного обучения, в котором агент обучается принимать последовательность действий в среде для достижения определенных целей.

Идея. Обучение модели, которая не имеет сведений о системе, но имеет возможность производить какие-либо действия в ней. Действия переводят систему в новое состояние и модель получает от системы некоторое вознаграждение.

Ключевые компоненты Reinforcement Learning:

- *Агент* - сущность, которая принимает решения в среде. Агент имеет свое внутреннее состояние и может выполнять действия для взаимодействия со средой.
- *Среда* - внешняя система, с которой агент взаимодействует. Среда может быть физической или виртуальной. Среда также имеет свое состояние.
- *Действия* - набор действий, которые агент может выполнять в среде. Действия могут быть дискретными (например, выбор из ограниченного числа вариантов) или непрерывными (например, выбор числа в интервале).
- *Состояния* - описание текущего состояния среды или агента. В некоторых задачах состояния могут быть полностью наблюдаемыми (агент знает все о состоянии), а в других - частично наблюдаемыми (агент видит только часть информации).
- *Награды* - числовая оценка, которую агент получает от среды после выполнения каждого действия.
- *Политика* - стратегия, определяющая, какие действия агент должен выполнять в зависимости от текущего состояния. Политика может быть детерминированной (фиксированным правилом) или стохастической (вероятностным распределением).

Ключевые этапы Reinforcement Learning:

- *Исследование и Эксплуатация.* Агенту нужно балансировать между исследованием новых действий и эксплуатацией знаний, чтобы максимизировать награду.
- *Цель.* Агенту нужно определить цель задачи, обычно выраженную в виде функции награды. Цель агента - максимизировать награду, выбирая подходящие действия.
- *Обучение.* Агент обучается на основе накопленного опыта. Он может использовать различные алгоритмы обучения, такие как метод временных разностей, UCB, Q-обучение, обучение с помощью глубокого обучения (Deep Reinforcement Learning), и другие.
- *Оценка и обновление.* Агент оценивает, насколько хороши его текущие действия в среде, и обновляет свою политику для улучшения результатов.

Частный случай обучения с подкреплением. Задача о многоруком бандите

Понятие. *Задача о многоруком бандите (Multi-Armed Bandit Problem)* — это классическая задача принятия решений и обучения с подкреплением, которая является упрощенной моделью для исследования баланса между исследованием и эксплуатацией.

Название "многорукий бандит" происходит от аналогии с игрой на игровых автоматах (однорукие бандиты), где игрок сталкивается с несколькими ручками (действиями), каждая из которых может приносить различные награды. В задаче о многоруких бандитах у агента (игрока) есть набор ручек (действий), и его задачей является выбор оптимальной руки (действия) так, чтобы максимизировать суммарную награду, собранную в результате нескольких попыток.

Ключевые элементы задачи о многоруких бандитах:

- *Действия* - ручки, которые агент может выбирать (каждая рука имеет свою вероятность приносить награду).
- *Награды*. Каждая рука дает награду с некоторой вероятностью. Награды могут быть случайными и могут меняться от попытки к попытке.
- *Задача оптимизации*. Цель агента - найти стратегию выбора рук (действий), которая максимизирует ожидаемую суммарную награду после множества попыток.
- *Исследование и эксплуатация*. Агент должен решить, когда выбирать уже известные действия (эксплуатацию) и когда исследовать новые действия для более точного определения их наград (исследование).

Постановка задачи о многоруких бандитах и ее решение:

Дано:

A — множество возможных действий;

$p(r|a)$ — неизвестное распределение премии $r \in \mathbb{R}$ для $a \in A$;

$\pi_t(a)$ — стратегия агента в момент времени t , распределение на A .

Игра агента со средой:

инициализация стратегии $\pi_1(a)$;

для всех $t = 1, \dots, T, \dots$:

агент выбирает действие $a_t \sim \pi_t(a)$;

среда генерирует премию $r_t \sim p(r|a_t)$;

агент корректирует стратегию $\pi_{t+1}(a)$;

Задача агента – максимизировать премию за t раундов!

Замечание. Критическую роль в принятии решений агентом в задаче о многоруких бандитах является знание средней премии в раундах (формула 1.1) и ценности действий (формула 1.2). Эти оценки позволяют агенту лучше понимать, какие действия следует выбирать, чтобы максимизировать его общую награду в условиях ограниченного числа попыток.

$$Q_t(a) = \frac{\sum_{i=1}^t r_i[a_i = a]}{\sum_{i=1}^t [a_i = a]} \quad (1.1)$$

$$Q^*(a) = \lim_{t \rightarrow \infty} Q_t(a) \rightarrow \max \quad (1.2)$$

Чтобы решить задачу о многоруких бандитах, необходимо обучить агента. Для этого существуют алгоритмы, такие как жадная стратегия и эпсилон жадная стратегия (Epsilon-Greedy), Upper Confidence Bound (UCB), Softmax и другие, которые определяют, какие действия следует выбирать в зависимости от текущего знания об агенте.

Алгоритмы решения задачи о многоруких бандитах. Greedy | Epsilon-Greedy, Softmax, Upper Confidence Bound (UCB)

Greedy / Epsilon-Greedy strategies.

Понятие. Жадная стратегия (Greedy Strategy) — это интуитивно понятный метод в обучении с подкреплением. Основная идея жадной стратегии заключается в том, что агент всегда выбирает действие, которое имеет наивысшую оценку ценности в текущем состоянии. Это означает, что агент максимизирует мгновенную награду, не учитывая будущие последствия. Формула 1.4 описывает жадную стратегию через вспомогательную формулу 1.3.

$$A_t = \underset{a \in A}{\operatorname{Arg\,max}} Q_t(a) \quad (1.3)$$

$$\pi_t(a) = \frac{1}{|A_t|} [a \in A_t] \quad (1.4)$$

Плюсы жадной стратегии:

- *Простота.* Жадная стратегия легко понимается и реализуется.
- *Мгновенная награда.* Агент максимизирует текущую награду на каждом шаге.

Минусы жадной стратегии:

- *Не всегда оптимальна.* Жадная стратегия может привести к субоптимальным результатам, поскольку она не учитывает будущие последствия. В некоторых задачах, оптимальные решения могут потребовать жертву мгновенной награды в пользу долгосрочной выгоды.

- *Ограничение в исследовании.* По некоторым действиям a можем так и не набрать статистику для оценки $Q_t(a)$.

Замечание. Чтобы побороть недостаток жадной стратегии, можно включить в рандомизацию возможность применить любое действие с некоторой вероятностью. Так и появился частный случай жадной стратегии – эпсилон-жадная стратегия.

Понятие. Эпсилон-жадная стратегия (Epsilon-Greedy Strategy) — это распространенный метод в обучении с подкреплением, который совмещает в себе элементы жадной стратегии и исследовательской стратегии для достижения баланса между мгновенными наградами и изучением неизведанных вариантов (формула 1.5).

$$\pi_t(a) = \frac{1 - \varepsilon}{|A_t|} [a \in A_t] + \frac{\varepsilon}{|A|} \quad (1.5)$$

Замечание. Сначала нам нужны разведывательные действия и равномерное распределение на множестве всех действия, для того чтобы набрать статистику об этих действиях, но по мере изучения действий параметр ε можно уменьшать, делая стратегию более жадной.

На рисунке 1.1 показано, какую агент получает награду в зависимости от ε .

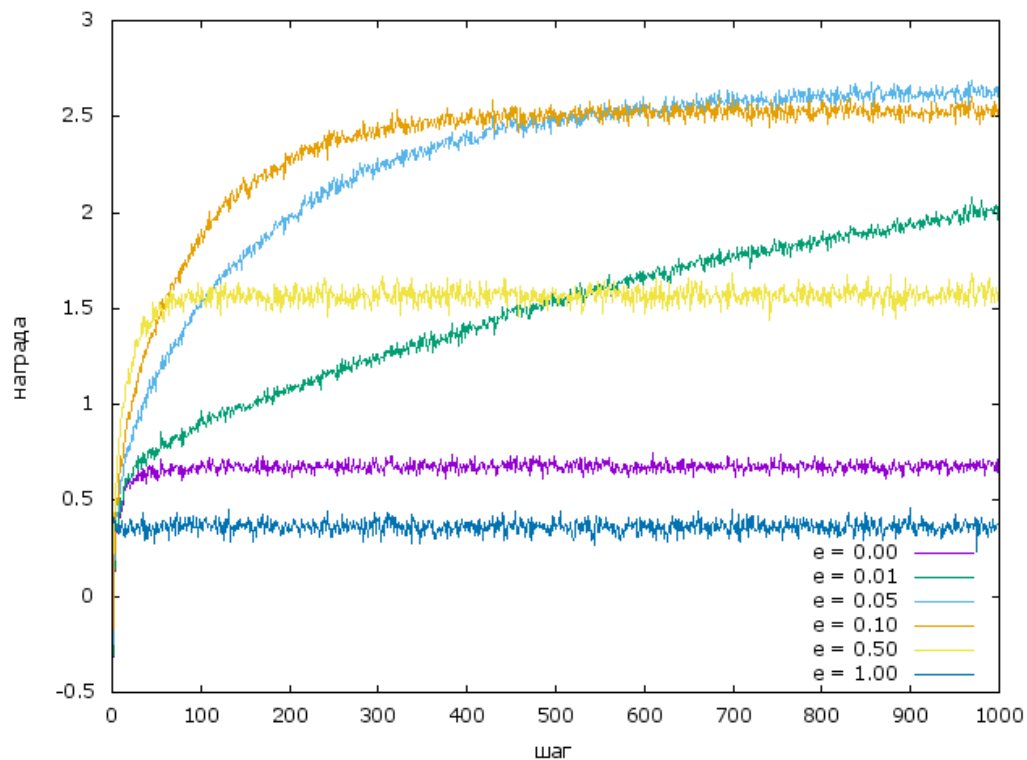


Рисунок 1.1. Пример графика зависимости награды для стратегии с разным параметром ε .

Softmax strategy (распределение Гиббса).

Понятие. Softmax strategy (распределение Гиббса) – еще один метод обучения с подкреплением в задаче о многоруких бандитах. В отличие от эпсилон-жадной стратегии, где случайные действия выбираются с фиксированной вероятностью ϵ , Softmax стратегия выбирает действия с вероятностями, зависящими от их оцененных ценностей. Описание стратегии агента через распределение Гиббса отображено на формуле 1.6.

$$\pi_t(a) = \frac{\exp\left(\frac{1}{\tau} Q_t(a)\right)}{\sum_{b \in A} \exp\left(\frac{1}{\tau} Q_t(b)\right)}, \quad (1.6)$$

где τ – параметр температуры, который при стремлении к 0 делает стратегию более жадной, а при стремлении к бесконечности – равномерной, то есть чисто исследовательской.

Замечание. Параметр τ можно уменьшать со временем, как и параметр ϵ в Epsilon-Greedy Strategy.

Плюсы softmax стратегии:

- *Гибкость в управлении исследованием.* Параметр температуры (τ) позволяет агенту управлять степенью исследования.
- *Исследование и эксплуатация.* Softmax позволяет агенту с учетом ценностей действий балансировать исследование и эксплуатацию.

Минусы softmax стратегии:

- *Дополнительный параметр.* Необходимость выбора значения параметра температуры может быть неочевидной и требует настройки.
- *Вычислительные затраты.*

На рисунке 1.2 можно увидеть сравнение Эпсилон-Жадной и Softmax стратегии в конкретной задаче при разных значениях температуры и ϵ .

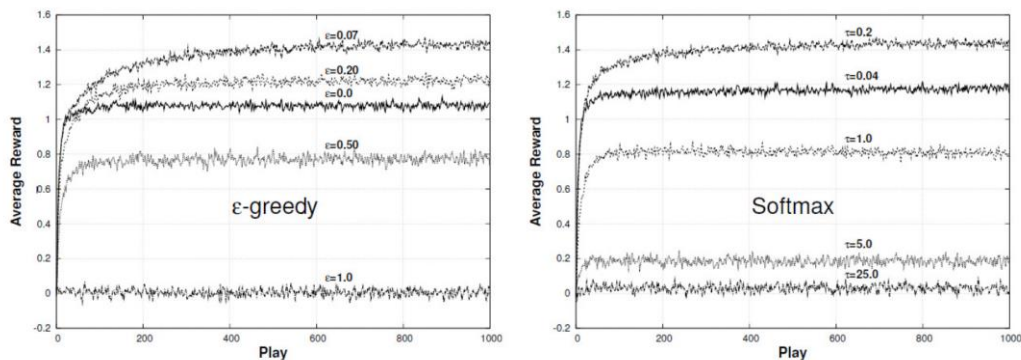


Рисунок 1.2. Пример сравнения Softmax Strategy и Epsilon-Greedy Strategy при различных τ | ϵ .

Upper Confidence Bound (UCB).

Понятие. Upper Confidence Bound (UCB) — это один из лучших и популярных алгоритмов выбора действий в задачах обучения с подкреплением. Этот алгоритм часто называют «полужадным», потому что по сути он является жадным алгоритмом с некоторой поправкой: большая вероятность дается тем действиям, которые мало использовались до сих пор.

Основные идеи UCB в контексте обучения с подкреплением:

- UCB использует понятие верхней границы ценности (Upper Confidence Bound), которая измеряет, насколько агент уверен в том, что конкретное действие даст хорошее вознаграждение.
- *Exploration-Exploitation Trade-off*. Алгоритм балансирует между исследованием новых действий (эксплорацией) и выбором действий с высокой оценкой вознаграждения (эксплуатацией).
- *Выбор действий*. На каждом временном шаге агент оценивает верхнюю границу ценности для каждого доступного действия и выбирает действие с самой высокой верхней границей. Это позволяет агенту исследовать новые действия, которые могут иметь потенциал для большего вознаграждения, и в то же время выбирать действия с хорошими оценками вознаграждения.

$$A_t = \underset{a \in A}{\operatorname{Arg\,max}} \left(Q_t(a) + \delta \sqrt{\frac{2 \ln t}{k_t(a)}} \right), \text{ где } k_t(a) = \sum_{i=1}^t [a_i = a]; \quad (1.7)$$

δ – параметр \exp/\exp компромисса.

Замечание. В формуле 1.7 можно менять параметр δ и $k_t(a)$. Чем меньше $k_t(a)$, тем менее исследована стратегия, тем выше должна быть вероятность выбрать a . Чем больше δ , тем стратегия более исследовательская (параметр δ можно уменьшать со временем).

Плюсы алгоритма UCB:

- *Эффективность в исследовании*. UCB хорошо сбалансирован между исследованием новых действий и эксплуатацией лучших действий.
- *Доверительные интервалы*. UCB использует доверительные интервалы для оценки верхней границы уверенности в отношении вознаграждения. Это позволяет агенту принимать решения на основе статистических оценок и учитывать неопределенность в данных.

Минусы алгоритма UCB:

- *Вычислительная сложность*. Расчет верхней границы уверенности может быть вычислительно затратным, особенно в задачах с большим количеством действий.
- *Чувствительность к модели*. UCB может быть чувствителен к выбору модели или функции расчета верхней границы уверенности. Неправильный выбор модели может привести к неправильным решениям.
- *Не учитывает долгосрочные последствия*. UCB обычно рассматривает только моментальное вознаграждение и не учитывает долгосрочные последствия действий.

- *Неспособность адаптироваться к изменениям.* UCB может быть менее эффективным в ситуациях, где динамика задачи меняется со временем или структура вознаграждения нестационарна.

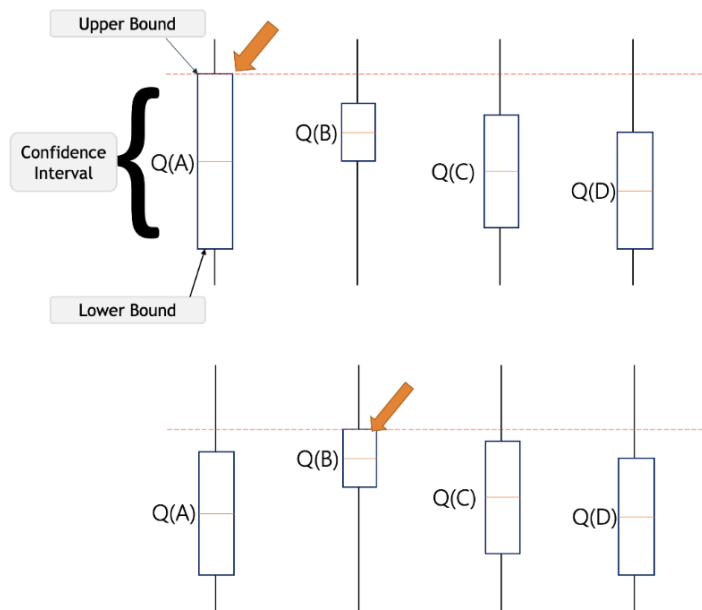


Рисунок 1.3. Принцип работы *Upper Confidence Bound (UCB)*.

Вывод: Итак, задача о многоруких бандитах является классической задачей в области обучения с подкреплением, которая моделирует ситуацию, где решающая система должна выбирать действия из ограниченного набора ресурсов с целью максимизации общей награды. Ключевыми компонентами задачи являются действия, награды, задача оптимизации, исследование и эксплуатация. Чтобы решить задачу о многоруких бандитах, необходимо обучить агента. Это делается с помощью алгоритмов: Жадная стратегия и Эпсилон-Жадная стратегия (Epsilon-Greedy), UCB, Softmax и другие. В задаче о многоруких бандитах нет идеальной стратегии обучения агента! UCB более подходит для среды, где вознаграждения более стохастичны. Эпсилон-жадная стратегия может быть хорошим выбором, когда стоимость ошибок невелика. Softmax стратегия полезна в средах, где различие в ценности ручек невелико.

Среды с состояниями. Общая постановка задачи. Алгоритмы решения

Понятие. Обучение с подкреплением в среде с состоянием (State-Based Reinforcement Learning) — это подход к решению задач искусственного интеллекта, в которых агент взаимодействует с окружающей средой, имеющей определенное состояние. В данном контексте, "состояние" представляет собой описание текущего положения среды, которое агент использует для принятия решений и максимизации награды.

Ключевые аспекты задачи в средах с состояниями такие же, как и в задаче о многоруких бандитах с добавлением состояния среды.

Постановка задачи в средах с состояниями и ее решение:

Дано:

S – множество состояний среды;

Игра агента со средой:

инициализация стратегии $\pi_1(a | s)$ и состояние среды s_1 ;

для всех $t = 1, \dots, T, \dots$:

агент выбирает действие $a_t \sim \pi_t(a | s_t)$;

среда генерирует премию $r_{t+1} \sim p(r|a_t, s_t)$ и новое состояние $s_{t+1} \sim p(s|a_t, s_t)$;

агент корректирует стратегию $\pi_{t+1}(a | s)$;

Замечание. Критическую роль в данном типе задач играют функции ценности состояния и ценности действия, а также выгода использования действий агента, которые описаны в формулах 1.8, 1.9, 1.10 и 1.11.

$$R_t = r_{t+1} + r_{t+2} + \dots + r_{t+k} + \dots \quad (1.8)$$

$$R_t = \gamma r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{k-1} r_{t+k} + \dots, \text{ где} \quad (1.9)$$

$\gamma \in [0, 1]$ – коэффициент дисконтирования

Указание. В формуле 1.9 есть некоторая эвристика. Чем выше γ , тем агент более дальновидный.

$$V^\pi = E_\pi(R_t | s_t = s) = E_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right) = E_\pi(r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s), \quad (1.10)$$

где E_π – математическое ожидание при условии, что агент следует стратегии π .

$$Q^\pi(s, a) = E_\pi(R_t | s_t = s, a_t = a) = E_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right) \\ = E_\pi(r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a), \quad (1.11)$$

где E_π – математическое ожидание при условии, что агент следует стратегии π .

Алгоритмы решения задач в средах с состояниями.

Метод временных разностей $TD(0)$ и $TD(\lambda)$.

Идея. Ключевая идея методов временных разностей (TD) в обучении с подкреплением заключается в обновлении оценок ценности состояний и действий на основе разницы между предсказанием будущих наград и фактическими наградами, полученными агентом. TD(0) оценивает ценности с учетом только текущей награды и одного шага вперед, в то время как TD(λ) учитывает как текущие награды, так и долгосрочные последствия, учитывая вклад всех предыдущих состояний и действий на текущую оценку.

Описание алгоритма.

Сначала необходимо выбрать действие a_t и стали известны r_{t+1} (премия) и s_{t+1} (состояние среды) в момент времени $t+1$. Далее оценить $V^\pi(s)$ по формуле 1.10 экспоненциальным скользящим средним.

$$V(s_t) := V(s_t) + \alpha_t(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \quad (1.12)$$

Замечание. Если α_t уменьшается в формуле 1.12, и все s посещаются бесконечное количество раз, то $V(s_t) \rightarrow V^\pi(s)$, $t \rightarrow \infty$.

Указание. Чтобы иметь более надёжную оценку $V(s)$ или $Q(s, a)$ приближающуюся к дисконтированной выгоде R_t можно усреднять прошлые, а не будущие наблюдения, так как премии в момент t неизвестны. Асимптотически это приводит к тому же результату!

Плюсы метода временных разностей:

- *Способность к онлайн-обучению.* Методы временных разностей позволяют агенту обучаться в режиме реального времени, что особенно полезно, когда среда динамична и меняется с течением времени.
- *Баланс между исследованием и эксплуатацией.* TD-методы помогают агенту балансировать исследование новых действий и эксплуатацию наилучших известных действий, что важно в задачах с ограниченными ресурсами и неопределенностью.
- *Способность к обучению без модели.* TD-методы могут обучаться без полной модели среды, что делает их применимыми в сценариях, где модель среды неизвестна или сложно построить.
- *Эффективность.* TD-методы, включая TD(0) и TD(λ) вычислительно эффективны.

Минусы метода временных разностей:

- *Неустойчивость к шуму.* TD-методы могут быть чувствительны к случайным флуктуациям и шуму в данных, что может привести к нестабильности в обучении.
- *Необходимость настройки гиперпараметров.*
- *Сложности с большими пространствами состояний и действий.*

Метод SARSA(0) и SARSA(λ) (State–Action–Reward–State–Action).

Идея и описание алгоритма.

Идейно алгоритмы SARSA и метод временных разностей довольно похожи, однако имеются некоторые различия в работе самих алгоритмов.

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha_t(r_{t+1} + \gamma Q(s_t, a') - Q(s_t, a_t)) \quad (1.13)$$

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha_t(r_{t+1} + \gamma Q(s_t, a') - Q(s_t, a_t)) \quad (1.13)$$

Игра агента со средой:

Сначала необходимо инициализировать стратегию $\pi_1(a|s)$ и изначальное состояние среды.

Далее для всех $t = 1, \dots, T, \dots$:

агент выбирает действие $a_t \sim \pi_t(a | s_t)$ ($a_t = \operatorname{argmax}_a Q(s_t, a)$ – жадная стратегия, но возможны и другие);

среда генерирует премию $r_{t+1} \sim p(r|a_t, s_t)$ и новое состояние $s_{t+1} \sim p(s|a_t, s_t)$;

агент разыгрывает еще один шаг $a' \sim \pi_t(a | s_t)$;

корректируем стратегию агента по формуле 13.

Плюсы SARSA:

- *Обучение на основе непосредственного опыта.* SARSA обучается непосредственно на основе последовательности состояний, действий и наград, полученных агентом в процессе взаимодействия с средой.
- *Способность обучения с учетом последствий.* SARSA учитывает долгосрочные последствия действий, так как он учитывает будущие награды и стремится к устойчивым стратегиям.
- *Применимость к различным средам.* SARSA может быть применен к разнообразным задачам и средам, включая среды с дискретными или непрерывными пространствами состояний и действий.

Минусы SARSA:

- *Требуется много данных.* SARSA требует большого объема данных для обучения, особенно в сложных средах.
- *Чувствительность к выбору параметров.*
- *Сложность с большими пространствами состояний и действий.*

Метод Q-обучения.

Идея и описание алгоритма. Метод Q-обучения от метода SARSA отличается отсутствием предпоследнего шага алгоритма и замены формулы 1.13 на формулу 1.14 в последнем шаге.

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha_t(r_{t+1} + \gamma \max_a Q(s_t, a') - Q(s_t, a_t)) \quad (1.14)$$

Плюсы Q-обучения:

- *Модель-фри.* Q-обучение является моделью-фри методом, что означает, что агент обучается без явного знания о модели среды. Это делает его применимым в средах, где модель среды неизвестна или сложно построить.

- *Способность работать в больших пространствах состояний и действий.* Q-обучение может работать с большими пространствами состояний и действий, благодаря чему оно подходит для разнообразных задач.

Минусы Q-обучения:

- *Чувствительность к количеству состояний и действий.* С ростом количества состояний и действий в среде Q-обучение сталкивается с проблемой "проклятия размерности" и требует большого объема обучающих данных.
- *Требует большого количества обучающих данных.* Для сходимости к оптимальной стратегии Q-обучению требуется много данных, что может быть проблематично в некоторых сценариях.
- *Неустойчивость к шуму.* Q-обучение чувствительно к случайным флуктуациям и шуму в данных, что может вызвать нестабильность в обучении.

Вывод: Итак, задачи в среде со состояниями среды представляют собой важный класс задач в обучении с подкреплением, где агент взаимодействует с окружающей средой, чтобы оптимизировать свою стратегию и максимизировать награду, а среда может изменяться по мере изучения ее агентом (отличие от задачи о многоруком бандите). Ключевые элементы задачи здесь те же, что и в задаче о многоруком бандите с добавлением состояния среды. Для решения данного типа задач возможны следующие алгоритмы: Q-learning, SARSA, метод временных разностей и другие. Q-обучение (Q-learning) может быть хорошим выбором, когда среда статична, нет необходимости в учете долгосрочных последствий и данные доступны в больших объемах. SARSA может быть предпочтительным в средах, где долгосрочные зависимости могут повлиять на стратегию. Методы временных разностей (TD-методы), такие как TD(0) и TD(λ), могут быть применены в широком спектре сред, включая среды с неполной информацией и неполностью известными моделями.

Глава II. Практическая часть

Класс *GameRenderer*. Создание объектов и физики игры Air hockey

Перед тем, как приступить к обучению с подкреплением, мы поставили задачу визуализации игрового поля и стандартных объектов воздушного хоккея, а также добавления физики в игру. Нами была использована библиотека SFML — бесплатная и открытая кроссплатформенная мультимедийная библиотека, написанная на C++. Она предназначена для разработки мультимедийных приложений и игр.

В первую очередь, мы ввели константы для определения параметров игры (ширина, высота окна, высота шайбы, ширина и высота слайдеров, скорость шайбы, ширина ворот и другие). Вся визуализация, а также элементы физики игры в аэрохоккей были написаны в методах класса *GameRenderer*. Рассмотрим данный класс подробнее.

В защищенные поля класса *GameRenderer* были помещены:

- *sf::RenderWindow& window* - ссылка на объект окна SFML, используемого для отображения графики.
- *sf::RectangleShape field* – игровое поле.
- *sf::RectangleShape midLine* - срединная линия поля.
- *sf::CircleShape centerCircle* - окружность в центре поля.
- *sf::CircleShape centerDot* - точка в центре поля.
- *sf::CircleShape lowerSemiCircle* и *sf::CircleShape upperSemiCircle* - полуокружности у ворот.
- *sf::RectangleShape blueSlider* и *sf::RectangleShape redSlider* - игровые слайдеры для синей и красной сторон соответственно.
- *sf::RectangleShape puck* - шайба.
- *sf::RectangleShape ourGoal* и *sf::RectangleShape enemyGoal* - ворота для своей и вражеской стороны.
- *float blueSliderVelocity* - скорость перемещения синего слайдера.
- *float puckVelocity* - скорость перемещения шайбы.
- *sf::Vector2f puckDirection* - вектор направления движения шайбы.

Конструктор *GameRenderer* инициализирует параметры игрового поля (размер, цвет, положение элементов, таких как центральная окружность, линия посередине, полуокружности у ворот и т.д.) и устанавливает начальные параметры для слайдеров, шайбы и ворот, начальное направление и скорость движения шайбы, генерирует случайную начальную скорость синего слайдера (здесь была применена библиотека *random* и написана дополнительная функция *randomValue* вне класса).

Рассмотрим главные методы класса *GameRenderer* (в основном это функции, определяющие физику игры).

Примечание. В SFML при задании прямоугольных объектов мы знаем только их верхнюю левую точку (другие мы можем узнать при прибавлении к соответствующим координатам высоту или ширину объекта).

Итак, в методы класса *GameRenderer* были помещены:

- *isBorder*. Данная функция проверяет, находится ли шайба в пределах границы заданных ворот на поле, и возвращает булевское значение. Параметрами данного метода являются координаты точки и ворот.

Принцип работы функции показан на рисунке 2.1 (гол засчитан только тогда, когда шайба полностью влетела в ворота).

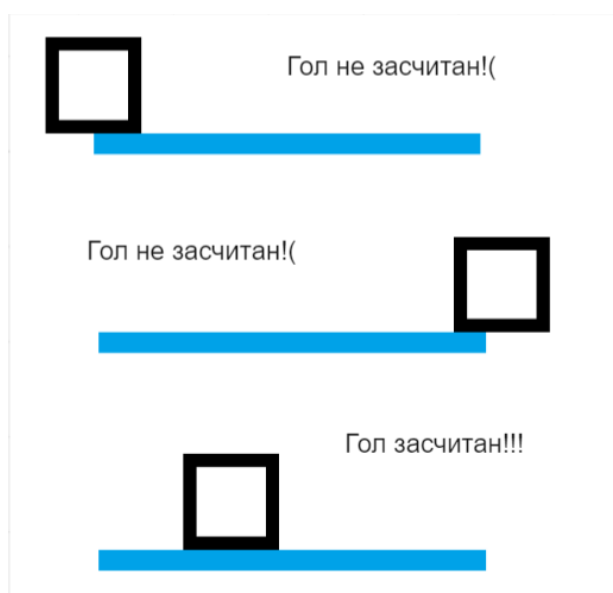


Рисунок 2.1. Случаи, когда гол будет засчитан / не засчитан.

- *isRedSlider*. Данная функция проверяет, произошло ли столкновение шайбы с красным слайдером сверху (нам необходима данная функция для перезапуска раунда с нейтральной премией). Использует хитрый метод для определения оси пересечения и направления столкновения.

Принцип работы *isRedSlider*:

- 1) Вычисляем центр шайбы и красного слайдера.
- 2) Вычисляем вектор расстояния между центрами шайбы и красного слайдера и проверяем пересекаются ли ограничивающие прямоугольники шайбы и слайдера.
- 3) Рассчитываем перекрытие по оси X (это расстояние между краем шайбы и краем слайдера по оси X) и перекрытие по оси Y (это расстояние между краем шайбы и краем слайдера по оси Y).
- 4) Если шайба находится в нижней половине поля, движется вниз (по положительной оси Y), и перекрытие по оси X больше, чем по оси Y, то мы фиксируем столкновение шайбы с красным слайдером.

- *isGoal*. Данная функция с помощью функции *isBorder* проверяет, произошло ли забрасывание шайбы в ворота с синей или красной стороны поля. С помощью данной функции далее мы будем выдавать максимальную / минимальную премию за попадание в наши или вражеские ворота.
- *resetRound*. Данная функция сбрасывает скорость шайбы, позиции шайбы и слайдеров в начальные значения и генерирует новую случайную скорость для синего слайдера. Фактически данная функция перезапускает раунд после попадания в ворота или в наш слайдер.
- *handlePuckSliderCollision* - улучшенный метод обработки столкновения шайбы со слайдером. Реализует точные вычисления столкновения и определение оси пересечения. Его принципом работы пользуется функция *isRedSlider*, однако данный метод учитывает не только попадания сверху, но и с других сторон, учитывая, что слайдер еще может и двигаться!

Принцип работы *handlePuckSliderCollision*:

- 1) Также как и в *isRedSlider* проверяем на пересечение ограничивающих прямоугольников шайбы и слайдера.
 - 2) Далее определяем ось пересечения, где произошло более глубокое перекрытие для обработки столкновения.
 - 3) Если перекрытие больше по оси X, то шайба отскакивает влево или вправо и происходит коррекция её положения. Если перекрытие больше по оси Y, то шайба отскакивает вверх или вниз и снова происходит коррекция.
- *update*. Цель данного метода – обновить положение синего слайдера, проверить столкновения шайбы с границами поля и слайдерами и обновить положение шайбы в соответствии с её направлением и скоростью.

Помимо методов определяющих физику игры в аэрохоккей, в классе *GameRenderer* были введены методы отрисовки элементов игрового поля, а также геттеры и сеттеры для скорости, чтобы управлять ей извне класса. Результат применения методов отрисовки можно увидеть на рисунке 2.2.

Примечание. Так как в SFML нет методов для рисования полуокружностей, нами было принято решение отрисовать окружности возле ворот и ограничить их игровым полем так, чтобы получились их половинки.

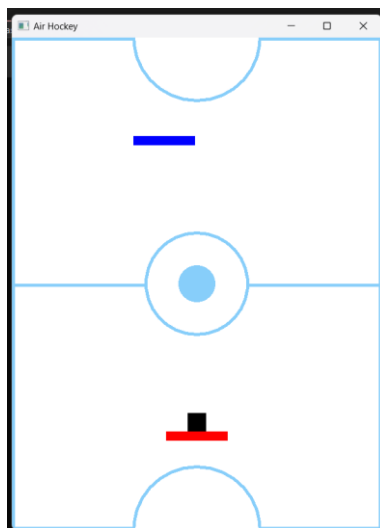


Рисунок 2.2. Визуализация игрового поля и объектов аэрохоккея.

Класс Agent. Создание агента для задачи обучения с подкреплением. Методы обучения агента

Теперь перейдем непосредственно к задаче обучения с подкреплением. Для этого определим новый класс *Agent*, где агентом будет красный слайдер, которому необходимо научиться выбирать наиболее оптимальные углы, чтобы попадать во вражеские ворота.

Из постановки задачи о многоруких бандитах, сформулированной в первой главе курсовой работы, нам необходимо в полях класса *Agent* указать вектора возможных действий, вероятностей выбора каждого из действий, количество каждого из выбранных действий, реакции агента на каждое действие (будет применяться для пересчета вектора средней премии) и самого вектора средней премии на каждое действие. Также в полях класса будет указан параметр типа *float*, для уменьшения / увеличения жадности той или иной стратегии, номер раунда для применения в *UCB* стратегии и указатель на функцию для выбора стратегии.

Теперь детально рассмотрим методы класса *Agent*.

- 1) Пересчет вектора средней премии на определенное действие (*QRecalc*).

Здесь, мы поступили хитро. Вместо того, чтобы каждый раз пересчитывать Q значение для определенного действия по формуле 1.1, мы воспользовались рекуррентной формулой 2.1, которая уменьшает асимптотическое время подсчета вектора средних премий.

$$Q_{t+1}(a) = (1 - \alpha_t)Q_t(a) + \alpha_t r_{t+1} = Q_t(a) + \alpha_t(r_{t+1} - Q_t(a)),$$

$$\text{где } \alpha_t = \frac{1}{k_t(a)+1}, \quad k_t(a) = \sum_{i=1}^t [a_i = a]. \quad (2.1)$$

Примечание. Так как у нас определен вектор количества выборов конкретного действия (*Counts*), то вместо подсчета $k_t(a)$ каждый раз, мы будем использовать значения из него.

- 2) Выбор конкретного действия на основе вероятностных распределений (*ChooseAction*).

В данном методе мы используем дискретное распределение случайных чисел на основе вероятностей, заданных в векторе *Probabilities*. Функция возвращает результат генерации случайного числа с использованием заданного дискретного распределения, таким образом, выбирая более оптимальное действие для получения максимальной премии в игре агента.

- 3) Функция для пересчета стратегии (*StrategyRecalc*). Она используется для применения конкретной стратегии агента.

Теперь перейдем к стратегиям обучения агента в задаче о многоруком бандите. Все они будут применяться в момент времени t . Это значит, что мы будем изменять вектор вероятностных распределений один раз за раунд, применяя соответствующие алгоритмы из теоретической части.

- Жадная стратегия (*Greedy*).

В данной стратегии мы выбираем максимальные из текущих средних премий (у вектора Q) ищем количество таких элементов и изменяем вектор вероятностей выбора действия (*Probabilities*) у тех действий, чья средняя премия максимальна в момент времени t (пересчет вектора вероятностей выбора действия был выполнен по формуле 1.4).

- Эпсилон-жадная стратегия (*EpsilonGreedy*).

Алгоритм этой стратегии отличается от жадной в корректировке вектора вероятностей выбора действия. Здесь мы добавляем к измененной вероятности выбора действия по жадной стратегии еще и некоторый параметр ϵ , деленный на количество всех действий, который мы можем менять, делая стратегию более или менее исследовательской / жадной (формула 1.5).

- Распределение Гиббса. *Softmax* стратегия (*Softmax*).

Алгоритм данной стратегии заключается в корректировке вероятности выбора действия в момент времени t (*Probabilities[i]*) путем деления экспоненциальной функции, зависящей от средней премии и параметра τ , на сумму других экспоненциальных функций, зависящих от остальных средних премий и данного параметра τ (формула 1.6).

- *Upper Confidence Bound (UCB)*.

Так как *Upper Confidence Bound* является «полужадной» из-за изменения вектора действий в момент времени t , то здесь нам необходимо применить жадную стратегию к скорректированному вектору действий A_t по формуле 1.7. Здесь необходимо выбрать максимальное из средних премий (из вектора Q), добавив некоторую поправку, которая позволяет выбирать те действия, которые мало выбирались до сих пор (подробнее в теоретической части).

Класс Game. Логика игры агента в Air Hockey

Теперь поговорим о логике игры агента в воздушный аэрохоккей, описанной в классе Game нашей задачи.

Класс *Game* представляет собой абстракцию для игрового процесса, включая в себя агента (*Agent*), окно для отображения игры (*sf::RenderWindow*), количество раундов (*num_rounds*), функцию для пересчета параметра (*parametr_reculc*), вектор для хранения средней награды (*AvgReward*), который понадобится для дальнейших модельных экспериментов, текущее действие (*curaction*), объект для рендеринга игры (*GameRenderer*), и часы (*sf::Clock*) для измерения времени.

Примечание. Так как по ходу игры мы можем уменьшать или увеличивать параметр в стратегиях, делая их более или менее жадными, нам необходимо ввести функции для пересчета параметра вне класса. Мы решили сначала выставить большое значение параметра (0.8), чтобы изучить все действия агента, и, спустя каждые 100 действий, уменьшать его на 0.05, делая стратегию более жадной.

Класс *Game* состоит из двух методов: *Play* и *GetAvgReward*.

- Метод *Play*.

Данный метод содержит в себе главный цикл игры, он и определяет логику агента. Исходя из постановки задачи, нам необходимо определить премии за действия агента. Мы приняли решение, что максимальную награду (1) агенту будем выдавать при попадании во вражеские ворота, а максимальный штраф (-1) при попадании в свои ворота. Также было принято решение выдавать нейтральную премию (0) при попадании в свой слайдер из-за того, что агент часто выигрывает. Логика выдачи премий можно увидеть на рисунке 2.3.

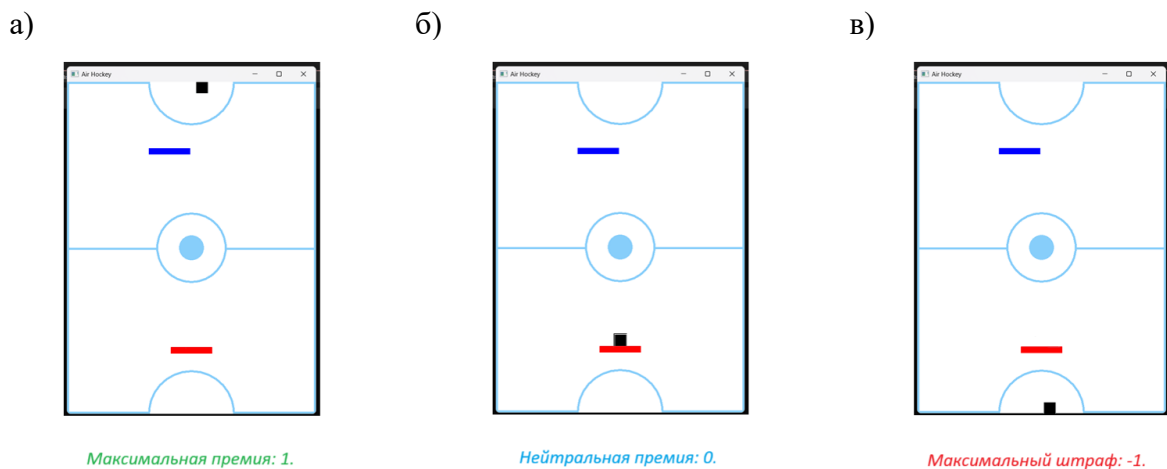


Рисунок 2.3. Логика выдачи премий агенту, где а) выдача максимальной премии, б) выдача нейтральной премии, в) выдача максимального штрафа.

Замечание. Класс *Game* является дружественным классом для класса *Agent*, так как в нем будут задействованы многие методы из последнего класса.

Примечание. В главном цикле игры мы вычисляли время прошедшее между двумя итерациями игры. С помощью него и скорости объекта мы можем указывать на какое расстояние передвинуть объект ($s = v \cdot t$).

Именно в методе *Play* мы проверяем забил ли агент гол или попал в себя. Если премия или штраф были выданы, то мы обновляем его текущее действие (*curaction*) с помощью метода *ChooseAction* класс *Agent*, вектор *Q* с помощью *QRecalc*, выбираем стратегию по которой агент будет обучаться и пересчитываем параметр, если количество превысило значение 100.

В данном методе также происходит расчет средней премии после выдачи награды для всех действий. Это необходимо для дальнейшего сравнения стратегий по мере обучения агента на основе графиков зависимости.

- Метод *GetRewards*.

Собственно, данный метод является геттером для извлечения вектора средних премий для каждого момента *t*. С помощью него мы организуем сравнение стратегий по мере обучения агента, построив графики зависимости на базе библиотеки *matplotlib* в *Google Colab*.

Класс *AgentStatistics*. Сравнение стратегий для обучения агента в воздушном хоккее. Эффективность стратегии с различным значением параметра

Итак, теперь нам остается проверить эффективность обучения агента для игры в аэрохоккей с применением стратегий, описанных в классе *Agent*. Для этого мы написали класс *AgentStatistics*, главная цель которого собирать статистику средней премии за все действия, полученной агентом в ходе игры за один раунд.

Итак, как именно мы будем получать эту статистику?

Ответ: Мы будем генерировать среду из 2000 задач — это 2000 разных игр по 500 раундов каждая. В каждой из этих 2000 игр будет задаваться какой-то свой агент с 10 разными действиями. То есть имеется 2000 массивов по 10 элементов каждый, которые могут быть различными, а могут быть одинаковыми. Соответственно 2000 различных распределений премии для каждого действия, которые обеспечиваются за счет случайного выбора скорости слайдера. Естественно, в каждой из игр на 500-ом раунде уже известны самые лучшие действия, но для первой игры это, например, угол 45, для второй игры угол 60 и т.д. Допустим в 500ом раунде первая игра выбрала действие 45 градусов и получила премию 1, вторая игра выбрала действие 60 градусов и получила премию 1, третья игра выбрала действие 15 градусов получила премию 0 и т.д. Теперь для каждого раунда необходимо сложить все 2000 премий и поделить на 2000. Так мы получим среднюю за 2000 игр премию для каждого раунда.

Теперь посмотрим на реализацию получения данной статистики (реализацию класса *AgentStatistics*).

Поля класса *AgentStatistics*.

- Двумерный массив *reward_histories* – вектор векторов, содержащий истории вознаграждений для каждого раунда и каждой игры.
- *agentType* – указатель на функцию, представляющую тип агента (используемую стратегию).
- *exp_parameter* – параметр экспоненциального распределения.
- *num_rounds* – количество раундов (периодов) в каждой игре.
- *num_games* – количество игр для сбора статистики.

Методы класса AgentStatistics.

- Метод *choose_actions*.

Данный метод генерирует вектор чисел от 0.05 до 0.95 с шагом 0.05 и перемешивает его. Далее он выбирает первые 10 элементов этого перемешанного вектора и умножает их на значение π , сохраняя результат в новом векторе. Таким образом мы получаем те самые 10 случайных направлений (10 случайных углов для направления удара агента), чтобы применить их в 500 раундах за 1 игру.

- Метод *GetAvgReward*.

Данный метод для каждой игры создает объект агента, используя функцию *choose_actions*, и объект игры. Он запускает игру и сохраняет историю вознаграждений в *reward_histories*. Затем вычисляет среднее значение вознаграждений по всем играм для каждого раунда и возвращает вектор средних значений вознаграждений, на значениях которых мы и будем строить графики зависимости.

Первое, что хочется проанализировать – это, как сильно влияет параметр на обучение агента при его различных значениях и выявить при каком значении параметра агент оптимальнее обучается забивать голы врагу.

Примечание. Для модельных экспериментов я буду использовать *Google Colab* и библиотеку *matplotlib*, с помощью которой буду изображать графики зависимости.

- Эпсилон-жадная стратегия.

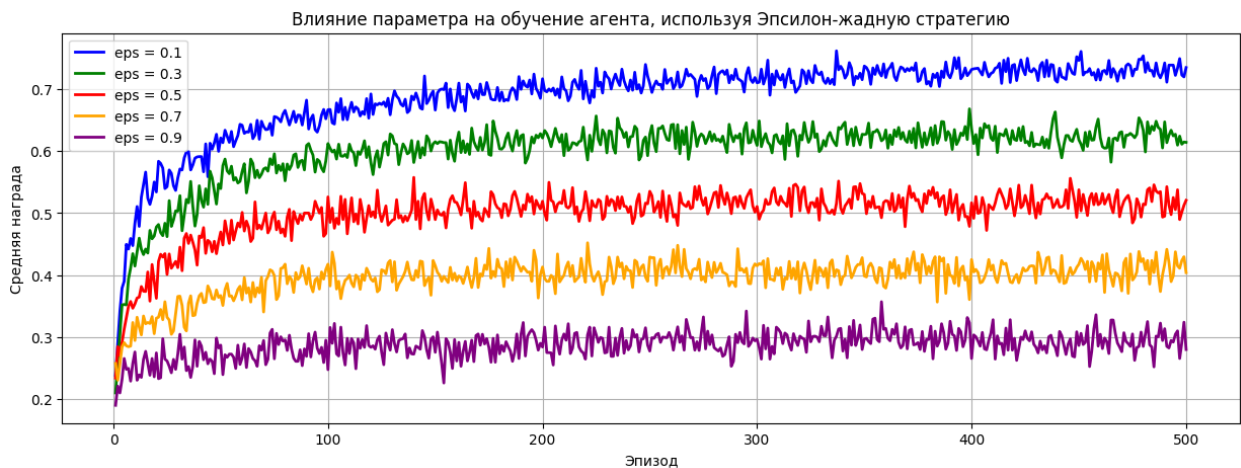


Рисунок 2.4. Влияние параметра на обучение агента, используя Эпсилон-жадную стратегию.

Вывод: Итак, протестировав обучение агента на 2000 играх, в каждой из которых происходит обучение агента на 500 итерациях с изменением параметра с 0.1 до 0.9 с шагом 0.2 (рисунок 2.4), можно сказать, что брать изначально большие значения параметра не имеет смысла, так как средняя премия от этого не увеличивается, а значит исследовательская тактика в данной стратегии нам не подходит. Наоборот, чем меньше значение параметра, тем больше и стабильнее наш агент получает среднюю премию за

выбор своего действия. По графику зависимости заметно, что оптимальным параметром ϵ для данной стратегии является $\epsilon = 0.1$.

- Распределение Гиббса. SoftMax стратегия.



Рисунок 2.5. Влияние параметра на обучение агента, используя SoftMax стратегию.

Вывод: В данной стратегии нет четких границ для значения параметра, нежели в Эпсилон-жадной стратегии, поэтому для интереса были взяты большие значения параметра. Протестировав поведение агента и проанализировав изменение средней премии, заметно, что исследовательские стратегии с большим значением τ хуже влияют на рост средней премии нежели жадные с меньшим значением параметра. Это обусловлено большой вероятностью выбора плохих действий, из-за которых агент не может получить большую среднюю премию. На графике зависимости (рисунок 2.5) видно, что оптимальным параметром является значение $\tau = 0.1$.

- Upper Confidence Bound (UCB).

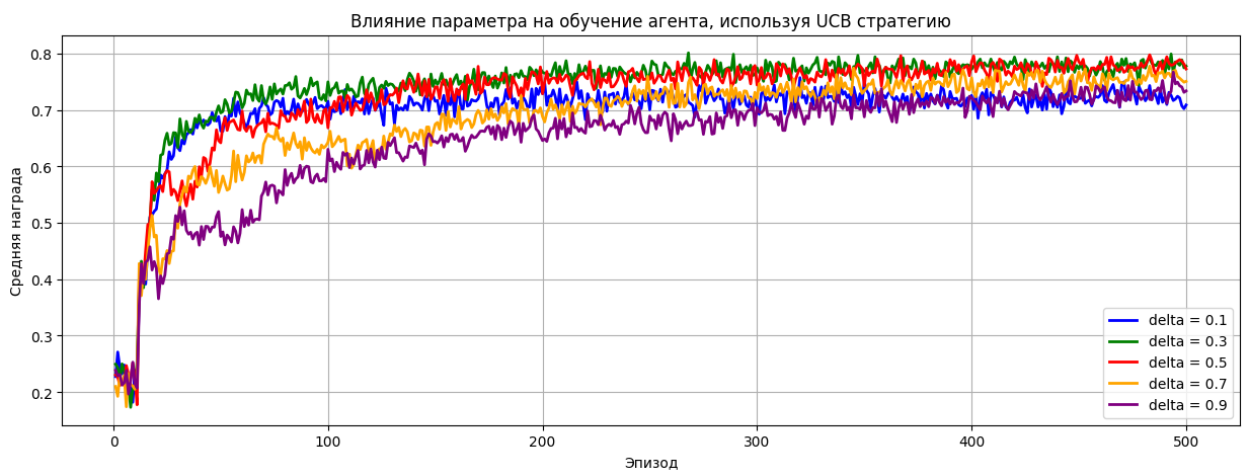


Рисунок 2.6. Влияние параметра на обучение агента, используя UCB стратегию.

Вывод: В данной стратегии были установлены пределы параметра от 0.1 до 0.9, как в Эпсилон-жадной стратегии. Заметно, что и в данной стратегии исследовательская тактика с малым значением параметра δ нам не улучшает получение большой средней премии за игру. На графике зависимости (рисунок 2.6) заметно, что оптимальным параметром для UCS стратегии является значение $\delta = 0.3$.

Теперь сравним стратегии обучения агента с лучшими подобранными параметрами с помощью графика зависимости.



Рисунок 2.7. Сравнение стратегий с лучшим значением параметра.

Вывод: Анализируя график зависимости (рисунок 2.7), мы понимаем, что в задаче обучения с подкреплением в воздушном хоккее стратегия Upper Confidence Bound (UCB) получилась наиболее оптимальной для обучения агента. Разбирая ее устройство, можно сделать вывод, что корректировка множества действий с максимальной текущей оценкой ценности для задачи аэрохоккея очень помогает для выбора лучших направлений для удара слайдером.

Заключение

Мы совместно с моим одногруппником провели исследование задачи обучения с подкреплением в аэрохоккее. В ходе анализа теоретической базы мы выявили, что данная задача сопоставима с типовыми задачами о многоруких бандитах, так как её суть заключается в выборе из множества возможных действий (направлений удара слайдера) с неизвестным распределением выигрышей для каждого действия. Мы определили стратегии обучения для агента в подобных сценариях, которые также были реализованы в практической части.

В процессе решения поставленной задачи мы приобрели опыт использования библиотеки SFML, которую использовали для визуализации объектов аэрохоккея и корректного отражения физики для шайбы. Проведя тестирование обучения агента на 2000 играх с 500 раундами, мы нашли оптимальные параметры компромисса для каждой стратегии (*exp/exp*): для стратегии *UCB* – 0.3, для *SoftMax* – 0.1, и для *EpsilonGreedy* – 0.1.

Полученные результаты позволяют сделать вывод о том, что в контексте аэрохоккея обучение с подкреплением не требует использования исследовательских тактик (стратегий с высоким значением параметра). Это подтверждается тем, что увеличение средней выигрышной премии можно добиться, выбирая меньшее количество действий. Наконец, при сравнении стратегий с оптимальными параметрами мы пришли к выводу, что наилучшей стратегией для обучения агента в данной задаче является стратегия *Upper Confidence Bound (UCB)*.

Список литературы

1. Sutton R. S. Reinforcement Learning: An Introduction [Текст] / Sutton R. S., Barto A. G. — 2-е. — London: The MIT Press, 2014, 2015 — 352 с.
2. Bertsekas D. P. Reinforcement Learning and Optimal Control [Текст] / Bertsekas D. P. — 3-е. — USA: Athena Scientific, 2019 — 388 с.
3. Simchowitiz M. Exploration and Incentives in Reinforcement Learning [Текст] / Simchowitiz M., Slivkins A. — 1-е. — : , 2021 — 49 с.
4. Воронцов К.В. Машинное обучение. Обучение с подкреплением. / Воронцов К.В. [Электронный ресурс] // Школа анализа данных, Яндекс : [сайт]. — URL: https://www.youtube.com/watch?v=iEUrX_eEWNY (дата обращения: 28.11.2023).
5. Lapan M. Deep Reinforcement Learning Hands-On [Текст] / Lapan M. — 2-е. — : Packt, 2020 — 826 с.

Приложения

Приложение 1

```
#define _USE_MATH_DEFINES
#include <cmath>
#include <math.h>
#include <ctime>
#include <random>
#include <vector>
#include <string>
#include <iostream>
#include <algorithm>
#include <numeric>
#include "SFML/Graphics.hpp"

template <class T>
std::ostream& operator<<(std::ostream& out, const std::vector<T>& s)
{
    out << "[ ";
    for (int i = 0; i < s.size(); i++)
        out << s[i] << ", ";
    out << "] ";
    return out;
}

// Константы для определения параметров игры
const int windowHeight = 600;
const int windowHeight = 800;
const float puckHeight = 30.0f;
const float sliderWidth = 100.0f;
const float sliderHeight = 15.0f;
```

```

const float puckSpeed = 10000.0f;
const float goalWidth = 200.0f; // Ширина ворот
const float borderWidth = 5.0f; // Ширина границы поля и разметки
const float circleRadius = 80.0f; // Радиус большой окружности

float randomValue(float left = 2500000.0f, float right = 15000000.0f) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<float> distribution(left, right);
    return distribution(gen);
}

class GameRenderer {
protected:
    sf::RenderWindow& window;
    sf::RectangleShape field;
    sf::RectangleShape midLine;
    sf::CircleShape centerCircle;
    sf::CircleShape centerDot;
    sf::CircleShape lowerSemiCircle; // Левая полуокружность у ворот
    sf::CircleShape upperSemiCircle; // Правая полуокружность у ворот
    sf::RectangleShape blueSlider;
    sf::RectangleShape redSlider;
    sf::RectangleShape puck;
    sf::RectangleShape ourGoal;
    sf::RectangleShape enemyGoal;
    float blueSliderVelocity;
    float puckVelocity;
    sf::Vector2f puckDirection;

public:

```

```

GameRenderer(sf::RenderWindow& window, float alpha) : window(window),
blueSliderVelocity(randomValue()),

puckVelocity(puckSpeed), puckDirection(cos(alpha), -sin(alpha)) {
// Инициализация игрового поля
field.setSize(sf::Vector2f(windowWidth - 2 * borderWidth, windowHeight - 2 * borderWidth));
field.setPosition(borderWidth, borderWidth);
field.setFillColor(sf::Color::White);
field.setOutlineThickness(borderWidth);
field.setOutlineColor(sf::Color(135, 206, 250)); // Light Sky Blue

// Инициализация линии посередине поля
midLine.setSize(sf::Vector2f(windowWidth - 2 * borderWidth, 5.0f));
midLine.setPosition(borderWidth, windowHeight / 2);
midLine.setFillColor(sf::Color(135, 206, 250)); // Light Sky Blue

centerCircle.setRadius(circleRadius);
centerCircle.setOutlineThickness(borderWidth); // Толщина линии, как у центральной линии
centerCircle.setOutlineColor(sf::Color(135, 206, 250)); // Цвет, как у центральной линии
centerCircle.setFillColor(sf::Color::White); // Прозрачное заполнение
centerCircle.setPosition(windowWidth / 2 - circleRadius, windowHeight / 2 - circleRadius);

centerDot.setRadius(puckHeight);
centerDot.setFillColor(sf::Color(135, 206, 250)); // Цвет, как у центральной линии
centerDot.setPosition(windowWidth / 2 - puckHeight, windowHeight / 2 - puckHeight);

float semiCircleRadius = goalWidth / 2; // Радиус полуокружности у ворот

// Левая полуокружность
lowerSemiCircle.setRadius(semiCircleRadius);
lowerSemiCircle.setOutlineThickness(borderWidth); // Толщина линии
lowerSemiCircle.setOutlineColor(sf::Color(135, 206, 250)); // Цвет линии
lowerSemiCircle.setFillColor(sf::Color::Transparent); // Прозрачное заполнение

```

```

lowerSemiCircle.setPosition(windowWidth / 2 - semiCircleRadius, -semiCircleRadius);

// Правая полуокружность
upperSemiCircle.setRadius(semiCircleRadius);
upperSemiCircle.setOutlineThickness(borderWidth); // Толщина линии
upperSemiCircle.setOutlineColor(sf::Color(135, 206, 250)); // Цвет линии
upperSemiCircle.setFillColor(sf::Color::Transparent); // Прозрачное заполнение
upperSemiCircle.setPosition(windowWidth / 2 - semiCircleRadius, windowHeight - semiCircleRadius);

// Инициализация слайдеров
blueSlider.setSize(sf::Vector2f(sliderWidth, sliderHeight));
blueSlider.setPosition(windowWidth / 2 - sliderWidth / 2, windowHeight / 5);
blueSlider.setFillColor(sf::Color::Blue);

redSlider.setSize(sf::Vector2f(sliderWidth, sliderHeight));
redSlider.setPosition(windowWidth / 2 - sliderWidth / 2, windowHeight * 4 / 5);
redSlider.setFillColor(sf::Color::Red);

// Инициализация шайбы
puck.setSize(sf::Vector2f(puckHeight, puckHeight));
puck.setPosition(windowWidth / 2 - puckHeight / 2, windowHeight * 4 / 5 - sliderHeight * 2);
puck.setFillColor(sf::Color::Black);

// Инициализация ворот
ourGoal.setSize(sf::Vector2f(goalWidth, borderWidth));
ourGoal.setFillColor(sf::Color::White);
enemyGoal.setSize(sf::Vector2f(goalWidth, borderWidth));
enemyGoal.setFillColor(sf::Color::White);
}

bool isBorder(float puckX, float puckY, float SliderX, float SliderY) {

```

```

return (SliderX <= puckX && puckX <= SliderX + goalWidth - puckHeight) &&
    (SliderY - puckHeight <= puckY && puckY <= SliderY + borderWidth);
}

bool isRedSlider(float deltaTime)
{
    sf::Vector2f puckCenter = puck.getPosition() + sf::Vector2f(puckHeight / 2, puckHeight / 2);
    sf::Vector2f sliderCenter = redSlider.getPosition() + sf::Vector2f(sliderWidth / 2, sliderHeight / 2);
    sf::Vector2f distance = puckCenter - sliderCenter;

    // Хитрая проверка пересечения
    if (std::abs(distance.x) < sliderWidth / 2 + puckHeight / 2 && std::abs(distance.y) < sliderHeight / 2 +
puckHeight / 2) {
        // Определяем ось пересечения (куда глубже вошли :P :о :3 8====D )
        float overlapX = sliderWidth / 2 + puckHeight / 2 - std::abs(distance.x);
        float overlapY = sliderHeight / 2 + puckHeight / 2 - std::abs(distance.y);

        return ((puck.getPosition().y > windowHeight / 2) && (puckDirection.y > 0) && (overlapX >
overlapY));
    }

    return false;
}

int isGoal() {
    if (isBorder(puck.getPosition().x, puck.getPosition().y,
        ourGoal.getPosition().x, ourGoal.getPosition().y))
        return -1;

    if (isBorder(puck.getPosition().x, puck.getPosition().y,
        enemyGoal.getPosition().x, enemyGoal.getPosition().y))
        return 1;

    return 0;
}

void resetRound() {

```



```

// Сброс позиции шайбы
puck.setPosition(windowWidth / 2 - puckHeight / 2, windowHeight * 4 / 5 - sliderHeight * 2);

// Сброс позиции слайдеров
blueSlider.setPosition(windowWidth / 2 - sliderWidth / 2, windowHeight / 5);
redSlider.setPosition(windowWidth / 2 - sliderWidth / 2, windowHeight * 4 / 5);

// Сброс скорости шайбы
puckVelocity = puckSpeed;
blueSliderVelocity = randomValue();
}

// Улучшенное определение столкновения шайбы со слайдером
void handlePuckSliderCollision(sf::RectangleShape& slider, float deltaTime) {
    sf::Vector2f puckCenter = puck.getPosition() + sf::Vector2f(puckHeight / 2, puckHeight / 2);
    sf::Vector2f sliderCenter = slider.getPosition() + sf::Vector2f(sliderWidth / 2, sliderHeight / 2);

    sf::Vector2f distance = puckCenter - sliderCenter;

    // Хитрая проверка пересечения
    if (std::abs(distance.x) < sliderWidth / 2 + puckHeight / 2 && std::abs(distance.y) < sliderHeight / 2 +
        puckHeight / 2) {

        float overlapX = sliderWidth / 2 + puckHeight / 2 - std::abs(distance.x);
        float overlapY = sliderHeight / 2 + puckHeight / 2 - std::abs(distance.y);

        if (overlapX > overlapY) {
            // Пересечение сверху или снизу
            puckDirection.y = -puckDirection.y;

            // Выталкивание в противоположном направлении
            float correction = (distance.y > 0) ? overlapY : -overlapY;
            puck.move(0, correction);
        }
    }
}

```

```

    }
    else {
        if (!((puck.getPosition().y < windowHeight / 2) && (puckDirection.x * blueSliderVelocity > 0)
            && (abs(puckDirection.x) * puckSpeed < abs(blueSliderVelocity)))) {
            //обработка врезания шайбы в синий слайдер сбоку при одном направлении
            puckDirection.x = -puckDirection.x;
        }
        // Выталкивание в противоположном направлении
        float correction = (distance.x > 0) ? overlapX : -overlapX;
        puck.move(correction, 0);
    }
}
}

void update(float deltaTime) {
    // Обновление синего слайдера
    if (blueSlider.getPosition().x <= borderWidth) {
        blueSliderVelocity = -blueSliderVelocity;
        blueSlider.setPosition(borderWidth + 1, blueSlider.getPosition().y); // Коррекция положения
        слайдера
    }
    else if (blueSlider.getPosition().x + sliderWidth >= windowWidth - borderWidth) {
        blueSliderVelocity = -blueSliderVelocity;
        blueSlider.setPosition(windowWidth - borderWidth - sliderWidth - 1, blueSlider.getPosition().y); //
        Коррекция положения слайдера
    }
    blueSlider.move(blueSliderVelocity * deltaTime, 0.0f);

    if (puck.getPosition().x <= borderWidth) {
        puckDirection.x = -1 * puckDirection.x;
        puck.setPosition(borderWidth + 1, puck.getPosition().y); // Сдвигаем шайбу внутрь поля
    }
}

```

```

else if (puck.getPosition().x + puckHeight >= windowWidth - borderWidth) {
    puckDirection.x = -1 * puckDirection.x;

    puck.setPosition(windowWidth - borderWidth - puckHeight - 1, puck.getPosition().y); // Сдвигаем
    шайбу внутрь поля
}

// Коррекция при столкновении с границами по оси Y
if (puck.getPosition().y <= borderWidth) {
    puckDirection.y = -1 * puckDirection.y;

    puck.setPosition(puck.getPosition().x, borderWidth + 1); // Сдвигаем шайбу внутрь поля
}

else if (puck.getPosition().y + puckHeight >= windowHeight - borderWidth) {
    puckDirection.y = -1 * puckDirection.y;

    puck.setPosition(puck.getPosition().x, windowHeight - borderWidth - puckHeight - 1); // Сдвигаем
    шайбу внутрь поля
}

// Улучшенная обработка соударения с синим слайдером
handlePuckSliderCollision(blueSlider, deltaTime);

// Улучшенная обработка соударения с красным слайдером
handlePuckSliderCollision(redSlider, deltaTime);

puck.move(puckVelocity * deltaTime * puckDirection.x, puckVelocity * deltaTime * puckDirection.y);
}

void drawField() {
    window.draw(field);

    window.draw(midLine); // Отрисовка линии посередине поля
    window.draw(centerCircle);
    window.draw(lowerSemiCircle);
    window.draw(upperSemiCircle);
}

```

```

        window.draw(centerDot);
    }

    void drawGoals() {
        enemyGoal.setPosition(windowWidth / 2 - goalWidth / 2, 0);
        window.draw(enemyGoal);
        ourGoal.setPosition(windowWidth / 2 - goalWidth / 2, windowHeight - borderWidth);
        window.draw(ourGoal);
    }

    void drawSliders() {
        window.draw(blueSlider);
        window.draw(redSlider);
    }

    void drawPuck() {
        window.draw(puck);
    }

    void render() {
        window.clear();
        drawField();
        drawGoals();
        drawSliders();
        drawPuck();
        window.display();
    }

    // Геттеры и сеттеры для скорости, если необходимо управлять скоростью извне класса
    void setBlueSliderVelocity(float velocity) {
        blueSliderVelocity = velocity;
    }

```

```

}

float getBlueSliderVelocity() const {
    return blueSliderVelocity;
}

void setPuckDirection(float alpha) {
    puckDirection = sf::Vector2f(cos(alpha), -sin(alpha));
}
};

class Agent {
protected:
    std::vector<float> Actions;
    std::vector<float> Probabilities;
    std::vector<float> Q;
    std::vector<float> Reactions;
    std::vector<int> Counts;
    float exp_parameter;
    int num_rounds;
    void (*agentType)(std::vector<float>, std::vector<float>&, std::vector<int>, float, int);

public:
    Agent(std::vector<float> Actions, void(*agentType)(std::vector<float>, std::vector<float>&,
std::vector<int>, float, int), float exp_parameter = 0.1f) : Actions(Actions), agentType(agentType),
exp_parameter(exp_parameter) {
        Probabilities = std::vector<float>(Actions.size(), 1.0 / Actions.size());
        Q = std::vector<float>(Actions.size());
        Reactions = std::vector<float>(Actions.size());
        Counts = std::vector<int>(Actions.size());
        num_rounds = 0;
    }
}

```

```

void QRecalc(int action) {
    Q[action] = Q[action] + 1.0 / (Counts[action] + 1) * (Reactions[action] - Q[action]);
}

int CooseAction() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.0, 1.0);

    std::discrete_distribution<> dist(Probabilities.begin(), Probabilities.end());

    return dist(gen);
}

virtual void StrategyRecalc() { // Функция для пересчета стратегии
    agentType(Q, Probabilities, Counts, exp_parameter, num_rounds);
}

friend void UCB(std::vector<float> Q, std::vector<float>& Probabilities, std::vector<int> Counts, float
exp_parameter, int num_rounds);

friend void SoftMax(std::vector<float> Q, std::vector<float>& Probabilities, std::vector<int> Counts,
float exp_parameter, int num_rounds);

friend void EpsilonGreedy(std::vector<float> Q, std::vector<float>& Probabilities, std::vector<int>
Counts, float exp_parameter, int num_rounds);

friend void Greedy(std::vector<float> Q, std::vector<float>& Probabilities, std::vector<int> Counts, float
exp_parameter, int num_rounds);

friend class Game;
};

void UCB(std::vector<float> Q, std::vector<float>& Probabilities, std::vector<int> Counts, float
exp_parameter, int num_rounds) {
    std::vector<float> A_t(Q.size());

```

```

for (int i = 0; i < Q.size(); i++)
    A_t[i] = (Q[i] + exp_parameter * std::sqrt(2.0 * std::log(num_rounds) / Counts[i]));

float max_elem = *max_element(A_t.begin(), A_t.end());

float cnt_max = std::count(A_t.begin(), A_t.end(), max_elem);

for (int i = 0; i < A_t.size(); i++)
    Probabilities[i] = 1.0 / cnt_max * (max_elem == A_t[i]);
};

void SoftMax(std::vector<float> Q, std::vector<float>& Probabilities, std::vector<int> Counts, float
exp_parameter, int num_rounds) {
    for (int i = 0; i < Q.size(); i++)
    {
        float sum = 0;
        for (int j = 0; j < Q.size(); j++)
            sum += (std::exp((1 / exp_parameter) * Q[j]));

        Probabilities[i] = (std::exp((1 / exp_parameter) * Q[i])) / sum;
    }
};

void EpsilonGreedy(std::vector<float> Q, std::vector<float>& Probabilities, std::vector<int> Counts, float
exp_parameter, int num_rounds) {
    float max_elem = *max_element(Q.begin(), Q.end());

    float cnt_max = std::count(Q.begin(), Q.end(), max_elem);

    for (int i = 0; i < Q.size(); i++)
        Probabilities[i] = ((1 - exp_parameter) / cnt_max * (max_elem == Q[i])) + (exp_parameter /
Probabilities.size());
};

```

```
};
```

```
void Greedy(std::vector<float> Q, std::vector<float>& Probabilities, std::vector<int> Counts, float  
exp_parameter, int num_rounds) {
```

```
    float max_elem = *max_element(Q.begin(), Q.end());
```

```
    float cnt_max = std::count(Q.begin(), Q.end(), max_elem);
```

```
    for (int i = 0; i < Q.size(); i++)
```

```
        Probabilities[i] = 1 / cnt_max * (max_elem == Q[i]);
```

```
};
```

```
float base_parametr_reculc(float parametr, int num_cycles) {
```

```
    return parametr;
```

```
}
```

```
float erasing_parametr_reculc(float parametr, int num_cycles) {
```

```
    if (num_cycles % 100 == 0 && num_cycles != 0 && parametr > 0.1f)
```

```
        return parametr - 0.05f;
```

```
    else
```

```
        return parametr;
```

```
}
```

```
class Game {
```

```
protected:
```

```
    Agent agent;
```

```
    sf::RenderWindow window;
```

```
    int num_rounds;
```

```
    float (*parametr_reculc)(float, int);
```

```
    std::vector<float> AvgReward;
```

```
    std::vector<float> rewards;
```

```
    int curaction;
```



```

GameRenderer renderer;

sf::Clock clock;

public:

    Game(Agent agent, int num_rounds = 2000, float (*parametr_reculc)(float, int) =
base_parametr_reculc) :

        agent(agent),        renderer(window,        agent.Actions[0]),        num_rounds(num_rounds),
parametr_reculc(parametr_reculc) {

        curaction = 0;

        window.create(sf::VideoMode(windowWidth, windowHeight), "Air Hockey");

    }

    void Play(int num_game = 0) {

        // Главный цикл игры
        while (window.isOpen()) {

            sf::Event event;

            while (window.pollEvent(event)) {

                if (event.type == sf::Event::Closed)

                    window.close();

            }

            if (agent.num_rounds >= num_rounds)

                window.close();

            // Вычисление прошедшего времени
            sf::Time elapsed = clock.restart();

            float deltaTime = elapsed.asSeconds();

            int flag = renderer.isGoal();

            int flag1 = renderer.isRedSlider(deltaTime);

            // Проверка на гол

```

```

if (flag || flag1) {

    agent.Reactions[curaction] = flag;
    agent.QRecalc(curaction);
    agent.Counts[curaction] += 1;

    agent.num_rounds++;

    agent.exp_parameter = parametr_reculc(agent.exp_parameter, agent.num_rounds);

    float sum = 0;

    for (int i = 0; i < agent.Q.size(); i++)
        sum += agent.Q[i] * agent.Counts[i];

    AvgReward.push_back(sum / 10);

    rewards.push_back(agent.Reactions[curaction]);

    agent.StrategyRecalc();

    curaction = agent.CooseAction();
    renderer.setPuckDirection(agent.Actions[curaction]);


    std::cout << "\nGame: " << num_game << "\nCycle: " << agent.num_rounds << "\nInterpreter: "
<< agent.exp_parameter

        << "\nCounts: " << agent.Counts << "\nProbabilities: " << agent.Probabilities
        << "\nQ: " << agent.Q << "\nReactions: " << agent.Reactions << "\n";

    renderer.resetRound(); // Перезапускаем раунд
}

```

```

        // Обновление игры
        renderer.update(deltaTime);

        // Рисование сцены
        renderer.render();
    }
}

std::vector<float> GetRewards() {
    return rewards;
}

};

class Agent_Statistics {
protected:
    std::vector<std::vector<float>> reward_histories;
    void (*agentType)(std::vector<float>, std::vector<float>&, std::vector<int>, float, int);
    float exp_parameter;
    int num_rounds;
    int num_games;

    std::vector<float> choose_actions() {
        std::random_device rd;
        std::mt19937 generator(rd());

        std::vector<float> numbers; // Вектор размером 18 (0.05 до 0.95 с шагом 0.05)

        // Используем std::iota для заполнения вектора
        for (float i = 0.05f; i < 1; i += 0.05f)
            numbers.push_back(i);
    }
};

```

```

// Перемешивание вектора
std::shuffle(numbers.begin(), numbers.end(), generator);

std::vector<float> ans;
for (int i = 0; i < 10; ++i)
    ans.push_back(numbers[i] * M_PI);

return ans;
}

public:
    Agent_Statistics(void (*agentType)(std::vector<float>, std::vector<float>&, std::vector<int>, float, int),
float exp_parameter,
    int num_rounds, int num_games) :
        agentType(agentType),          exp_parameter(exp_parameter),          num_rounds(num_rounds),
num_games(num_games) { }

std::vector<float> GetAvgReward() {

    for (int i = 0; i < num_games; i++) {

        std::vector<float> acts = choose_actions();

        Agent agent(acts, agentType, exp_parameter);
        Game game(agent, num_rounds);

        game.Play(i);
        reward_histories.push_back(game.GetRewards());
    }

    std::vector<float> ans;

```

```

    for (int i = 0; i < num_rounds; i++) {
        float sum = 0;
        for (int j = 0; j < num_games; j++)
            sum += reward_histories[j][i];
        ans.push_back(sum / num_games);
    }

    return ans;
}

};

int main() {

    std::vector<float> interpret;

    interpret.push_back(100.0f); interpret.push_back(50.0f); interpret.push_back(1.0f);
    interpret.push_back(0.5f); interpret.push_back(0.1f);

    for (int i = 0; i < interpret.size(); i++) {

        std::cout << interpret[i] << "\n";

        Agent_Statistics agent1(SoftMax, interpret[i], 500, 2000);
        std::vector<float> episode1_rewards = agent1.GetAvgReward();

        std::vector<float> episodes(episode1_rewards.size());
        std::iota(episodes.begin(), episodes.end(), 1);

        std::cout << episode1_rewards << "\n\n";
    }
}

```

```
    return 0;
}
```

Приложение 2

```
import matplotlib.pyplot as plt
import numpy as np
```

```
episode_reward_1 = [...]
episode_reward_2 = [...]
episode_reward_3 = [...]
episode_reward_4 = [...]
episode_reward_5 = [...]
```

```
episodes = np.arange(1, 500 + 1)
plt.figure(figsize=(15, 5))
```

```
plt.plot(episodes, episode_reward_1, label='delta = 0.1', linewidth=2, color='blue')
plt.plot(episodes, episode_reward_2, label='delta = 0.3', linewidth=2, color='green')
plt.plot(episodes, episode_reward_3, label='delta = 0.5', linewidth=2, color='red')
plt.plot(episodes, episode_reward_4, label='delta = 0.7', linewidth=2, color='orange')
plt.plot(episodes, episode_reward_5, label='delta = 0.9', linewidth=2, color='purple')
```

```
plt.title('Влияние параметра на обучение агента, используя ... стратегию')
plt.xlabel('Эпизод')
plt.ylabel('Средняя награда')
plt.legend()
plt.grid(True)
plt.show()
```