



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
«Применение обучения с подкреплением в воздушном хоккее»

Студент группы КМБО-03-22

Лазарев А.К.

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

Петрусович Д.А.

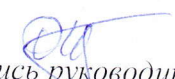
Работа представлена к
защите

«21» *сеп* 20 *23* г.


(подпись студента)

«Допущен к защите»

«21» *сеп* 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.А. Шатина А.В. Шатина

«22» сентября 2023 г.

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

Студент *Лазарев А.К.*

Группа *КМБО-03-22*

1. Тема: «Применение обучения с подкреплением в воздушном хоккее»

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.

 (Петрусеvич Д.А.)

Задание на курсовую
работу получил

«22» сентября 2023 г.

 (Лазарев А.К.)

Оглавление

ОГЛАВЛЕНИЕ	2
ГЛАВА I. ТЕОРИЯ	3
Общие понятия обучения с подкреплением	3
Алгоритмы обучения с подкреплением (общий случай)	4
Задача о многоруком бандите. Алгоритмы решения	6
Адаптивные стратегии	9
Выводы	10
ГЛАВА II. ПРАКТИКА	11
Постановка задачи и структура решения	11
Класс <i>GAMERENDERER</i>	12
Класс <i>AGENT</i> . Обучение с подкреплением	15
Класс <i>GAME</i> . Логика игры агента в воздушный хоккей	16
Класс <i>AGENT_STATISTICS</i> . Оценка качества стратегии агента	18
Результаты	19
ЗАКЛЮЧЕНИЕ	23
СПИСОК ЛИТЕРАТУРЫ	24
ПРИЛОЖЕНИЕ	25

Глава I. Теория

Общие понятия обучения с подкреплением

Обучение с подкреплением (Reinforcement Learning, RL) — это область машинного обучения, где агент обучается взаимодействовать со средой для максимизации некой награды. В отличие от обучения с учителем, в RL у агента нет явных примеров правильного поведения или сведений об окружающей среде, и он определяет оптимальную стратегию взаимодействия со средой на основе полученных наград. Такой метод обучения наиболее близок к человеческому, в котором не приходится заранее собирать и анализировать большое количество информации (обучающую выборку) для построения прогнозов. Вместо этого человек выбирает как ему лучше поступать, основываясь на собственном ранее полученном опыте.

В последние годы обучение с подкреплением сыскало большую популярность в виду универсальности своего математического аппарата. Его методы нашли применение в задача следующих прикладных задачах:

- Рекомендация новостных статей пользователям;
- Показ рекламы в Интернете;
- Управление технологическими процессами;
- Управление роботами;
- Управление ценами и ассортиментом в сетях продаж;
- Игра на бирже;
- Маршрутизация в телекоммуникационных сетях;
- Маршрутизация в беспроводных сенсорных сетях;
- Стратегические игры: шахматы, го, Dota2, StarCraft2...

Основными компонентами обучения с подкреплением являются:

- Агент: сущность, принимающая решения.
- Среда: всё, что окружает агента и на что он может воздействовать.
- Действие: взаимодействие агента со средой.
- Состояние: описание среды в данный момент времени.
- Награда: сигнал, который агент получает от среды после выполнения действия в определенном состоянии.
- Стратегия (политика): правило выбора действия агентом, приводящее к максимизации премии.
- Раунд: интервал, в которой происходит взаимодействие агента со средой.

Алгоритмы обучения с подкреплением (общий случай)

Алгоритмы обучения с подкреплением охватывают широкий спектр методов, которые агенты используют для обучения оптимальной стратегии взаимодействия с окружающей средой. Основная цель агента — максимизировать кумулятивную награду в течение времени.

Эти алгоритмы можно классифицировать в зависимости от того, используют ли они модель среды в процессе обучения:

1) Основанные на модели

В этих алгоритмах агент пытается узнать или аппроксимировать модель среды.

- **Динамическое программирование (Dynamic Programming):** это классический метод, который использует знание модели для нахождения оптимальной политики. Примеры включают в себя итерацию по политике (Policy Iteration) и итерацию по функции ценности (Value Iteration).
- **Планирование (Planning):** агент использует свою модель среды для имитации возможных будущих сценариев и принятия решений на основе этих сценариев. Например, Monte Carlo Tree Search (MCTS) использует случайные сэмплы для исследования пространства действий.
- **Адаптивное планирование:** при изменении среды агент может адаптировать свою модель и соответственно изменять свою политику. Примером такого алгоритма является метод временных разностей (Temporal-Difference, TD).

На основании тестов, предоставленных в книге Ричарда Саттона и Эндрю Барто “Reinforcement Learning: An Introduction” было выявлено, что такие алгоритмы могут быстро корректировать решения, однако с ростом числа вариаций действий агента или состояний среды время работы модельных алгоритмов многократно увеличивается. Это связано с высокой оценкой математической сложности каждого алгоритма, что делает такие методы невыгодными для агентов с большим числом действий.

2) Без модели

Агент не пытается явно узнать модель среды, но вместо этого напрямую оценивает ценность состояний или действий.

- **Q-learning:** оффлайн метод, который оценивает функцию ценности действия, не зависящую от текущей политики агента.
- **SARSA:** оценивает функцию ценности действия, основываясь на текущем и следующем действиях.
- **Policy Gradient Methods:** вместо оценки функции ценности агент напрямую оптимизирует свою политику с использованием градиентного восхождения.

- Actor-Critic Methods: комбинируют идеи из методов, основанных на политике, и методов оценки ценности, имея две компоненты: актер, выбирающий действия, и критик, оценивающий действия.

Такие алгоритмы отличаются своей устойчивостью к количеству действий агента и состояниям среды, так как вместо предсказаний используют уже набранную статистику. Хотя для накопления такой статистики требуется много времени, результатом обучения являются стабильные агенты с минимальным количеством ошибок в действиях.

Вывод

Выбор между алгоритмами, основанными на модели, и без модели зависит от специфики задачи, доступности данных, вычислительных ресурсов и других факторов. В то время как методы, основанные на модели, могут быть более эффективными в терминах количества требуемых взаимодействий со средой, поскольку они могут "думать вперед", методы без модели часто проще в реализации и могут быть более устойчивыми в сложных или непредсказуемых средах.

Задача о многоруком бандите. Алгоритмы решения

До текущего момента обучение с подкреплением рассматривалось как задача о среде с состояниями. То есть среда могла меняться и адаптироваться, реагируя на действия агента, что оказывало влияние на дальнейший выбор действий. Однако существует частный случай обучения с подкреплением, в котором среда является стационарной. Такая задача получила название задачи о многоруком бандите (Multi-Armed Bandit).

Постановка задачи:

Пусть A – множество всех возможных действий агента.

$p(r|a)$ – неизвестное распределение премии $r \in \mathbb{R}$ для действия $a \in A$.

$\pi_t(a)$ – стратегия агента в момент t для действия, распределенная на A .

Тогда игра агента со средой производится следующим образом:

- 1) Инициализируется стратегия $\pi_1(a)$;
- 2) Для всех раундов $t = 1, 2, \dots, T, \dots$;
 - 2.1) Агент выбирает действие согласно стратегии: $a_t \sim \pi_t(a)$;
 - 2.2) Среда в ответ на действие генерирует премию: $r_t \sim p(r|a_t)$;
 - 2.3) В зависимости от премии агент корректирует стратегию: $\pi_{t+1}(a)$.

Таким образом, после t раундов для каждого из действий $a \in A$ агентом может быть накоплена некоторая статистика, обозначающая среднюю премию, получаемую за действие a в t раундах. Метод подсчета этой статистики приведен в формуле (1).

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

Если устремить число раундов к бесконечности, получится узнать среднюю оценку премии для каждого действия. Тогда для максимизации выигрыша останется лишь выбрать действие с максимальной средней премией и все время использовать его. Однако провести бесконечное число раундов невозможно, из-за чего вводится абстрактное понятие ценности действия a , описанное формулой (2).

$$Q^*(a) = \lim_{t \rightarrow \infty} Q_t(a) \rightarrow \max \quad (2)$$

С учетом введенных понятий, задача о многоруком бандите сводится к быстрому предсказыванию величины Q^* для всех действий и выбора максимального значения.

То есть требуется сыграть минимальное количество раундов t , достаточное для построения оценки средней премии $Q_t(a)$ для каждого действия $a \in A$, с помощью которой удастся с как можно большей вероятностью “угадать” ценность $Q^*(a)$ каждого из действий и выбрать из них самое ценное.

Выбор стратегии.

Ключевым шагом в обучении с подкреплением является шаг 2.3) игры агента со средой. Интуитивно понятным методом выбора следующего действия является жадная стратегия

(Greedy Policy). В ней предпочтение отдается действиям, уже имеющим максимальную среднюю оценку премии, а другие действия удаляются из рассмотрения. Обновление множества действий производится по правилу, описанному в формуле (3).

$$A_t = \underset{a \in A}{\operatorname{Argmax}} Q_t(a) \quad (3)$$

Тогда вероятность выбора действия в момент времени t может осуществляться по формуле (4).

$$\pi_t(a) = \frac{1}{|A_t|} [a \in A_t] \quad (4)$$

Однако жадная стратегия содержит существенный недостаток: из рассмотрения при первой же возможности убираются действия, не давшие статистически значимых результатов на данный момент, но которые в последствии в среднем могли показать себя лучше оставленных действий. Очевидно, что для полноты обучения на каждом шаге следует оставлять вероятность выбора действия, до текущего момента не принесшего максимальной выгоды.

Для этого был введен принцип изучения–применения (Exploration–Exploitation), который заключается в переходе от равномерного выбора действия к жадному.

Простейшую реализацию этого принципа можно встретить в алгоритме эпсилон-жадной стратегии (Epsilon-Greedy Policy), приведенном в формуле (5).

$$\pi_t(a) = \frac{1 - \varepsilon}{|A_t|} [a \in A_t] + \frac{\varepsilon}{|A|}, \quad \varepsilon \in [0; 1] \quad (5)$$

Здесь при больших ε стратегия будет стремиться к равномерной (изучающей), а при малых, наоборот, к жадной (применяющей). Это позволит набрать необходимую статистику $Q_t(a)$ в начале обучения, а затем перейти к жадной стратегии, уменьшив параметр ε со временем.

Метод изучения-применения является ключевым не только в задаче о многоруком бандите, но и в задаче обучения с подкреплением в общем. Все описанные выше алгоритмы используют этот принцип в том или ином виде.

Остановливаясь подробнее на равномерно-жадных стратегиях для задачи о многоруком бандите, необходимо отметить идею, состоящую в том, что чем больше средняя оценка действия $Q_t(a)$, тем больше должна быть вероятность следующего выбора этого действия a (с поправкой на возможность исследования других действий). Таким образом появляется необходимость сопоставления множеству ценности действий Q_t множества вероятностей выбора.

SoftMax Policy (Распределение Гиббса) – более мягкий, нежели ε -жадная стратегия, вариант компромисса “изучение-применение”, включающее в себя вышеописанную идею: чем больше $Q_t(a)$, тем больше вероятность выбора действия a . Реализация приведена в формуле (6).

$$\pi_t(a) = \frac{\exp\left(\frac{1}{\tau} Q_t(a)\right)}{\sum_{b \in A} \exp\left(\frac{1}{\tau} Q_t(b)\right)}, \quad \tau \in (0; +\infty) \quad (6)$$

Параметром, отвечающим за баланс между изучением и применением, в данном случае, является параметр τ – температура, которая при стремлении к 0 делает стратегию более жадной, а при стремлении к бесконечности – равномерной, то есть чисто исследовательской. Для достижения наилучших результатов τ , как и ε в ε -жадной стратегии, можно уменьшать со временем.

Очевидно, что, пользуясь таким принципом, можно придумать большое количество стратегий, однако жадность алгоритма всегда будет позволять производить обучение быстрее. Хотелось бы иметь оптимальную жадную стратегию, умеющую адаптироваться от изучения к применению с минимальными потерями.

Возвращаясь к жадной стратегии, нельзя не отметить метод UCB, одну из самых популярных стратегий выбора следующего действия.

UCB Method (Upper Confidence Bound) – метод выбора следующего действия, использующий верхнюю оценку ценности действия, которая, в свою очередь, зависит как от размера средней премии $Q_t(a)$, так и от частоты использования этого действия. Чтобы набирать статистику для каждого действия, считается, что чем чаще выбирается действие, тем с меньшей вероятностью оно должно быть выбрано снова. Реализация метода приведена в формуле (7).

$$A_t = \underset{a \in A}{\operatorname{Argmax}} \left(Q_t(a) + \delta \sqrt{\frac{2 \ln t}{k_t(a)}} \right), \quad \text{где } k_t(a) = \sum_{i=1}^t [a_i = a] \quad (7)$$

Формула 7. Метод UCB выбора действий с максимальной верхней оценкой ценности. Параметр $k_t(a)$, стоящий в знаменателе отвечает за частоту выбора действия a . Параметр δ , как и упомянутые ранее τ и ε , является обязательным условием сохранения компромисса изучения-применения и так же может со временем уменьшаться от большего к меньшему, приводя стратегию от исследовательской к жадной. Очевидно, что коррекция стратегии будет осуществлена согласно формуле (4).

Все перечисленные алгоритмы имеют линейную оценку сложности. Из-за этого выбор оптимальной стратегии зависит от условий задачи, а именно от случайного распределения премий, генерируемого средой. Таким образом, определение лучшего алгоритма для конкретной задачи должно быть основано на экспериментах с различными алгоритмами и их параметрами изучения-применения.

Адаптивные стратегии

Главной проблемой всех описанных выше алгоритмов является долгое время пересчета функции средней ценности действия, которая по своей сути является средним арифметическим. Это ведет к квадратичному возрастанию времени обучения, что особенно плохо для задач с большим числом возможных действий.

Исправить не недостаток помогает экспоненциальное скользящее среднее, дающее, например, общую рекуррентную формулу вычисления среднего арифметического Q_t для корректировки стратегии (формула (8)).

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{k_t(a) + 1} (r_{t+1} - Q_t(a)), \quad \text{где } k_t(a) = \sum_{i=1}^t [a_i = a] \quad (8)$$

Экспоненциальное скользящее среднее может найти применение во многих вариациях задачи о многоруких бандитах. Например, если результаты SoftMax не показывают должного роста обучения, но задача подразумевает мягкий компромисс изучения-применения, можно попробовать “сгладить” подаваемые стратегии величины как разности средней премии с текущей. Этот метод получил название сравнения с подкреплением (Reinforcement Comparison). В результате обучения наибольшее предпочтение отдается действию, получающему в среднем максимальную премию. Реализация такого подхода приведена в формуле (9).

$$\begin{aligned} \bar{r}_{t+1} &= \bar{r}_t + \alpha(r_t - \bar{r}_t) - \text{средняя премия по всем действиям;} \\ p_{t+1}(a_t) &= p_t(a_t) + \beta(r_t - \bar{r}_t) - \text{предпочтения действий;} \\ \pi_{t+1}(a) &= \frac{\exp(p_{t+1}(a))}{\sum_{b \in A} \exp(p_{t+1}(b))} - \text{SoftMax - стратегия агента.} \end{aligned} \quad (9)$$

Однако применение экспоненциального скользящего среднего имеет еще одно важное свойство. С его помощью можно усреднять не только числовой временной ряд, но и временной ряд, состоящий из дискретных распределений. Таким образом можно оптимизировать жадную стратегию (формула 4), которая, например, применяется при использовании метода UCSB. Такая реализация получила название метод преследования (Pursuit). Его реализация приведена в формуле (10).

$$\pi_{t+1}(a) = \pi_t(a) + \beta \left(\frac{[a \in A_t]}{|A_t|} - \pi_t(a) \right) \quad (10)$$

Отмечу, что адаптивные стратегии существенно отличаются от полу-жадных тем, что на коэффициенты сглаживания не накладывается никаких ограничений, в отличие от коэффициентов изучения-применения, которые легко определяются постепенным уменьшением. Для эффективного подбора коэффициентов сглаживания, среда агента должна быстро отзываться на действия, а ее случайные распределения премий должны быстро предугадываться. Этот факт делает адаптивные стратегии конкурентноспособными в задачах простой среды и малого числа ручек.

Выводы

В этой главе были приведены общие сведения о таком методе машинного обучения как обучение с подкреплением. Была предложена классификация задач обучения с подкреплением на общие (имеющие динамические среды) и на задачи о многоруком бандите (имеющие статичные среды). В классе задач с динамическими средами были выделены самые известные алгоритмы стратегий, которые были сгруппированы по принципу необходимости модели среды в их реализации. Было дано подробное описание самым популярным эпсилон-жадным стратегиям агента в задаче о многоруком бандите, а также его адаптивным стратегиям.

К сожалению, к задаче обучения с подкреплением нет универсального подхода, и все приведенные алгоритмы невозможно сравнивать без условий конкретной задачи и опытов, проделанных в испытательной среде. Это связано с серьезными различиями в строении алгоритмов стратегий, которые подразумевают сведения о возможных действиях агента и о состояниях среды. Однако можно выделить принцип, по которым можно отбирать необходимые стратегии и классифицировать задачу обучения с подкреплением.

Если среда является динамической, то есть распределение премий для каждого действия агента может меняться, в зависимости от предшествующих действий, то следует определить, как программе будут подаваться сведения о среде. Если можно легко составить модель среды, отвечающую каждому действию агента, то оптимальными методами будут динамическое программирование, планирование и адаптивное планирование. Когда среду тяжело предсказать, следует использовать алгоритмы, не использующие модель, такие как Q-learning, SARSA, Policy Gradient Method, Actor-Critic Methods и так далее.

Если среда статична, то следует воспринимать задачу как задачу о многоруком бандите. В зависимости от числа действий, распределения премий и продолжительности раундов следует выбирать между жадными, например UCB, и мягкими, например SoftMax, стратегиями. При большом числе действий первый вариант предпочтительней, так как отсекает больше ненужных действий. Так же во внимание следует принять адаптивные стратегии, использующие экспоненциальное скользящее среднее для сглаживания результатов обработки, хотя такие стратегии редко конкурируют с эпсилон-жадными, так как имеют совершенно другой подход и используются в задачах, где можно легко подобрать коэффициенты сглаживания (в задачах, где хорошо предугадываются вероятностные распределения премий).

В виду того, что сложность алгоритмов обучения в динамических средах зачастую на порядок выше, чем в задачах о многоруком бандите, всегда следует оценивать возможность сведения среды к статичной.

Глава II. Практика

Постановка задачи и структура решения

Задача работы заключалась в разработке искусственного интеллекта на базе методов обучения с подкреплением для игры в воздушный хоккей. В целях проработки базовой составляющей обучения с подкреплением, было принято решение свести задачу к задаче о многоруком бандите.

Более подробная трактовка задачи заключается в следующем:

На игровом поле присутствуют два слайдера (синий – защищающий, красный – агент), за каждым из которых находятся ворота. В начальный момент времени рядом с красным слайдером так же находится шайба. Защищающий слайдер движется по горизонтали со случайной скоростью, меняя свое направление на противоположное при каждом соприкосновении с границей поля, а слайдер агента постоянно статичен. Агент может менять направление шайбы, то есть ручки бандита – различные углы направления полета шайбы. Требуется, используя методы обучения с подкреплением, определить такую траекторию шайбы, при которой число попаданий шайбы в ворота за синим слайдером (вражеские) будет максимальным.

Такая постановка удовлетворяет задаче о многоруком бандите по следующим причинам:

- 1) Генерация премий происходит за счет случайного выбора скорости слайдера из нормального распределения с нулевым матожиданием и единичной дисперсией. В итоге шайба либо успевает пролететь за синий слайдер, либо нет, что приводит к случайным исходам игры и, соответственно, к случайным премиям.
- 2) За счет статичного положения слайдера агента и возвращению синего слайдера на исходную позицию при перезапуске раунда среда остается статичной.

Таким образом задача сводится к построению среды, определению хода игры, реализации алгоритмов для решения задачи о многоруких бандитах и созданию пользовательского интерфейса по средствам объектно-ориентированного программирования.

Все решение можно структурно разделить на 4 класса: *GameRenderer*, *Agent*, *Game* и *Agent_Statistics*. Детальный разбор каждого из них приведен в работе далее.

Класс *GameRenderer*

Данный класс предназначен для создания объектов игры, определения ее физики и отрисовки ее процесса. Большая часть методов этого класса - методы библиотеки SFML – одной из самых популярных библиотек для визуализации на C++.

В защищенные поля класса *GameRenderer* входят:

- *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* – защищающий слайдер и слайдер-агент 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.

Большинство полей инициализируются заданными заранее константами, однако *blueSliderVelocity* и *puckDirection* требуют особого определения, так как могут меняться во время игры. Для этих переменных были написаны геттеры и сеттеры.

Скорость синего слайдера задается отдельной функцией *randomValue* вне класса, основанной на методе *std::uniform_real_distribution* библиотеки `<random>`, который обеспечивает получения желанной случайной скорости из нормального распределения.

Далее в классе определяется физика игры. Первым важным шагом было научить среду фиксировать коллизии объектов.

Метод *isGoal* определяет, произошло ли столкновение шайбы с воротами по средствам фиксации пересечения прямоугольных форм. Принцип работы функции показан на рисунке 2.1 (гол засчитан только тогда, когда шайба полностью влетела в ворота). Он так же помогает установить, в какие именно ворота пришелся гол, по координате шайбы относительно середины поля. Это полезно для дальнейшего назначения награды/штрафа.

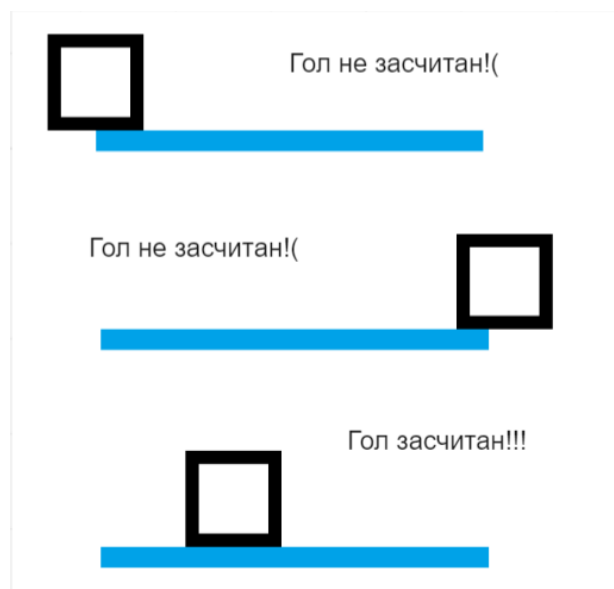


Рисунок 2.1. Случаи, когда гол будет засчитан / не засчитан.

Метод *handlePuckSliderCollision* используется для фиксации и обработки соударения шайбы и какого-либо из слайдеров методом глубины пересечения. В этом методе так же присутствует важная часть всей физики игры – коррекция положения шайбы. Так как среда будет тестироваться на высоких скоростях, велика вероятность, что за один кадр шайба будет проходить значительное расстояние, из-за чего она может застрять внутри слайдера и ее дальнейшее поведение не будет соответствовать ожиданиям. Эту проблему исправляет «коррекция», благодаря которой шайба в определенном кадре мгновенно выталкивается из шайбы, если за кадр она оказалась там, на минимальное расстояние. Так же корректно обрабатываются и ситуации, в которых синий слайдер въезжает в шайбу сбоку (в таком случае шайба не должна менять направления).

Метод *isRedSlider* работает по тому же принципу, что и предыдущий, однако нужен только для фиксации попадания шайбы в красный слайдер со стороны бьющей кромки. Совместно с методом *isGoal* он обеспечивает правильное отслеживание моментов завершения ранда и позволяет определить награду за последнюю сессию.

Метод *resetRound* возвращает в начальную позицию шайбу и синий слайдер и генерирует новую случайную скорость для синего слайдера.

Метод *update* предназначен для обработки изменений среды за один кадр процессорного времени (*deltaTime*). В нем происходит перемещение шайбы и синего слайдера по средствам метода *move* библиотеки SFML, который отлично вписывается в решение (зная скорость объектов по каждой из осей, а также время движения, легко определить расстояние, на которое нужно передвинуть объект). В этом методе также обрабатываются все возможные столкновения слайдера с границей и шайбы со слайдерами.

Методы, которые описаны в этом классе далее, такие как *drawField*, *drawGoals*, *drawSliders*, *drawPuck* и *render* отвечают за отрисовку всех элементов игры, включая декоративные. Результат применения методов отрисовки в начальный момент игры можно увидеть на рисунке 2.2.

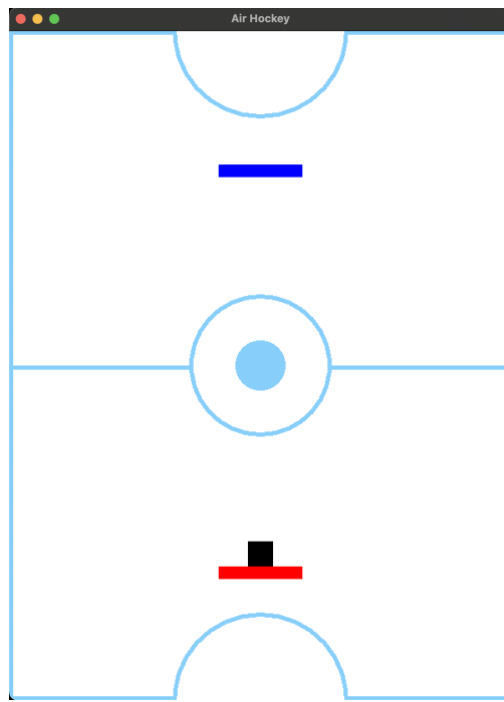


Рисунок 2.2. Визуализация игрового поля и объектов аэрохоккея.

Класс *Agent*. Обучение с подкреплением

Данный класс предназначен для обучения агента по средствам методов обучения с подкреплением в задаче о многоруком бандите. Класс тесно связан с *GameRenderer*, так как взаимодействует с полученными за раунд наградами и действиями агента (направлениями шайбы).

В защищенные поля класса *Agent* входят:

- *std::vector<float> Actions* – возможные действия агента;
- *std::vector<float> Probabilities* – вероятности выбора каждого действия;
- *std::vector<float> Q* – оценка средней награды для каждого действия;
- *std::vector<float> Reactions* – последние награды, полученные за каждое действие;
- *std::vector<int> Counts* – число выборов каждого действия;
- *float exp_parameter* – параметр изучения-применения;
- *int num_rounds* – счетчик уже сыгранных раундов;
- *void (*agentType)(std::vector<float>, std::vector<float>&, std::vector<int>, float, int)* – тип агента (указатель на функцию, содержащую стратегию выбора действия).

Все поля инициализируются в конструкторе, который обязательно должен принять вектор возможных действий агента и тип агента. Остальные векторы инициализируются нулями. Значение параметра исследования-применения по умолчанию 0.1, пользовательское значение может быть передано в конструктор.

Далее в коде представлена реализация алгоритмов, описанных в теоретической главе:

- Метод *QRecalc* реализует рекуррентный пересчет среднего арифметического награды за конкретное действие (формула (8)). Сложность такого пересчета $O(1)$.
- Метод *CooseAction* возвращает индекс выбираемого действия на основе вероятностей выбора каждого действия в массиве. Он работает на основе выбора величины из массива дискретных распределений *std::discrete_distribution*.
- Метод *StrategyRecalc* вызывает функцию типа агента, за счет которой пересчитывается вектор *Probabilities*.

Стратегии пересчитываются согласно алгоритмам для решения задач о многоруком бандите. Вне класса реализованы 4 основных алгоритма: *Greedy*, *EpsilonGreedy*, *SoftMax* и *UCB* (формулы (4), (5), (6) и (7) соответственно). Вектор самых выгодных действий в каждом из этих методов определен жадным выбором согласно формуле (3).

Для удобства работы с защищенными переменными класса *Agent* функции *Greedy*, *EpsilonGreedy*, *SoftMax* и *UCB*, а также класс *Geme* были объявлены дружественными.

Класс *Game*. Логика игры агента в воздушный хоккей

Имея статичную среду со случайным выделением премии и агента задачи многорукого бандита, можно построить их взаимодействие. Реализация этого приведена в классе *Game*, в котором агент играет в воздушный хоккей, попутно обучаясь согласно заданному алгоритму.

В защищенные поля этого класса входят:

- *Agent agent* – агент, взаимодействующий со средой;
- *sf::RenderWindow window* – окно, отображающее все, что происходит на игровом поле;
- *int num_rounds* – число раундов игры;
- *float (*parametr_reculc)(float, int)* – указатель на функцию, которая будет менять параметр изучения-применения с ходом игры;
- *std::vector<float> rewards* – вектор премий, выданных средой, за каждый сыгранный раунд;
- *int curaction* – текущее действие агента;
- *GameRenderer renderer* – игровая среда;
- *sf::Clock clock* – машинные часы, необходимы для отслеживания процессорного времени.

В классе присутствует главный метод *Play*, отвечающий поставленной выше задаче. Его работа заключается в следующем:

Основной цикл продолжается пока открыто созданное в конструкторе игровое окно. Он может завершиться в двух случаях: при закрытии вкладки окна напрямую пользователем либо при достижении необходимого числа сыгранных раундов.

Между каждой итерацией цикла засекается время, благодаря которому определяется положение объектов игры в каждом кадре (используется вызывается метод *update*). Также на каждой итерации проверяются условия перезапуска раунда: попадание в ворота или на верхнюю сторону красного слайдера. Если срабатывает один из методов *isGoal* или *isRedSlider*, значит раунд завершен. В таком случае в вектор *rewards* помещается награда за текущий раунд. Логика распределения премий продемонстрирована на рисунке 2.3.

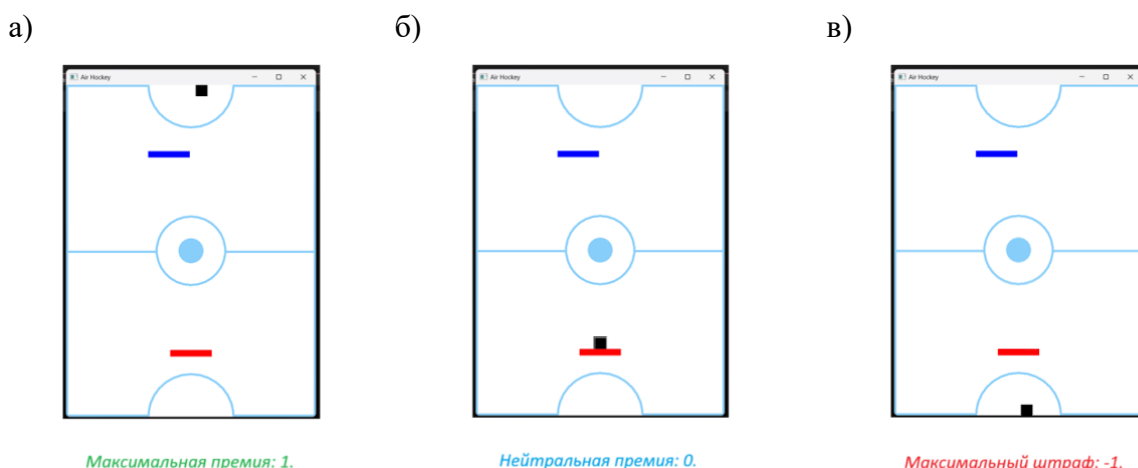


Рисунок 2.3. Логика выдачи премий агенту, где а) выдача максимальной премии, б) выдача нейтральной премии, в) выдача максимального штрафа.

Таким образом агент будет стремиться как можно чаще забивать и как можно реже допускать попадания шайбы в свои ворота. Также агент будет стремиться подбирать оптимальную по времени траекторию полета шайбы, чтобы та возвращалась с вражеской половины поля как можно реже.

После выдачи премии агент обновляет ключевые поля по средствам методов *QRecalc* и *StrategyRecalc* корректирует стратегию и начинается следующий раунд с новым выбранным действием. Важно отметить, что, если игра подразумевает переход от более исследовательской стратегии к жадной, функция пересчета параметра исследования-применения должна быть передана в конструктор класса. По умолчанию поле *parametr_reculc* задается функцией *base_parametr_reculc*, которая ничего не меняет, однако доступен вариант использования функции *erasing_parametr_reculc*, которая будет уменьшать значение параметра с 0.8 до 0.1 с шагом в 0.05 раз каждые 100 раундов.

После окончания игры с заданным количеством раундов имеем заполненный вектор наград в каждом из них. Он полезен для анализа прошедшей игры и оценки стратегии агента и поведения среды, что будет продемонстрировано в следующем классе. Для простого доступа к полю *rewards* написан геттер *GetRewards*.

Класс *Agent_Statistics*. Оценка качества стратегии агента

Для выявления оптимальной стратегии для игры в воздушный хоккей с заданными условиями был написан класс, реализующий тестовую среду.

Принцип его работы заключается в усреднении награды, полученной в множестве игр в каждом раунде.

В защищенные поля класса *Agent_Statistics* входят:

- *std::vector<std::vector<float>> reward_histories* вектор наград, хранящий награды за каждый раунд всех проведенных игр;
- *void (*agentType)(std::vector<float>, std::vector<float>&, std::vector<int>, float, int)* – указатель на функцию типа исследуемого агента;
- *float exp_parameter* – параметр изучения-применения исследуемого агента;
- *int num_rounds* – число раундов каждой игры, необходимых для сбора статистики;
- *int num_games* – число необходимых для сбора статистики игр.

Также в классе представлен защищенный метод *choose_actions*, в котором генерируются 18 различных углов от 0.05π до 0.95π , которые затем перемешиваются между собой по средствам метода *std::shuffle* библиотеки *random*. Из полученного множества получаются первые 10 элементов, что предоставляет возможность обучать агента на различных действиях в каждой игре.

Основным методом класса является метод *GetAvgReward*, возвращающий вектор из *num_rounds* значений премий, полученных за каждый раунд и усредненных по *num_games* играм. В его основном в цикле сначала создается агент на основе случайно выбранных 10 действий и заданного значения изучения-применения. С ним при помощи метода *Play* класса *Game* проводится игра с заданным числом раундов, после чего полученный при помощи метода *GetRewards* вектор наград добавляется в вектор историй наград *reward_histories*. Цикл повторяется для заданного числа игр.

По завершении основного цикла, награды в каждом раунде усредняются по заданному числу игр и записываются в результирующий вектор.

Примечание: так как отрисовка элементов игры в отдельном окне на каждом занимает много ресурсов процессора, при тестировании системы на больших скоростях возникает проблема туннелирования: за один кадр шайба проходит слишком большое расстояние и может пройти слайдер насквозь или вовсе улететь с поля, так как процессор долго обрабатывает изменения в коне и их отрисовку. При необходимом объеме раундов (для каждого агента в общей сложности играется 1000000 раундов) тестирование на низких скоростях невозможно. Из-за этого код класса *Game* был изменен, избавившись от необходимости отрисовки окна и его элементов. Таким образом тестирование стало проводиться вслепую, без наблюдения за тем, что происходит на поле.

Результаты

При помощи класса *Agent_Statistics* был проведен ряд тестов различных стратегий с различными значениями параметра изучения-применения с целью определения лучшего алгоритма решения задачи о многоруком бандите в условиях игры в воздушный хоккей. На основании статистических характеристик каждого из агентов при помощи библиотеки *matplotlibcpp* были построены графики зависимости средней за 2000 игр награды в каждом из 500 раундов (результат работы метода *GetAvgReward*) от номера раунда.

В первую очередь выявлялись оптимальные значения параметра изучения-применения.

Результирующее построение для оценки алгоритма *EpsilonGreedy* с различными значениями параметра ϵ приведены на рисунке 2.4. Приближенный вариант этого же графика приведен на рисунке 2.5.

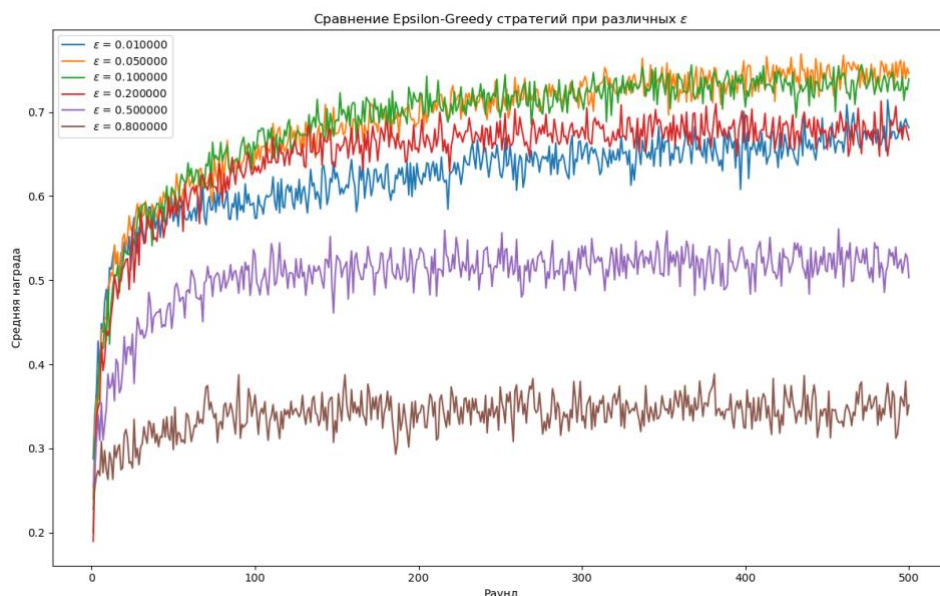


Рисунок 2.4. График средней награды при использовании эпсилон-жадной стратегии с различными значениями параметра ϵ .

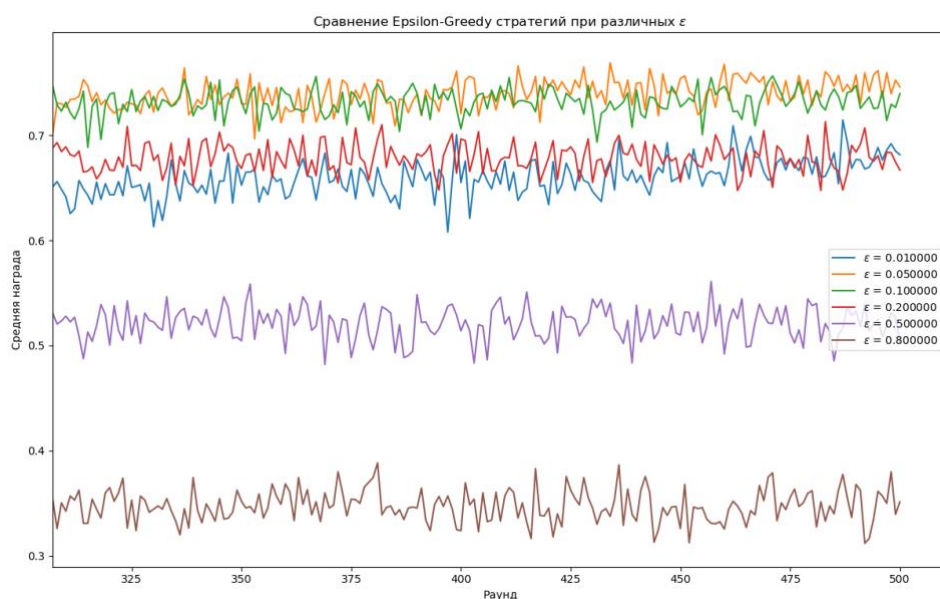


Рисунок 2.5. Приближенный график средней награды при использовании эпсилон-жадной стратегии с различными значениями параметра ϵ .

По данным графиков можно сделать следующий вывод: оптимальным параметром исследования-применения для эpsilon-жадной стратегии в игре в воздушный хоккей является $\epsilon = 0.05$. Хотя агент с таким параметром обучается немного медленнее, чем при $\epsilon = 0.1$, в последних 100 раундах он показывает стабильно большие результаты. Большие значения ϵ не эффективны, стратегия становится слишком исследовательской, из-за чего оптимальное действие выбирается реже. То же самое касается и слишком жадно стратегии с $\epsilon = 0.01$, в которой высока вероятность принятия ошибочного действия за оптимальное.

Результирующее построение для оценки алгоритма *SoftMax* с различными значениями параметра τ приведены на рисунке 2.6. Приближенный вариант этого же графика приведен на рисунке 2.7.

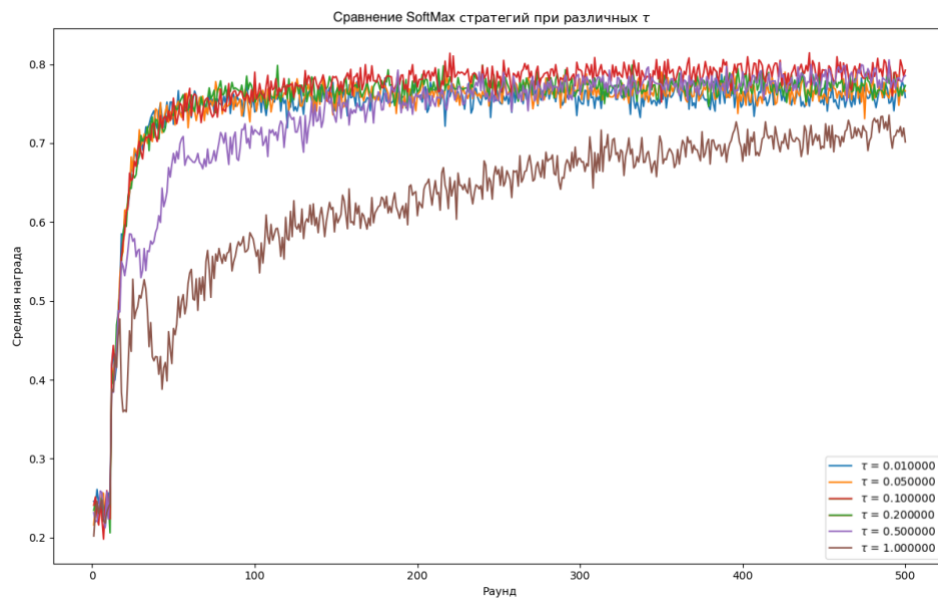


Рисунок 2.6. График средней награды при использовании *SoftMax* стратегии с различными значениями параметра ϵ .

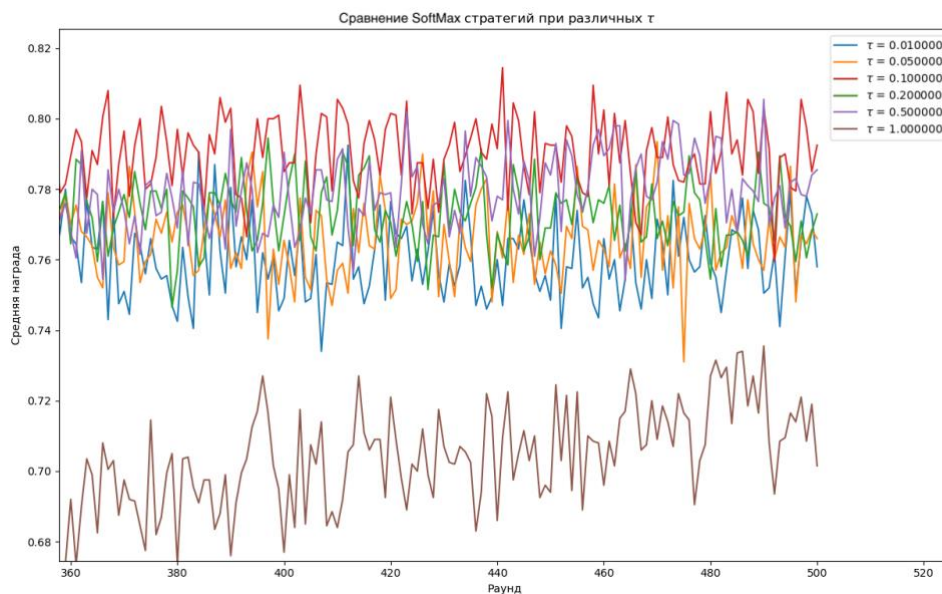


Рисунок 2.7. Приближенный график средней награды при использовании *SoftMax* стратегии с различными значениями параметра ϵ .

По данным графиков можно сделать следующий вывод: оптимальным параметром исследования-применения для *SoftMax* стратегии в игре в воздушный хоккей является $\tau=0.1$. Такой агент быстро обучается и стабильнее всех показывает лучшие результаты. Большие значения параметра не эффективны, так как агенты с такими τ обучаются заметно медленнее, чем оптимальный вариант.

Результирующее построение для оценки алгоритма *UCB* с различными значениями параметра δ приведены на рисунке 2.8. Приближенный вариант этого же графика приведен на рисунке 2.9.

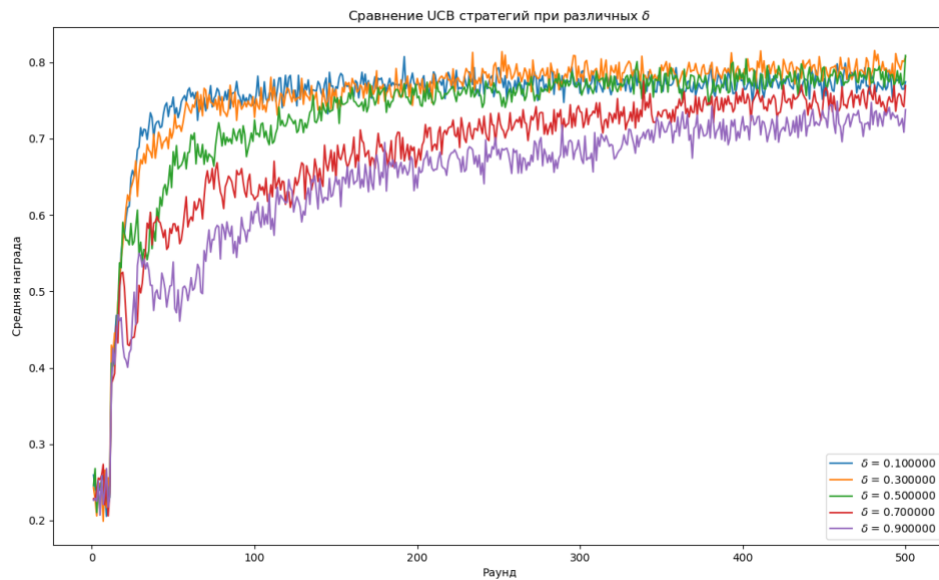


Рисунок 2.8. График средней награды при использовании *UCB* стратегии с различными значениями параметра δ .

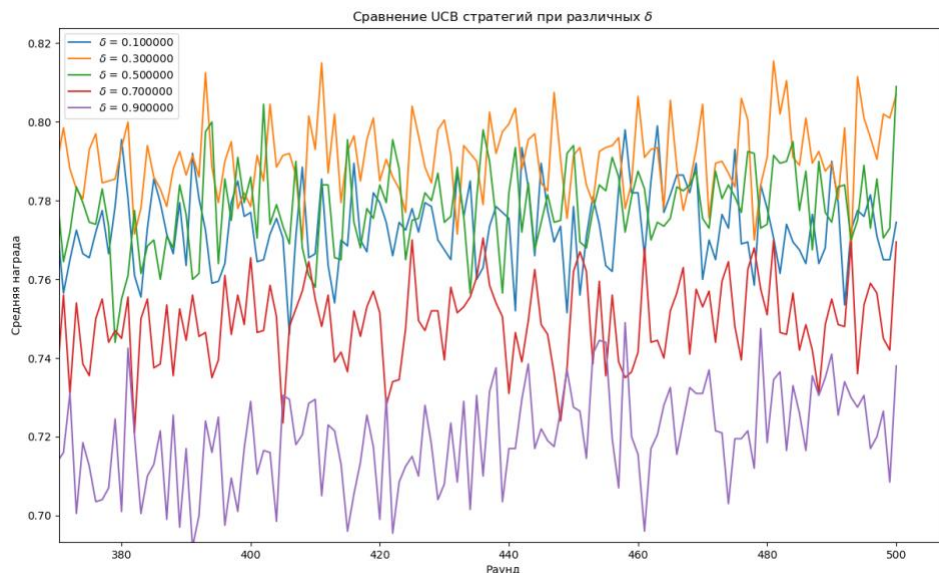


Рисунок 2.9. Приближенный график средней награды при использовании *UCB* стратегии с различными значениями параметра δ .

По данным графиков можно сделать следующий вывод: оптимальным параметром исследования-применения для *UCB* стратегии в игре в воздушный хоккей является $\delta=0.3$. Такой агент обучается немного медленнее, чем агент с $\delta=0.1$, но стабильнее всех

показывает лучшие результаты в последних 200 раундах. Большие значения параметра не эффективны, так как агенты с такими δ обучаются заметно медленнее, чем оптимальный вариант и не достигают высокого уровня.

После выяснения оптимальных значений параметров изучения-применения для определения лучшей стратегии для агента в игре в воздушный хоккей оставалось сравнить агентов с лучшими параметрами каждого алгоритма.

Результирующее построение для сравнения алгоритмов *UCB*, *SoftMax*, *Epsilon-Greedy* и *Greedy* с оптимальными значениями параметров *exploration-exploitation* приведены на рисунке 2.10.

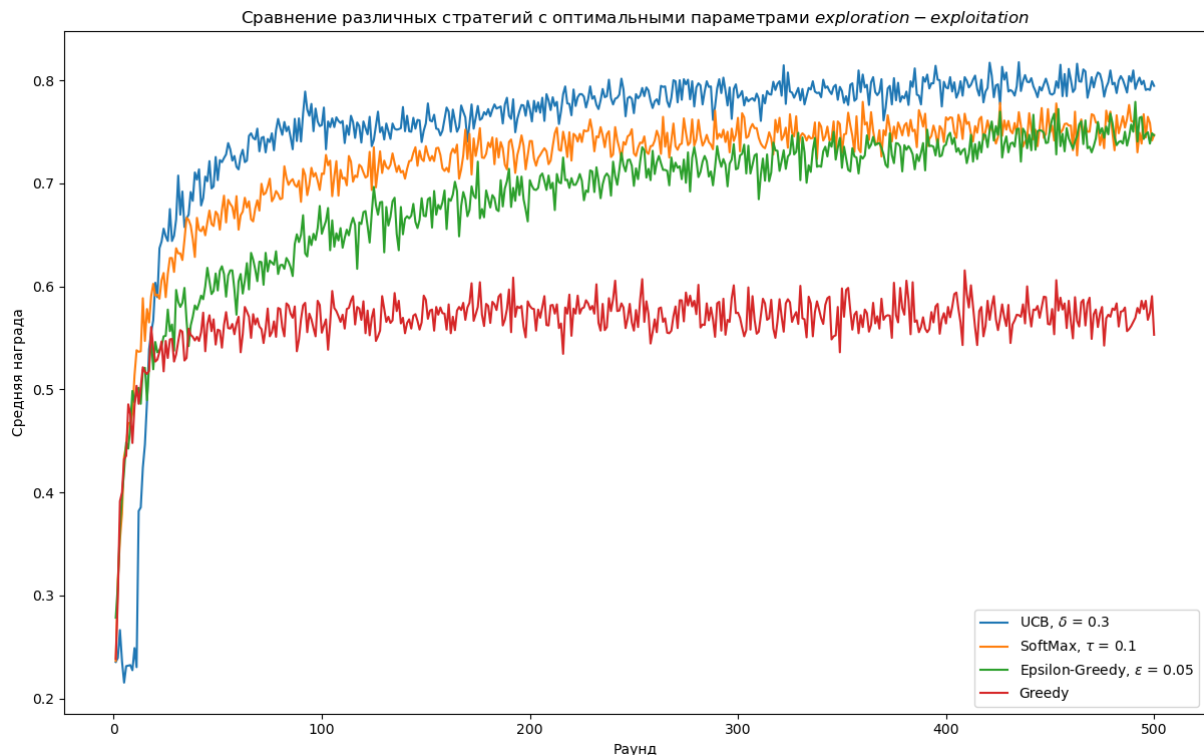


Рисунок 2.10. Сравнение различных алгоритмов задачи о многоруком бандите с оптимальными параметрами *exploration-exploitation*.

По данным графика становится очевидным, что оптимальной стратегией выбора следующего действия агента в воздушном хоккее как интерпретации задачи о многоруком бандите является алгоритм *UCB* с $\delta=0.3$. Это единственный из приведенных алгоритмов, который гарантированно опробует все действие перед выбором приоритетного. Вторым по результативности является алгоритм *SoftMax*, показывающий самый гладкий переход к стационарному состоянию агента. *Epsilon-Greedy* показывает отличные результаты роста, конкурирующие с *SoftMax* в последних раундах, однако времени на обучение эта стратегия требует слишком много, велика вероятность упустить выгоду. *Greedy* стратегия ожидаем показывает худшие результаты, зачастую принимая неоптимальные действия за лучшие.

В общем случае все графики показывают большой разброс между соседними раундами, что указывает на непредсказуемость поведения среды. Это указывает на правильную реализацию всех алгоритмов и их способность к адаптации в непредсказуемых условиях. Ни один агент не показывает результата ниже 0.5, что говорит от том, что в среде гораздо проще победить, нежели проиграть.

Код решения приведен в приложении.

Заключение

В работе приведены сведения о такой области машинного обучения, как обучение с подкреплением. Приведена трактовка задачи и основные методы решения ее общего случая. Детально разобран частный случай обучения с подкреплением – задача о многоруком бандите. Для нее расписана логика решения, а также даны подробные пояснения к самым популярным алгоритмам-стратегиям. Сделан вывод о классификации задачи: если можно легко составить модель среды, отвечающую каждому действию агента, то оптимальными методами будут динамическое программирование, планирование и адаптивное планирование. Когда среду тяжело предсказать, следует использовать алгоритмы, не использующие модель, такие как Q-learning, SARSA, Policy Gradient Method, Actor-Critic Methods и так далее. Если среда статична, то следует воспринимать задачу как задачу о многоруком бандите и выбирать оптимальное решение из стратегий *UCB*, *SoftMax*, *Epsilon-Greedy* или *Greedy*.

В практической части работы приведен вариант решения задачи обучения с подкреплением в игре в воздушный хоккей, сведенной к задаче о многоруком бандите. Сформированы 4 класса, решающих поставленную задачу: *GameRenderer* – отвечает за работу среды, *Agent* – реализует поведение агента задачи о многоруком бандите, *Game* – определяет логику взаимодействия агента со средой и *Agent_Statistics* – помогает собирать статистику об агентах в условиях поставленной задачи.

Для каждого агента построены графики оценки премии в каждом из 500 раундов, усредненные по 2000 игр. На их основе для каждой стратегии определено оптимальное значение параметра изучения-применения. Проведено сравнение всех стратегий с лучшими параметрами, по графику которого было заключено, что оптимальной стратегией выбора следующего действия агента в воздушном хоккее как интерпретации задачи о многоруком бандите является алгоритм *UCB* с $\delta=0.3$. Вторым по результативности является алгоритм *SoftMax*, показывающий самый гладкий переход к стационарному состоянию агента. *Epsilon-Greedy* показывает значительный рост, конкурирующий с *SoftMax* в последних раундах, однако времени на обучение эта стратегия требует слишком много и велика вероятность упустить выгоду. *Greedy* стратегия ожидаем показывает худшие результаты, зачастую принимая неоптимальные действия за лучшие.

В общем случае все графики показывают большой разброс между соседними раундами, что указывает на непредсказуемость поведения среды. Это указывает на правильную реализацию всех алгоритмов и их способность к адаптации в непредсказуемых условиях. Ни один агент не показывает результата ниже 0.5, что говорит о том, что в среде гораздо проще победить, нежели проиграть.

Список литературы

1. Richard, S. Sutton Reinforcement Learning: An Introduction / S. Sutton Richard, G. Barto Andrew. — London, England : MIT Press, 2018 — 552 с.
2. Lattimore T. Bandit Algorithms / T. Lattimore , C. Szepesvári. — 1-st edition. — United Kingdom: Cambridge University Press, 2020 — 536 с.
3. Szepesvári C. Algorithms for Reinforcement Learning / C. Szepesvári — 1-st edition. — United Kingdom: Morgan and Claypool Publishers;, 2010 — 104 с.
4. Bertsekas D. Reinforcement Learning and Optimal Control / D. Bertsekas — 1-st edition. — Massachusetts: Athena Scientific; 1-st edition, 2019 — 388 с.
5. Lapan M. Deep Reinforcement Learning Hands-On / M. Lapan — 1-st edition. — Moscow, Russia: Packt Publishing, 2018 — 546 с.

Приложение

```
#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"
#include "../matplotlibcpp.h"

namespace plt = matplotlibcpp;

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 windowWidth = 600;
const int windowHeight = 800;
const float puckHeight = 30.0f;
const float sliderWidth = 100.0f;
const float sliderHeight = 15.0f;
const float puckSpeed = 5000000.0f;
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 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
        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) {
        // Определяем ось пересечения
        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
внутри 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
внутри 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> 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");
        //window.create(sf::VideoMode(windowWidth, windowHeight), "Air Hockey", sf::Style::None);
    }

```

```

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)
            //break;
            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);

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

            agent.StrategyRecalc();

```

```

    curaction = agent.ChooseAction();
    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)

        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> episodes (500);
    std::iota(episodes.begin(), episodes.end(), 1);

    Agent_Statistics agent1(UCB, 0.3f, 500, 2000);
    std::vector<float> episode1_rewards = agent1.GetAvgReward();
    plt::plot(episodes, episode1_rewards, {"label", "UCB,  $\delta = 0.3$ "});

    Agent_Statistics agent2(SoftMax, 0.1f, 500, 2000);
    std::vector<float> episode2_rewards = agent2.GetAvgReward();
    plt::plot(episodes, episode2_rewards, {"label", "SoftMax,  $\tau = 0.1$ "});

    Agent_Statistics agent3(EpsilonGreedy, 0.05f, 500, 2000);
    std::vector<float> episode3_rewards = agent3.GetAvgReward();
    plt::plot(episodes, episode3_rewards, {"label", "Epsilon-Greedy,  $\epsilon = 0.05$ "});

    Agent_Statistics agent4(Greedy, 0, 500, 2000);
    std::vector<float> episode4_rewards = agent4.GetAvgReward();
    plt::plot(episodes, episode4_rewards, {"label", "Greedy"});

    plt::legend();
    plt::title("Сравнение различных стратегий с оптимальными параметрами  $\epsilon$ -exploration-exploitation");
    plt::xlabel("Раунд");
    plt::ylabel("Средняя награда");
    plt::show();

    return 0;
}

```