



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 игре на плоском дискретном поле»

Студент группы КМБО-01-23

*Исакин М.А.*

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

*Петрусович Д.А.*

Работа представлена к  
защите

«25» *ген* 2024 г.

*(подпись студента)*

«Допущен к защите»

«20» *ген* 2024 г.

*(подпись руководителя)*

МОСКВА — 2024



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

---

Институт искусственного интеллекта

Кафедра высшей математики

Утверждаю  
Заведующий  
кафедрой А.В.Шатина

«21» сентября 2024 г.

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

Студент *Исакин М.А.* Группа *КМБО-01-23*

**1. Тема: «Применение обучения с подкреплением в игре на плоском дискретном поле»**

**2. Исходные данные:**

Построить класс для модели реализации обучения с подкреплением (метод временных разностей и UCS, как минимум)

На игровом поле ячейки динамически окрашиваются определенными цветами (открывающиеся соседние с агентом ячейки), при этом, есть неизвестные распределения вероятности в зависимости от того, в ячейке какого цвета находится агент. Ячейки чёрного цвета запрещённые – за них даётся максимальный штраф. Максимальное поощрение даётся за выход с поля в противоположной части от стартового положения, небольшие разные поощрения – за ход на ячейку некоторого цвета (не чёрного, с приближением к конечной точке)


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 (строить переход через препятствие или выбрать ход в нужном направлении)  
Продemonстрировать изменение выигрыша агента со временем

**4. Срок представления к защите курсовой работы: до «21» декабря 2024 г.**

Задание на курсовую  
работу выдал

«21» сентября 2024 г.

 (Петрусеvич Д.А.)

Задание на курсовую  
работу получил

«21» сентября 2024 г.

 (Исакин М.А.)

## Оглавление

Введение .....	3
Глава 1 Обучение с подкреплением в игре на плоском дискретном поле .....	4
Основные концепции ОСП для игры "2048" .....	4
Среда и агент .....	4
Вознаграждения и штрафы .....	4
Алгоритмы .....	6
Алгоритмы на основе методов ценности .....	6
Алгоритмы на основе методов стратегии .....	8
Сравнения алгоритмов .....	11
Глава 2 Реализация обучения с подкреплением .....	13
Описание алгоритма работы программы .....	13
Структура проекта .....	13
Описание файла training.ipynb .....	15
Описание файла agent.py .....	18
Описание класса Agent .....	18
Описание класса ReplayBuffer .....	20
Описание файла model.py .....	22
Архитектура нейронной сети .....	22
Взаимодействие класса QNetwork с проектом .....	23
Описание game.cpp .....	24
Результаты обучения .....	28
Заключение .....	31
Список литературы .....	32
Приложение .....	33

## **Введение**

В данной работе рассмотрены алгоритмы обучения с подкреплением: Q-learning, SARSA, DQN, REINFORCE, Trust Region Policy Optimization (TRPO), Proximal Policy Optimization (PPO). В игре на плоском дискретном поле был реализован алгоритм DQN, также был проведен анализ обучения.

# **Глава 1 Обучение с подкреплением в игре на плоском дискретном поле**

## **Основные концепции ОСП для игры "2048"**

Обучение с подкреплением (Reinforcement Learning) — это раздел машинного обучения, в котором агент обучается взаимодействовать с окружающей средой, принимая решения, основанные на получении обратной связи в виде наград или штрафов. Под агентом здесь понимается виртуальная сущность (программа, модель), которая получает на вход текущее состояние среды, в которой она действует. Главная цель агента — научиться такой последовательности действий, которая максимизирует суммарную награду за время.

В качестве среды для реализации обучения с подкреплением я выбрал известную игру на плоском дискретном поле «2048».

### **Среда и агент**

Среда представляет из себя игровое поле «2048» в виде сетки 4 на 4, где каждая клетка содержит число (степень двойки) или пустое место (ноль). После каждого действия агента, среда:

- Сдвигает числа по правилам игры.
- Объединяет клетки с одинаковыми числами (например,  $2+2=4$ ).
- Добавляет новую плитку на случайную пустую клетку (обычно это 2 с вероятностью 90% или 4 с вероятностью 10%).

Игра заканчивается, если все клетки заполнены и невозможно выполнить ни одно движение.

Агент - сущность, которая взаимодействует со средой, выполняя действия и получая вознаграждения. Он стремится максимизировать свое общее вознаграждение, набирая как можно больше очков. Агент видит текущее состояние среды, представленное как  $4 \times 4$  матрица чисел. Это единственная доступная информация: агент не знает, где появятся новые плитки. Агент может выполнять одно из четырёх действий: сдвиг вправо, сдвиг влево, сдвиг вверх, сдвиг вниз.

### **Вознаграждения и штрафы**

Первой идеей было считать награду как сумму значений всех клеток на поле. Казалось, что этот счет и будет тем самым двигателем прогресса для агента, ведь по постепенному увеличению счета можно судить, правильные действия выбираются или нет, и именно это можно использовать в качестве награды. Оказалось, что нет. И причина здесь - в механике игры.

Допустим, у вас есть 2 клетки с одинаковыми значениями, стоящие рядом. Вы их схлопываете, но счет на доске не изменился. Потому что их значения по отдельности равняются значению новой клетки. То есть совершив правильное действие, агент не получит позитивного подкрепления и не узнает, что делать нужно именно так. Более того, после каждого действия заполняется новая случайная клетка, как я писал выше, значением 2 или 4. То есть какое бы действие ни совершил агент, он всегда будет получать в ответ значение, которое равняется [счет до шага + 2 или 4]. Очевидно, этой информации недостаточно, чтобы понимать, насколько хорошее действие агент выбрал. И именно из-за этого обучение практически не прогрессировало.

Поэтому награду пришлось реализовать по-другому. Сначала я попробовал выдавать ему не текущую сумму клеток на доске, а сумму схлопнувшихся клеток за все время с начала игры. Теперь у агента появился более надежный ориентир, и обучение пошло быстрее: агент видел, какие из его действий сильно увеличивали счет, а какие - нет. Но даже так обучение шло не настолько быстро, насколько хотелось бы, поэтому пришла мысль показывать еще более специфичный ориентир: выдавать в качестве награды не всю сумму схлопываний за все предыдущие шаги, а только сумму схлопнувшихся клеток на текущем шаге. И вот это уже позволило ему четко понимать, какие действия к каким результатам должны приводить, и существенно ускорить обучение.

Но тут кроется еще одна деталь, связанная с механикой игры. Награда за сдвиг в двух противоположных направлениях будет одинаковой, но доски окажутся в разных состояниях, и приведут к разным последствиям на следующих шагах. Для решения этой проблемы введем штрафы. В моем случае я решил каждый ход штрафовать агента за все клетки, которые сдвинулись (то есть изменили свое положение) после выбранного действия. И чем больше клеток было сдвинуто, тем выше был штраф. То есть если он накапливает крупные значения в правом нижнем углу, и продолжает делать ходы вправо или вниз, то штрафом для него на каждом шагу будут только значения новых клеток. Если же он вдруг решит сделать ход вверх или влево, то сдвинется не только новая клетка, но также сместятся и все те, которые концентрировались в правом нижнем углу, и штраф будет огромным. Со временем агент понял, что наибольшая награда получается не только когда он схлопывает более крупные клетки, но и когда он двигает наименьшее количество клеток - и это заставляет его придерживаться выбранной стратегии, и только в исключительных случаях делать шаги в “неприоритетных” направлениях - когда ожидаемая награда с учетом штрафов оказывается действительно выше.

## Алгоритмы

Алгоритмы обучения с подкреплением можно классифицировать на две основные группы в зависимости от их подхода к обучению: алгоритмы на основе методов ценности (Value-based methods) и алгоритмы на основе методов стратегии (Policy-based methods)

### Алгоритмы на основе методов ценности

Эти алгоритмы фокусируются на оценке функции ценности (value function), которая отражает ожидаемую совокупную награду от нахождения в определённом состоянии или выполнения определённого действия.

Основная цель — найти оптимальную стратегию (policy), опосредованно максимизируя функцию ценности.

Рассмотрим наиболее известные алгоритмы на основе методов ценности.

#### 1. *Q-learning*.

Q-learning — это базовый и широко используемый алгоритм обучения с подкреплением, основанный на методах ценности. Основная идея которого - обучение оптимальной функции  $Q(s, a)$  которая оценивает "качество" действия  $a$ , выполненного в состоянии  $s$ . Эта функция позволяет выбирать наилучшие действия, чтобы максимизировать суммарную награду.

Формула обновления функции:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)],$$
 где  $Q(s_t, a_t)$  - текущая оценка ценности действия  $a_t$  в состоянии  $s_t$ ;  $\alpha$  - скорость обучения (learning rate);  $r_t$  - мгновенная награда, полученная в результате действия;  $\gamma$  - коэффициент дисконтирования, учитывающий важность будущих наград;  $\max_{a'} Q(s_{t+1}, a')$  - оценка наилучшего возможного действия в следующем состоянии  $s_{t+1}$ .

Особенности:

- Off-policy алгоритм: выбор действий для обновления Q-функции не зависит от текущей стратегии (policy). Он обновляется на основе максимального значения, даже если текущее действие выбрано случайно.
- Подходит для дискретных пространств действий.
- Работает в табличной форме для небольших сред или использует аппроксимацию для больших.

#### 2. *SARSA (State-Action-Reward-State-Action)*

SARSA — это алгоритм, похожий на *Q-Learning*, но отличается способом обновления Q-функции. Обновление  $Q(s, a)$  происходит на основе действий, выбранных текущей стратегией. Это делает алгоритм зависимым от текущей стратегии (on-policy).



Формула обновления Q-функции:

$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$ , где  $Q(s_{t+1}, a_{t+1})$  - ценность действия  $a_{t+1}$ , выбранного стратегией в следующем состоянии  $s_{t+1}$ .

Особенности:

- On-policy алгоритм: Алгоритм обновляет Q-функцию на основе действий, выбранных текущей стратегией.
- Более устойчивый к неустойчивости поведения (например, при исследовании).
- Подходит для задач, где важно учитывать текущую стратегию, например, в робототехнике.

### 3. Deep Q-Network (DQN)

DQN — это расширение Q-Learning, которое использует нейронные сети для аппроксимации функции  $Q(s, a)$ . Этот алгоритм был разработан для работы в сложных средах с большими пространствами состояний, где невозможно использовать табличное представление. Вместо хранения таблицы  $Q(s, a)$ , нейронная сеть принимает на вход состояние  $s$  и предсказывает  $Q(s, a)$  для всех возможных действий  $a$ . Нейросеть обучается минимизировать ошибку между предсказанным  $Q(s, a)$  и целевым значением.

Формула целевой функции:

$L(\theta) = E \left[ \left( r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$ , где  $\theta$  - параметры основной сети;  $\theta^-$  - параметры целевой сети (target network), которая обновляется медленнее для стабильности обучения.

Ключевые техники DQN:

1. Experience Replay:
  - Переходы  $(s_t, a_t, r_t, s_{t+1})$  сохраняются в буфере и выбираются случайным образом для обучения.
  - Это снижает корреляцию между последовательными примерами.
2. Target Network:
  - Используется отдельная сеть для вычисления целевых значений, что стабилизирует обучение.
3. Аппроксимация Q-функции:
  - Нейронная сеть используется для обработки сложных состояний, таких как изображения или большие числовые пространства.

Особенности:

- Подходит для непрерывных пространств состояний.

- Может работать в сложных средах, таких как игры или робототехника.
- Требуется больше вычислительных ресурсов.

Таблица 1. Сравнение алгоритмов на основе методов ценности.

Характеристика	Q-Learning	SARSA	DQN
Тип алгоритма	Off-policy	On-policy	Off-policy
Функция ценности	Таблица или аппроксимация	Таблица или аппроксимация	Нейронная сеть
Пространство состояний	Дискретное	Дискретное	Дискретное (или с обработкой визуальных данных)
Сложность	Простая	Простая	Сложная (использует нейронные сети)
Проблемы	Исследование / эксплуатация	Зависимость от стратегии	Высокие вычислительные затраты, сложность настройки
Применение	Простые среды	Простые среды	Игры, визуальные задачи, сложные среды

## Алгоритмы на основе методов стратегии

Эти алгоритмы напрямую обучают стратегию (policy)  $\pi(a|s)$ , которая описывает вероятность выполнения действия  $a$  в состоянии  $s$ . Основная цель — оптимизировать параметры стратегии для максимизации ожидаемой совокупной награды.

### 1. REINFORCE

Основная идея:

REINFORCE — это простой алгоритм на основе градиентного спуска, где параметры стратегии ( $\theta$ ) обновляются, чтобы максимизировать ожидаемую награду.

Целевая функция (обучение стратегии):

Стратегия определяется через вероятностное распределение  $\pi_{\theta}(a|s)$ , которое задаёт вероятность выполнения действия  $a$  в состоянии  $s$ . Цель — максимизировать ожидаемую награду:

$J(\theta) = E_{\pi_{\theta}}[G_t]$ , где  $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$  — совокупная дисконтированная награда.

Правило обновления параметров:

Используя стохастический градиент, обновление параметров выполняется как:

$\theta \leftarrow \theta + \alpha \cdot \nabla_{\theta} J(\theta)$ , где градиент:  $\nabla_{\theta} J(\theta) = E_{\pi_{\theta}}[\nabla_{\theta} \log \pi_{\theta}(a_t|s_t) \cdot G_t]$

Преимущества:

- Простота реализации.
- Прямое обучение стратегии, подходит для непрерывных пространств действий.

Ограничения:

- Высокая дисперсия градиентов, что делает обучение медленным.
- Может игнорировать важность действий, если их награды не скорректированы относительно среднего значения.

## 2. Trust Region Policy Optimization (TRPO)

Основная идея:

TRPO улучшает REINFORCE, добавляя ограничения на обновление стратегии, чтобы избежать больших скачков, которые могут привести к дестабилизации обучения.

Целевая функция:

Оптимизация проводится по следующей задаче:

$$\max_{\theta} E_{s \sim \pi_{\theta_{old}}} \left[ \frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\theta_{old}}(s, a) \right], \text{ где } \frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)} - \text{отношение вероятностей новой и}$$

старой стратегии (обозначает степень изменения);  $A_{\theta_{old}}(s, a)$  - функция преимущества (Advantage function), оценивающая полезность действия относительно среднего.

Ограничение:

Чтобы избежать слишком сильного изменения стратегии, вводится ограничение на дивергенцию Кульбака-Лейблера ( $D_{KL}$ ):

$$E_{s \sim \pi_{\theta_{old}}} [D_{KL}(\pi_{\theta_{old}} || \pi_{\theta})] \leq \delta, \text{ где } \delta - \text{допустимый уровень изменения стратегии.}$$

Преимущества:

- Стабильность: Ограничение дивергенции предотвращает резкие изменения в стратегии.
- Хорошо подходит для сложных сред с непрерывными действиями.

Ограничения:

- Сложность вычисления: Решение требует второго порядка производных.
- Высокие вычислительные затраты.

## 3. Proximal Policy Optimization (PPO)

Основная идея:

PPO упрощает TRPO, заменяя сложные ограничения на дивергенцию на более простое и эффективное приближение.

Целевая функция:

PPO использует клипированную функцию, чтобы ограничить отношение вероятностей:

$$L^{CLIP}(\theta) = E[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)], \quad \text{где} \quad r_t(\theta) = \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} -$$

отношение вероятностей новой и старой стратегии;  $\epsilon$  - порог клипирования, который ограничивает изменение вероятности;  $A_t$  - функция преимущества.

Как работает клипирование:

Если изменение вероятности выходит за пределы  $[1 - \epsilon, 1 + \epsilon]$ , градиенты обнуляются.

Это предотвращает резкие скачки стратегии.

Преимущества:

Простота и эффективность: PPO легко реализовать, и оно стабильно обучается.

Широкое применение: Используется в задачах робототехники, играх и других сложных средах.

Ограничения:

Требуется настройка гиперпараметров ( $\epsilon$ , learning rate).

Не всегда оптимален в задачах, где точное ограничение дивергенции важно.

Таблица 2. Сравнение алгоритмов на основе методов стратегии

Характеристика	REINFORCE	TRPO	PPO
Целевая функция	Градиент стратегии	Градиент с ограничением	Градиент с клипированием
Ограничение изменений	Нет	$D_{KL} \leq \delta$	Клипирование вероятностей
Стабильность обучения	Низкая	Высокая	Высокая
Сложность вычислений	Низкая	Высокая	Средняя
Применение	Простые задачи	Сложные среды	Универсальное

## Сравнения алгоритмов

Для наглядности реализуем сравнение алгоритмов в виде таблицы и выберем наиболее подходящий для нашей задачи алгоритм.

Таблица 3. Сравнение алгоритмов обучения с подкреплением

Характеристика	REINFORCE	TRPO	PPO	Q-Learning	SARSA	DQN
Тип метода	Policy-based	Policy-based	Policy-based	Value-based	Value-based	Value-based
Обновления стратегии	Градиент стратегии	Градиент с ограничениям	Градиент с клипированием	Табличное $Q(s, a)$	Табличное $Q(s, a)$	Нейронная сеть для $Q(s, a)$
Стабильность обучения	Низкая	Высокая	Высокая	Низкая	Средняя	Высокая
Используемая функция	$\pi(a s)$	$\pi(a s)$	$\pi(a s)$	$Q(s, a)$	$Q(s, a)$	$Q(s, a)$
Целевая функция	Совокупная награда	Advantage Function	Advantage Function	Дисконтированная награда	Дисконтированная награда	Дисконтированная награда
Баланс исследование / использование	Задаётся вероятностями стратегии	Задаётся вероятностями стратегии	Задаётся вероятностями стратегии	$\epsilon$ -жадная стратегия	$\epsilon$ -жадная стратегия	$\epsilon$ -жадная стратегия
Работа с пространством состояний	Непрерывные и дискретные	Непрерывные и дискретные	Непрерывные и дискретные	Дискретные	Дискретные	Непрерывные и дискретные
Работа с пространством действий	Непрерывные и дискретные	Непрерывные и дискретные	Непрерывные и дискретные	Дискретные	Дискретные	Дискретные
Алгоритм обучения	On-policy	On-policy	On-policy	Off-policy	On-policy	Off-policy
Особенности	Прост, но высокая дисперсия градиента	Требуется вычисление дивергенции	Прост в реализации, стабилен	Обучает максимальное $Q(s, a)$	Учитывает текущее действие	Использует replay buffer и target network

Проблемы	Высокая дисперсия градиентов	Высокая сложность вычисления	Нужна настройка гиперпараметров	Неэффективны в больших пространствах	Зависимость от стратегии	Зависимость от нейронной сети
Скорость обучения	Низкая	Средняя	Высокая	Средняя	Низкая	Высокая
Применение	Простые задачи, обучения агентов	Сложные среды, задачи с ограничениями	Универсальное	Простые среды, табличные задачи	Простые среды, обучение в реальном времени	Игры, визуальные данные, сложные среды

Из всех алгоритмов наиболее подходящим для нашей задачи будет DQN. Он больше всего нацелен на обучение моделей в играх, и имеет наивысшую стабильность обучения среди value based алгоритмов. Важным свойством DQN является высокая скорость обучения, что также очень важно нас. Про особенности этого алгоритма:

- нейронная сеть имеет простую архитектуру с несколькими полносвязными слоями, что делает её достаточно эффективной для задач с небольшим количеством состояний и действий, таких как игра "2048".

- использование слоев нормализации (BatchNorm1d) помогает улучшить стабильность обучения, что особенно важно при обучении с подкреплением, где градиенты могут быть нестабильными.

- функция активации ReLU используется для введения нелинейности в сеть, что позволяет модели аппроксимировать более сложные зависимости между состояниями и действиями.

Выходной слой с action\_size нейронами: Выходной слой имеет 4 нейрона, что соответствует 4 возможным действиям в игре "2048" (вверх, вниз, влево, вправо).

## Глава 2 Реализация обучения с подкреплением

### Описание алгоритма работы программы

#### Инициализация:

- 1) Создается экземпляр класса Game, который представляет собой среду игры 2048.
- 2) Создается экземпляр класса Agent, который представляет собой агента, обучаемого с помощью глубокого Q-обучения.
- 3) Инициализируется буфер воспроизведения (Replay Buffer) для хранения опыта агента.

#### Обучение:

- 1) В цикле по эпизодам (играм) агент взаимодействует со средой:
  - 1.1) Агент выбирает действие (ход) на основе текущего состояния игры.
  - 1.2) Среда выполняет действие и возвращает новое состояние, награду и флаг окончания игры.
  - 1.3) Агент сохраняет опыт (состояние, действие, награда, новое состояние, флаг окончания) в буфере воспроизведения.
  - 1.4) Агент обучается на случайной выборке из буфера воспроизведения.
- 2) После каждого эпизода агент обновляет свои параметры и сохраняет лучшие результаты.

#### Визуализация и сохранение:

- 1) В процессе обучения отображаются графики, показывающие прогресс обучения (например, средний счет за последние 50 эпизодов).
- 2) Лучшие результаты и история игр сохраняются для последующего анализа.

### Структура проекта

Проект включает семь файлов, которые взаимодействуют друг с другом для реализации процесса обучения с подкреплением (Reinforcement Learning). Основные этапы работы над проектом можно разделить на две части: разработка библиотеки и её использование.

#### 1. Разработка библиотеки game.pyd

Для создания библиотеки game.pyd, реализующей игру "2048", используются три файла: CMakeLists.txt, game.cpp и setup.py:

- CMakeLists.txt: Настраивает сборку проекта и объединяет game.cpp и setup.py.
- game.cpp: Содержит реализацию логики игры "2048".
- setup.py: Используется для компиляции game.cpp в библиотеку game.pyd с помощью инструментов сборки, таких как pybind11 и CMake.

После компиляции создаётся библиотека `game.pyd`, которая предоставляет интерфейс для взаимодействия с игрой "2048".

## **2. Использование библиотеки `game.pyd`**

Для обучения агента используются следующие файлы:

- `training.ipynb`: Основной файл, управляющий процессом обучения. В нём реализован алгоритм Deep Q-Network (DQN), который обучает агента играть в "2048". В этом файле импортируются:

- `game.pyd`: Используется для взаимодействия с игрой "2048".
- `agent.py`: Содержит агента, который взаимодействует с игрой и обучается.

- `agent.py`:

- Импортирует модель QNetwork из файла `model.py`.
- Взаимодействует с `game.pyd` для выполнения действий в игре "2048".
- Обучает модель QNetwork на основе данных, полученных из игры.

- `model.py`: Содержит реализацию модели QNetwork, которая используется агентом для обучения. Модель представляет собой глубокую нейронную сеть с тремя скрытыми слоями и функцией активации ReLU.

## **3. Обучение модели через агента**

Процесс обучения с подкреплением управляется файлом `training.ipynb`:

- Агент из `agent.py` взаимодействует с `game.pyd` для выполнения действий в игре "2048".

- Агент собирает данные о состояниях игры, действиях и результатах (например, награды за действия).

- Эти данные используются для обучения модели QNetwork из файла `model.py`.

- После обучения агент становится более эффективным в игре "2048", что демонстрируется повышением его результативности.



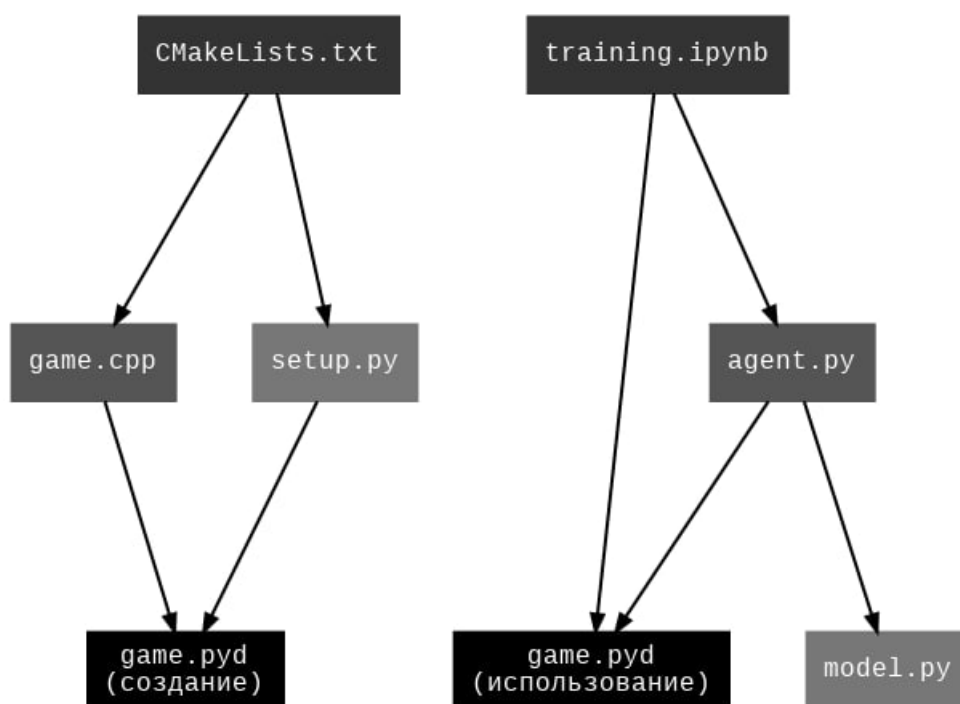


Рисунок 1. Схема взаимодействия файлов в проекте

Файлы `CMakeLists.txt`, `game.cpp` и `setup.py` создают библиотеку `game.pyd`. `training.ipynb` использует `agent.py` для управления и обучения модели и `game.pyd` для взаимодействия с игрой "2048". `agent.py` же взаимодействует с `model.py` для обучения модели и с `game.pyd` для выполнения действий в игре. Цель обучения — научить агента играть в "2048" на основе данных, полученных от среды, с использованием алгоритма Deep Q-Network (DQN).

### Описание файла `training.ipynb`

Файл `training.ipynb` представляет собой основной скрипт для обучения агента в игре '2048' с использованием алгоритма Deep Q-Learning. Этот файл содержит функцию `dqn`, которая управляет процессом обучения, а также функции для сохранения и загрузки состояния модели и истории игр.

#### 1. Импорт библиотек и модулей

Импортируются библиотеки для работы с массивами, построения графиков, работы с нейронными сетями, генерации случайных чисел, а также вспомогательные библиотеки для работы с файлами и отображения данных в Jupyter Notebook. Импортируются классы `Game` (из `game.cpp`) и `Agent` (из `agent.py`). Класс `Game` представляет собой среду игры '2048', а класс `Agent` — это агент, который обучается с помощью алгоритма Deep Q-Learning.

#### 2. Функция `transform_state`

Исходное состояние игры "2048" представляется в виде матрицы 4x4, где каждая ячейка содержит значение клетки. Для подачи этой матрицы в нейронную сеть её необходимо преобразовать в вектор 1x16. Однако, значения клеток имеют большой разброс (от 0 до сотен тысяч), что затрудняет обучение нейронной сети. Для решения этой проблемы используются различные методы преобразования состояния:

Логарифмическое масштабирование ( $\log_2$ ): Значения клеток преобразуются в логарифмическую шкалу, что позволяет привести их к более равномерному диапазону. Это делает обучение нейронной сети более стабильным, так как значения клеток становятся более предсказуемыми.

One-hot encoding: Каждая клетка кодируется как вектор с одной единицей, соответствующей её значению. Например, клетка со значением 2 будет представлена как вектор из 18 элементов, где второй элемент равен 1, а остальные — 0. Этот метод позволяет более точно представить состояние игры, но увеличивает размерность входных данных.

И так, функция `transform_state` преобразует состояние игры (матрицу 4x4) в одномерный вектор, который используется в качестве входных данных для нейронной сети.

Режимы преобразования:

`plain`: Просто преобразует матрицу в одномерный вектор.

`plain_hw`: Преобразует матрицу в одномерный вектор, а также добавляет транспонированную матрицу.

`log2`: Преобразует значения ячеек в логарифмическую шкалу (например,  $\log_2(\text{value})$ ), что помогает уменьшить диапазон значений и сделать обучение более стабильным.

`one_hot`: Преобразует состояние в one-hot encoding, где каждое значение ячейки кодируется как вектор из 18 элементов (от 0 до 17).

Все преобразования выполняются с помощью встроенных функций библиотеки `numpy`.

### 3. Функция `dqn`

Основная функция для обучения агента с использованием алгоритма Deep Q-Learning. Она управляет процессом взаимодействия агента с окружением, собирает опыт и обучает модель.

#### 3.1 Параметры функции:

`n_episodes`: Количество эпизодов (игр), которые будут проведены для обучения.

`eps_start`: Начальное значение эпсилон (вероятность случайного выбора действия).

`eps_end`: Минимальное значение эпсилон.

`eps_decay`: Коэффициент уменьшения эпсилон на каждом эпизоде.

`step_penalty`: Штраф за каждый шаг (если нужно уменьшить награду за простой шаг).

`sample_mode`: Режим выборки опыта из реплей буфера (например, случайный или на основе максимального значения на доске).

`start_learn_iterations`: Количество итераций обучения после каждого эпизода.

### 3.2 Основной цикл обучения:

**Сброс окружения.** В начале каждого эпизода окружение сбрасывается, и агент начинает новую игру.

**Получение состояния.** Состояние игры преобразуется в одномерный вектор с помощью функции `transform_state`.

**Выбор действия.** Агент выбирает действие на основе текущего состояния. Действие может быть выбрано случайным образом (с вероятностью `eps`) или на основе максимального Q-значения (эпсилон-жадная стратегия).

**Выполнение действия.** Агент выполняет выбранное действие в окружении, получает награду и новое состояние.

**Сохранение опыта.** Результаты шага (состояние, действие, награда, новое состояние, флаг завершения) сохраняются в реплей буфере.

**Обучение.** После завершения эпизода агент обучается на основе опыта, сохраненного в реплей буфере.

**Отслеживание прогресса.** В процессе обучения агент отслеживает различные метрики, такие как счет игры, общая награда, максимальное значение на доске, количество шагов и т.д.

**Сохранение модели и истории игр.** После каждых 100 эпизодов состояние модели и история лучших и худших игр сохраняются для анализа.

**Уменьшение эпсилон.** Эпсилон постепенно уменьшается на каждом эпизоде, чтобы уменьшить вероятность случайного выбора действий и перейти к использованию обученной политики.

## 4. Функции для сохранения и загрузки состояния

`save_state(name, eps)`: Сохраняет текущее состояние агента, включая веса нейронных сетей, оптимизатора, планировщика скорости обучения и статистику.

Параметры:

`name`: Имя версии агента.

`eps`: Текущее значение эпсилон.

`save_game_history(name, best_history, worst_history)`: Сохраняет историю лучших и худших игр для анализа.

Параметры:

`name`: Имя версии агента.

best\_history: История лучших игр.

worst\_history: История худших игр.

load\_game\_history(name): Загружает историю игр из файлов.

Параметры:

name: Имя версии агента.

## **5. Инициализация окружения и агента**

Создается объект env класса Game, который представляет собой среду игры '2048'. Окружение инициализируется с размером доски 4x4 и параметрами награды (reward\_mode='log2', negative\_reward=-3, cell\_move\_penalty=0.1). Также создается объект agent класса Agent, который будет взаимодействовать с окружением. Агент инициализируется с параметрами, такими как размер состояния (state\_size), размер действия (action\_size), количество нейронов в скрытых слоях (fc1\_units, fc2\_units, fc3\_units), размер буфера воспроизведения (buffer\_size), размер мини-батча (batch\_size), скорость обучения (lr), и другие параметры.

## **6. Запуск обучения**

Вызывается функция dqn с такими параметрами, как: количество эпизодов, начальное значение эпсилон, минимальное значение эпсилон, коэффициент уменьшения эпсилон, штраф за шаг, режим выборки опыта и количество итераций обучения после каждого эпизода. Запускается основной цикл обучения, и агент начинает взаимодействовать с окружением, собирать опыт и обучаться.

## **Описание файла agent.py**

В файле содержатся два основных класса: класс Agent и класс ReplayBuffer.

## **Описание класса Agent**

Класс Agent представляет собой агент, который взаимодействует с окружением (в данном случае с игрой '2048') и обучается с помощью алгоритма Deep Q-Learning. Агент использует нейронную сеть (QNetwork) для аппроксимации Q-функции, которая оценивает ожидаемую награду за выполнение определенного действия в заданном состоянии. При создании объекта класса Agent происходит инициализация различных параметров и структур данных, необходимых для работы агента.

### **Параметры инициализации класса Agent:**

state\_size: Размерность состояния (например, для игры '2048' это количество ячеек на доске).

action\_size: Количество возможных действий (в игре '2048' это 4 действия: вверх, вниз, влево, вправо).

seed: Сид для генерации случайных чисел.

fc1\_units, fc2\_units, fc3\_units: Количество нейронов в скрытых слоях нейронной сети.

buffer\_size: Размер буфера воспроизведения (replay buffer).

batch\_size: Размер мини-батча для обучения.

lr: Скорость обучения (learning rate).

use\_expected\_rewards: Флаг, указывающий, использовать ли ожидаемые награды для обучения.

predict\_steps: Количество шагов вперед для предсказания ожидаемых наград.

#### **Атрибуты класса:**

state\_size, action\_size, seed: Сохраняются параметры состояния, действий и сида.

batch\_size: Размер мини-батча.

losses: Список для хранения потерь (loss) во время обучения.

use\_expected\_rewards: Флаг для использования ожидаемых наград.

current\_iteration: Счетчик текущей итерации обучения.

scores\_list, last\_n\_scores, mean\_scores: Списки для отслеживания счетов игроков.

total\_rewards\_list, last\_n\_total\_rewards, mean\_total\_rewards: Списки для отслеживания общей награды.

max\_vals\_list, last\_n\_vals, mean\_vals: Списки для отслеживания максимального значения на доске.

max\_steps\_list, last\_n\_steps, mean\_steps: Списки для отслеживания количества шагов за эпизод.

actions\_avg\_list, actions\_deque: Списки для отслеживания распределения действий.

qnetwork\_local, qnetwork\_target: Локальная и целевая нейронные сети.

optimizer: Оптимизатор (Adam) для обучения нейронной сети.

lr\_decay: Планировщик скорости обучения (learning rate scheduler).

memory: Реплей буфер для хранения опыта.

t\_step, steps\_ahead: Счетчик шагов и количество шагов вперед для предсказания наград.

#### **Методы класса Agent**

1. save(self, name): Сохраняет веса локальной и целевой нейронных сетей, состояние оптимизатора и планировщика скорости обучения, статистику агента (например, счета, награды, шаги и т.д.) в файл с помощью pickle.

Параметры:

name: Имя версии агента.

2. step(self, state, action, reward, next\_state, done, error, action\_dist): Добавляет опыт в реплей буфер с помощью метода add класса ReplayBuffer.

Параметры:

state: Текущее состояние.

action: Выполненное действие.

reward: Награда за действие.

next\_state: Следующее состояние.

done: Флаг, указывающий, завершился ли эпизод.

error: Ошибка между предсказанной и фактической наградой.

action\_dist: Распределение действий.

3. `act(self, state, eps=0.)`: Преобразует состояние в тензор и передает его в локальную нейронную сеть (`qnetwork_local`) для получения Q-значений. Возвращает Q-значения в виде numpy массива.

Параметры:

state: Текущее состояние.

eps: Значение эpsilon для эpsilon-жадной стратегии.

4. `learn(self, learn_iterations, mode='board_max', save_loss=True)`: Обучает агента на основе опыта, хранящегося в реплей буфере.

Если используются ожидаемые награды, вычисляет их для опыта в реплей буфере. Добавляет опыт из текущего эпизода в реплей буфер. Если в реплей буфере достаточно опыта, выбирает мини-батч и обучает локальную нейронную сеть. Вычисляет ошибку (loss) между ожидаемыми и фактическими Q-значениями и обновляет веса сети. Сохраняет среднюю потерю, если `save_loss` равен True.

Параметры:

`learn_iterations`: Количество итераций обучения.

`mode`: Режим выборки из реплей буфера (например, 'board\_max' или 'random').

`save_loss`: Флаг, указывающий, сохранять ли потери.

## Описание класса `ReplayBuffer`

Класс `ReplayBuffer` отвечает за хранение опыта (experience), который агент получает во время взаимодействия с окружением. Этот опыт используется для обучения модели на случайных выборках, чтобы улучшить обобщающую способность алгоритма.

### Параметры инициализации (`__init__`)

`action_size` (int): Размерность действий в среде. Если в среде доступны 4 действия (влево, вправо, вверх, вниз), то `action_size = 4`.

`buffer_size` (int): Максимальный размер буфера воспроизведения. Это количество опыта, которое будет храниться в буфере. По умолчанию используется значение `BUFFER_SIZE = 100000`.

`batch_size` (int): Размер мини-батча, который будет использоваться для обучения модели.

По умолчанию используется значение `BATCH_SIZE = 1024`.

`seed (int)`: Зерно для генерации случайных чисел. Используется для обеспечения воспроизводимости случайной выборки из буфера.

#### **Атрибуты класса**

`action_size (int)`: Размерность действий в среде. Хранится для дальнейшего использования.

`memory (deque)`: Основной буфер, в котором хранится опыт. Реализован как очередь (`deque`) с максимальной длиной `buffer_size`. Когда буфер заполняется, самые старые опыты удаляются.

`episode_memory (list)`: Временный буфер для хранения опыта текущего эпизода. После завершения эпизода опыт из `episode_memory` переносится в основной буфер `memory`.

`batch_size (int)`: Размер мини-батча для обучения. Используется при выборке из буфера.

`seed (int)`: Зерно для генерации случайных чисел.

`experience (namedtuple)`: Именованный кортеж для хранения опыта.

#### **Поля:**

`state`: Текущее состояние.

`action`: Выбранное действие.

`reward`: Награда за действие.

`next_state`: Следующее состояние.

`done`: Флаг завершения эпизода.

`error`: Ошибка (не используется в данном коде).

`action_dist`: Распределение действий (не используется в данном коде).

`weight`: Вес опыта (не используется в данном коде).

#### **Методы класса**

`dump(self)`: Сохраняет состояние буфера в словарь.

Используется для сериализации буфера (например, для сохранения на диск).

`load(self, d)`: Загружает состояние буфера из словаря.

Используется для десериализации буфера (например, для загрузки с диска).

#### **Параметры:**

`d (dict)`: Словарь, содержащий сохраненное состояние буфера.

`add (self, state, action, reward, next_state, done, error, action_dist, weight=None)`: Добавляет новый опыт в текущий эпизод. Опыт сохраняется во временном буфере `episode_memory`.

#### **Параметры:**

`state (array_like)`: Текущее состояние.

`action (int)`: Выбранное действие.

`reward (float)`: Награда за действие.

`next_state` (array\_like): Следующее состояние.

`done` (bool): Флаг завершения эпизода.

`error` (float): Ошибка (не используется в данном коде).

`action_dist` (array\_like): Распределение действий (не используется в данном коде).

`weight` (float): Вес опыта (не используется в данном коде).

`add_episode_experiences(self)`: Переносит опыт из текущего эпизода (`episode_memory`) в основной буфер (`memory`). После переноса очищает `episode_memory`.

`calc_expected_rewards(self, steps_ahead=1)`: Вычисляет ожидаемые награды для текущего эпизода. Вычисляет сумму наград за `steps_ahead` шагов для каждого опыта в `episode_memory`. Заменяет текущую награду на ожидаемую награду.

Параметры:

`steps_ahead` (int): Количество шагов вперед для предсказания наград.

`sample(self, mode='board_max')`: Выбирает опыт из буфера в зависимости от режима. Возвращает мини-батч опыта в виде тензоров для обучения модели.

Параметры:

`mode` (str): Режим выборки.

'random': Случайная выборка из буфера.

'board\_max': Выборка с учетом максимального значения на игровом поле (например, для задачи 2048).

`__len__(self)`: Возвращает текущий размер буфера (длину очереди `memory`).

## Описание файла `model.py`

В файле реализован класс `QNetwork`. Он представляет собой нейронную сеть, которая используется для аппроксимации функции Q-значений в задаче обучения с подкреплением (Reinforcement Learning, RL). Q-значения представляют собой ожидаемую сумму будущих наград, которую агент может получить, выполнив определенное действие в заданном состоянии. Это ключевая концепция в RL, так как она позволяет агенту оценивать, какое действие приведет к наибольшей общей награде в долгосрочной перспективе.

## Архитектура нейронной сети

Класс `QNetwork` состоит из нескольких полносвязных (fully connected) слоев, что делает её подходящей для задач, где состояние и действия могут быть представлены в виде векторов. Архитектура сети включает следующие компоненты:

Входной слой: Принимает состояние игры (`state`) в виде вектора. Размер входного слоя равен `state_size`, что в данном проекте соответствует количеству ячеек на игровом поле (например, для поля 4x4 это 16 ячеек).



Скрытые слои:

Первый скрытый слой (fc1) с fc1\_units нейронами.

Второй скрытый слой (fc2) с fc2\_units нейронами.

Третий скрытый слой (fc3) с fc3\_units нейронами.

Каждый скрытый слой сопровождается функцией активации ReLU (Rectified Linear Unit) и слоем нормализации (BatchNorm1d). ReLU вводит нелинейность в сеть, что позволяет модели аппроксимировать более сложные зависимости между состояниями и действиями. Слой нормализации (BatchNorm1d) помогает стабилизировать обучение, уменьшая проблему "внутреннего ковариантного сдвига" (internal covariate shift), что ускоряет сходимость и улучшает стабильность обучения.

Выходной слой: Выходной слой имеет action\_size нейронов, где action\_size — это количество возможных действий, которые может выполнить агент. В данном случае, в игре '2048' агент может выполнять 4 действия: вверх, вниз, влево, вправо. Таким образом, выходной слой имеет 4 нейрона, каждый из которых соответствует одному из возможных действий.

## Взаимодействие класса QNetwork с проектом

### 1. Взаимодействие с agent.py

В файле agent.py класс Agent использует два экземпляра QNetwork: qnetwork\_local и qnetwork\_target. Эти сети взаимодействуют следующим образом:

qnetwork\_local: Это основная сеть, которая используется для выбора действий и обучения. Агент использует эту сеть для вычисления Q-значений в текущем состоянии и выбора действия с помощью эпсилон-жадной стратегии.

Метод act: В методе act агент преобразует текущее состояние в тензор, передает его в qnetwork\_local и получает Q-значения для всех возможных действий. Затем он выбирает действие либо случайным образом (с вероятностью eps), либо на основе максимального Q-значения.

Метод learn: В методе learn агент использует qnetwork\_local для вычисления ожидаемых Q-значений (Q\_expected) на основе текущих состояний и действий. Затем он вычисляет ошибку между ожидаемыми и фактическими Q-значениями и обновляет веса сети с помощью градиентного спуска.

qnetwork\_target: Это целевая сеть, которая используется для стабилизации обучения. Она обновляется мягко (soft update) с помощью параметров qnetwork\_local с использованием коэффициента TAU. Это помогает избежать быстрого изменения целевых Q-значений, что может привести к нестабильности обучения. Целевая сеть используется для

вычисления целевых Q-значений, которые сравниваются с ожидаемыми Q-значениями из `qnetwork_local`.

Обучение: В процессе обучения агент собирает опыт (experience) в реплей буфере и периодически обновляет веса `qnetwork_local` на основе мини-батчей из реплей буфера. Целевая сеть `qnetwork_target` используется для вычисления целевых Q-значений, которые сравниваются с ожидаемыми Q-значениями из `qnetwork_local`.

## **2. Взаимодействие с функцией `dqn` из `trainin.ipynb`**

Функция `dqn` в файле `trainin.ipynb` является основным циклом обучения, который управляет взаимодействием между агентом и окружением. Вот как `QNetwork` взаимодействует с этой функцией:

Инициализация: В начале функции `dqn` создается объект `env` класса `Game`, который представляет собой среду игры '2048'. Затем создается объект `agent` класса `Agent`, который использует `QNetwork` для аппроксимации Q-функции.

Цикл обучения: В цикле обучения агент взаимодействует с окружением, выполняя действия и получая награды. Состояние игры преобразуется в одномерный вектор с помощью функции `transform_state` и передается в `QNetwork` для вычисления Q-значений. Агент выбирает действие на основе этих Q-значений и эпсилон-жадной стратегии.

Обучение: После каждого эпизода агент обучается на основе опыта, сохраненного в реплей буфере. В процессе обучения `QNetwork` обновляет свои веса, чтобы минимизировать ошибку между ожидаемыми и фактическими Q-значениями.

Отслеживание прогресса: В процессе обучения агент отслеживает различные метрики, такие как счет игры, общая награда, максимальное значение на доске, количество шагов и т.д. Эти метрики используются для анализа прогресса обучения.

Класс `QNetwork` в данном проекте представляет собой простую, но эффективную нейронную сеть, которая хорошо подходит для задачи обучения с подкреплением в игре '2048'. Её архитектура с несколькими полносвязными слоями и использованием нормализации и ReLU активаций позволяет сети аппроксимировать Q-значения для каждого состояния и действия. Агент в `agent.py` взаимодействует с этой сетью для выбора действий и обучения, используя как локальную, так и целевую сети для стабилизации процесса обучения.

## **Описание `game.cpp`**

Код этой программы реализует нашу среду, то есть игровую механику популярной головоломкой "2048". Игрок управляет плитками на игровом поле фиксированного размера (по умолчанию 4x4), передвигая их в одном из четырех направлений (вверх, вниз, влево, вправо). После каждого хода плитки сливаются, если их значения равны, и на поле

появляется новая плитка со значением 2 или 4. Цель игры — набрать как можно больше очков, объединяя плитки с одинаковыми значениями. Игра завершается, если нет доступных ходов.

## **Класс Game**

### **Приватные переменные**

board\_dim: Размер игрового поля (4 для 4x4).

state\_size: Размер состояния игры, вычисляется как board\_dim \* board\_dim.

action\_size: Количество возможных действий (вверх, вниз, влево, вправо).

negative\_reward: Награда, которая выдается за неэффективные действия (например, попытка хода, который не изменяет поле).

reward\_mode: Режим расчета награды. Возможные значения:

- "log2": Награда вычисляется как логарифм по основанию 2 от значения плитки.
- Другое значение: Награда вычисляется как значение плитки.

cell\_move\_penalty: Штраф за перемещение плитки.

game\_board: Двумерный массив, представляющий игровое поле.

score: Текущий счет игры.

reward: Награда за последний ход.

current\_cell\_move\_penalty: Текущий штраф за перемещение плитки.

done: Флаг, указывающий, завершена ли игра.

steps: Количество сделанных ходов.

rewards\_list: Список наград за каждый ход.

scores\_list: Список счетов за каждый ход.

step\_penalty: Штраф за каждый ход.

history: История игры, содержит информацию о каждом ходе (действие, значения действий, состояние до хода, состояние после хода, счет, награда)

rng: Генератор случайных чисел для заполнения пустых ячеек.

moved: Флаг, указывающий, было ли поле изменено после хода.

### **Приватные методы**

shift:

- Параметры:

- 'const std::vector<std::vector<double>>& board' — текущее состояние поля.

- Описание: Сдвигает все плитки в каждой строке влево, объединяя соседние плитки с одинаковыми значениями.

- Возвращает: Новое состояние поля после сдвига.

transpose:

- Параметры:

- ``const std::vector<std::vector<double>>& board`` — текущее состояние поля.

- Описание: Транспонирует поле (меняет строки и столбцы местами).

- Возвращает: Транспонированное поле.

`flip_horizontal`:

- Параметры:

- ``const std::vector<std::vector<double>>& board`` — текущее состояние поля.

- Описание: Отражает поле по горизонтали.

- Возвращает: Отраженное поле.

`calc_board`:

- Параметры:

- ``const std::vector<std::vector<double>>& board`` — текущее состояние поля.

- Описание: Выполняет сдвиг и объединение плиток, вычисляя награду.

- Возвращает: Новое состояние поля после объединения.

`process_action`:

- Параметры:

- ``int action`` — действие (0 — вверх, 1 — вниз, 2 — влево, 3 — вправо).

- ``const std::vector<std::vector<double>>& board`` — текущее состояние поля.

- Описание: Обработывает действие, вызывая соответствующие преобразования поля.

- Возвращает: Новое состояние поля после выполнения действия.

### **Публичные методы**

Конструктор `Game`:

- Параметры:

- ``int size`` — размер поля (по умолчанию 4).

- ``int seed`` — начальное значение для генератора случайных чисел (по умолчанию 42).

- ``double negative_reward`` — награда за неэффективные действия (по умолчанию -10.0).

- ``std::string reward_mode`` — режим расчета награды (по умолчанию `"log2"`).

- ``double cell_move_penalty`` — штраф за перемещение плитки (по умолчанию 0.1).

- Описание: Инициализирует игру с заданными параметрами.

`reset`:

- Параметры:

- ``int init_fields`` — количество начальных плиток (по умолчанию 2).

- ``double step_penalty`` — штраф за каждый ход (по умолчанию 0.0).

- Описание: Сбрасывает игру в начальное состояние.

`current_state`:

- Описание: Возвращает текущее состояние игры в виде одномерного массива (вектора).
- Возвращает: `py::array_t<double>` — текущее состояние.

`step`:

- Параметры:
  - ``int action`` — действие (0 — вверх, 1 — вниз, 2 — влево, 3 — вправо).
  - ``const py::array_t<double>& action_values`` — значения действий.
- Описание: Выполняет действие, обновляет состояние игры и возвращает результат.
- Возвращает: Кортеж ``(game_board, reward, done)``.

`check_is_done` (перегрузка 1):

- Параметры:
  - ``const std::vector<std::vector<double>>& board`` — текущее состояние поля.
- Описание: Проверяет, завершена ли игра (нет пустых ячеек и нет возможных слияний).
- Возвращает: ``bool`` — ``true``, если игра завершена.

`check_is_done` (перегрузка 2):

- Описание: Проверяет, завершена ли текущая игра.
- Возвращает: ``bool`` — ``true``, если игра завершена.

`fill_random_empty_cell`:

- Описание: Заполняет случайную пустую ячейку на поле значением 2 или 4.

`draw_board`:

- Параметры:
  - ``const std::vector<std::vector<double>>& board`` — текущее состояние поля.
  - ``const std::string& title`` — заголовок для отображения.
- Описание: Визуализирует игровое поле с помощью ``matplotlib``.

### Геттеры, сеттеры:

`get_history`: Возвращает историю игры в виде списка кортежей.

`get_score`: Возвращает текущий счет игры.

`get_reward`: Возвращает награду за последний ход.

`get_negative_reward`: Возвращает награду за неэффективные действия.

`get_done`: Возвращает флаг, указывающий, завершена ли игра.

`get_moved`: Возвращает флаг, указывающий, было ли поле изменено после последнего хода.

`get_steps`: Возвращает количество сделанных ходов.

`set_moved`: Устанавливает флаг ``moved``.

`get_reward_mode`: Возвращает текущий режим расчета награды.

Код реализует среду для игры 2048, которая может использоваться для обучения агентов в Python. Он включает механизмы для выполнения действий, расчета наград, проверки завершения игры и визуализации состояния.

### Результаты обучения

В ходе обучения модель смогла достигнуть неплохих результатов за 6000 итераций. Рассмотрим их наглядно.



Рисунок 2. Шаги в наилучшей игре.

На рисунке 2 видим максимальный результат, которого удалось достичь. Максимальное значение в клетке – 512. Также можно заметить, что на предпоследнем ходу было сделано не лучшее действие (влево, вместо вправо) это связано с тем, что в стратегии преобладали только два направления. Можно предположить, что существует стратегия, с помощью которой модель могла бы получить больший результат за то же количество итераций.

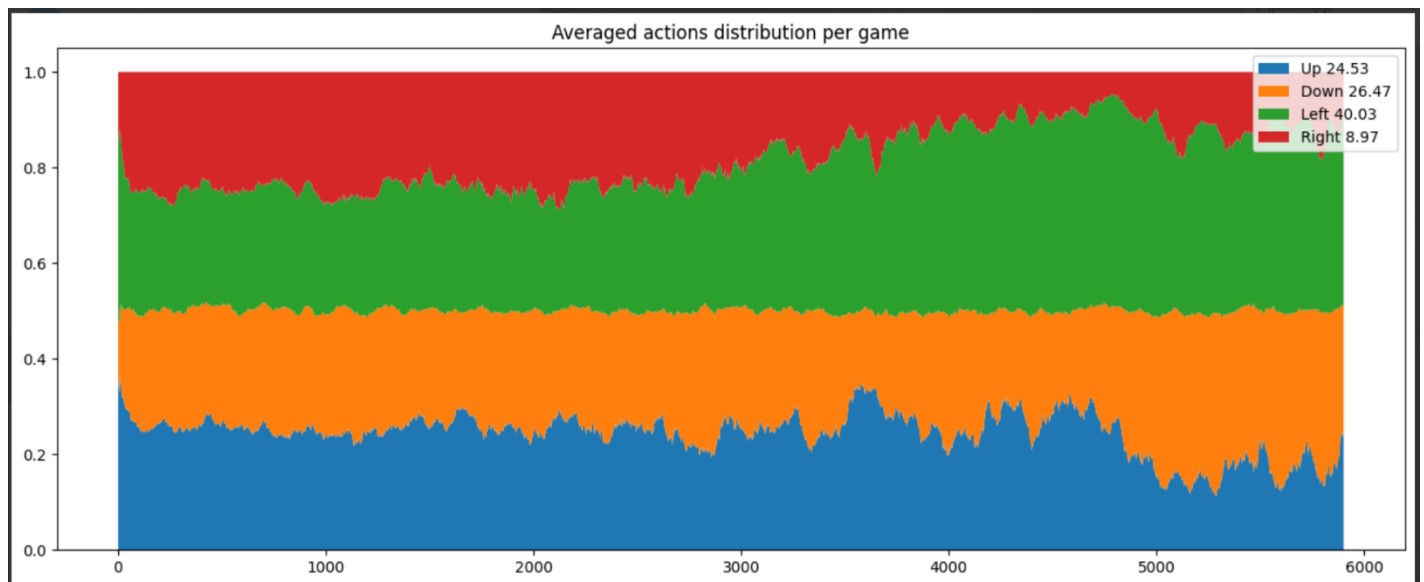


Рисунок 3. Распределение долей выбранных направлений ходов в каждой из игр.

На рисунке 3 видим, как постепенно менялась стратегия в правильную сторону: в первых играх направления сдвигов выбирались равномерно, но затем появились “приоритетные” - влево и вниз.

Основная стратегия игры "2048" заключается в концентрации крупных значений в одном углу доски. Сдвиг вправо или вверх нарушает эту стратегию, так как сдвигаются уже сосредоточенные клетки, что приводит к большому штрафу. Агент учится избегать таких сдвигов, так как они не приносят выгоды и только усложняют ситуацию на доске. Однако в исключительных случаях (например, когда сдвиг вправо или вверх может привести к схлопыванию крупных клеток и получению большой награды), агент может сделать такой ход, если ожидаемая награда с учётом штрафов оказывается выше.

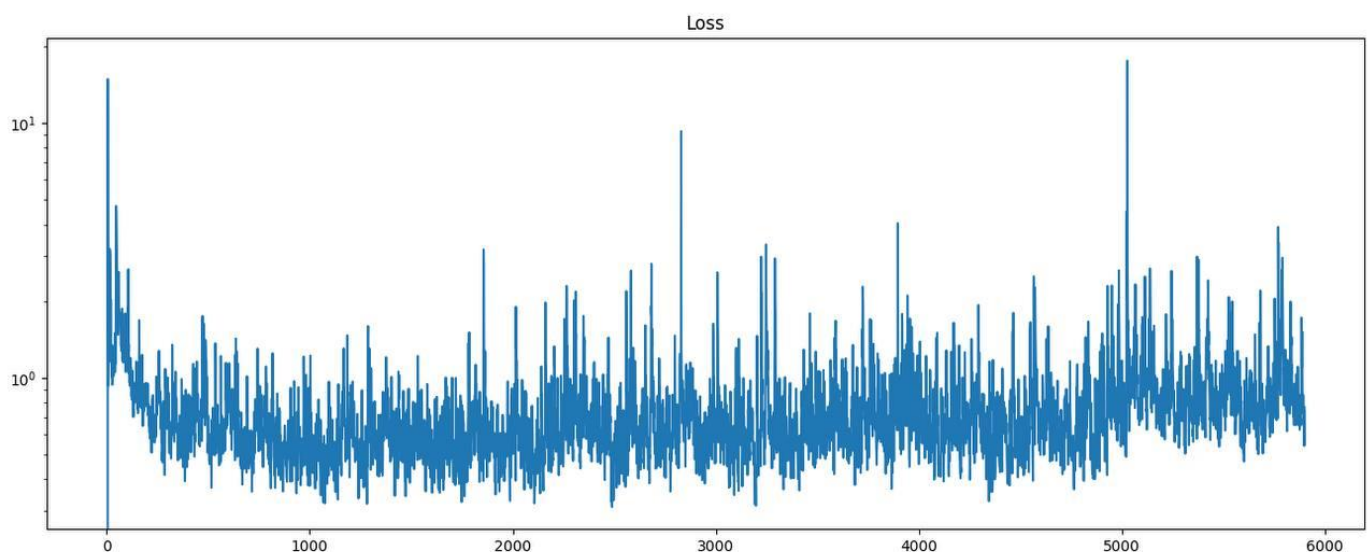


Рисунок 4. Функция потерь DQN.

Чтобы найти функцию потерь для метода Deep Q-Learning (DQN), вычисляется среднеквадратичная ошибка между текущими Q-значениями и целевыми Q-значениями. Целевые Q-значения рассчитываются на основе текущих наград и максимальных Q-значений следующего состояния, дисконтированных с учётом коэффициента гамма.

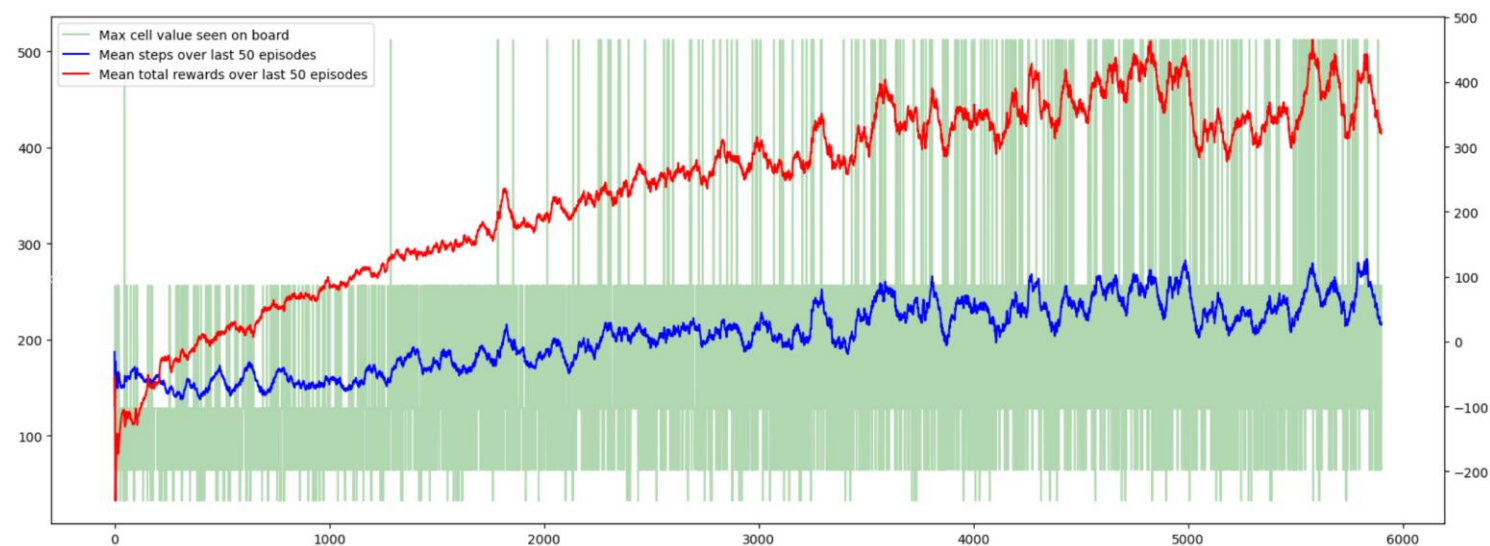


Рисунок 5. Основные показатели результата обучения.

Зеленый график на рисунке 5 показывает максимальное значение клетки в каждой игре. Видно, что с ростом количества итераций, учащаются более высокие значения в клетках. Примерно на 5600 итерациях программа стабильно доходила до максимального значения в клетке 512, это говорит о том, что модель и вправду обучалась, и, вполне вероятно, что, если дать ей больше времени на обучения, она смогла бы достигнуть цели игры – получить клетку 2048.

Также можно проследить положительный рост красной кривой. Значит с каждым разом средняя награда за последние 50 ходов в игре тоже возрастала, что опять же говорит об успешном обучении модели.



## **Заключение**

В ходе работы были рассмотрены различные алгоритмы обучения с подкреплением, а также проведен их сравнительный анализ. Была реализована игра на плоском дискретном поле «2048» с дальнейшим подключением к ней обучения с подкреплением. В качестве алгоритма обучения была реализована модель DQN (Deep Q-Network), которая использует нейронную сеть для оценки значений Q-функции. Обучение было успешно выполнено, что подтверждается графиками общих результатов работы модели в среде. Максимальное значение в клетке, которого удалось достичь – 512. Немало важно, что это не является выбросом и модель может практически стабильно достигать таких результатов.

## Список литературы

1. Ротова, О. М. ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ: ВВЕДЕНИЕ / О. М. Ротова, А. Д. Шибанова // "Теория и практика современной науки". – 2020. – № 1. – С. 477-482.
2. АЛГОРИТМЫ РАННЕГО ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ / А. Н. Шарибаев, Р. Н. Шарибаев, Б. Т. Абдулазизов, М. Р. Тохиржонова // "Экономика и социум". – 2023. – № 6. – С. 1124-1126.
3. Саттон, Р. Л. Подкрепление: основы, модели и алгоритмы / Р. Л. Саттон, А. Г. Бартон. — 2-е изд., перераб. и доп. — Москва : Диалектика, 2018. — 530 с.
4. Моралес М. Г. Грокаем глубокое обучение с подкреплением / М. Г. Моралес ; пер. с англ. — Санкт-Петербург : Питер, 2020. — 352 с.
5. Шилдт, Г. Самоучитель C++, 3-е издание / Г. Шилдт; пер. с англ. — Санкт-Петербург: БХВ-Петербург, 2003 — 688 с.

## Приложение

```
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>
#include <pybind11/stl.h>
#include <vector>
#include <random>
#include <algorithm>
#include <fstream>
#include <tuple>
#include <iostream>
#include <numeric>

namespace py = pybind11;

class Game {
private:
    int board_dim;
    int state_size;
    int action_size;
    double negative_reward;
    std::string reward_mode;
    double cell_move_penalty;
    std::vector<std::vector<double>> game_board;
    double score;
    double reward;
    double current_cell_move_penalty;
    bool done;
    int steps;
    std::vector<double> rewards_list;
    std::vector<double> scores_list;
    double step_penalty;
    std::vector<std::tuple<int, const double*,
std::vector<std::vector<double>>, std::vector<std::vector<double>>, double,
double>> history;
    std::mt19937 rng;
    bool moved;

    std::vector<std::vector<double>> shift(const
std::vector<std::vector<double>>& board) {
        std::vector<std::vector<double>> shifted_board(board_dim,
std::vector<double>(board_dim, 0.0));
        for (int i = 0; i < board_dim; ++i) {
            std::vector<double> shifted(board_dim, 0.0);
            int idx = 0;
            for (int j = 0; j < board_dim; ++j) {
                if (board[i][j] != 0.0) {
                    shifted[idx] = board[i][j];
                    if (j != idx) {
                        current_cell_move_penalty += cell_move_penalty *
board[i][j];
                    }
                    ++idx;
                }
            }
            shifted_board[i] = shifted;
        }
        return shifted_board;
    }

    std::vector<std::vector<double>> transpose(const
std::vector<std::vector<double>>& board) {
```

```

        std::vector<std::vector<double>> transposed_board(board_dim,
std::vector<double>(board_dim, 0.0));
        for (int i = 0; i < board_dim; ++i) {
            for (int j = 0; j < board_dim; ++j) {
                transposed_board[j][i] = board[i][j];
            }
        }
        return transposed_board;
    }

    std::vector<std::vector<double>> flip_horizontal(const
std::vector<std::vector<double>>& board) {
        std::vector<std::vector<double>> flipped_board(board_dim,
std::vector<double>(board_dim, 0.0));
        for (int i = 0; i < board_dim; ++i) {
            for (int j = 0; j < board_dim; ++j) {
                flipped_board[i][j] = board[i][board_dim - 1 - j];
            }
        }
        return flipped_board;
    }

    std::vector<std::vector<double>> calc_board(const
std::vector<std::vector<double>>& board) {
        reward = 0.0;
        current_cell_move_penalty = 0.0;

        std::vector<std::vector<double>> shifted_board = shift(board);
        std::vector<std::vector<double>> merged_board(board_dim,
std::vector<double>(board_dim, 0.0));
        for (int i = 0; i < board_dim; ++i) {
            for (int j = 0; j < board_dim - 1; ++j) {
                if (shifted_board[i][j] != 0.0 && shifted_board[i][j] ==
shifted_board[i][j + 1]) {
                    shifted_board[i][j] *= 2.0;
                    shifted_board[i][j + 1] = 0.0;
                    if (reward_mode == "log2") {
                        reward += std::log2(shifted_board[i][j]);
                    }
                    else {
                        reward += shifted_board[i][j];
                    }
                }
            }
            merged_board[i] = shifted_board[i];
        }
        merged_board = shift(merged_board);
        return merged_board;
    }

    std::vector<std::vector<double>> process_action(int action, const
std::vector<std::vector<double>>& board) {
        std::vector<std::vector<double>> temp_board = board;

        if (action == 0) { // ACTION_UP
            temp_board = transpose(calc_board(transpose(temp_board)));
        }
        else if (action == 1) { // ACTION_DOWN
            temp_board =
transpose(flip_horizontal(calc_board(flip_horizontal(transpose(temp_board))))
);
        }
        else if (action == 2) { // ACTION_LEFT

```

```

        temp_board = calc_board(temp_board);
    }
    else if (action == 3) { // ACTION_RIGHT
        temp_board =
flip_horizontal(calc_board(flip_horizontal(temp_board)));
    }

    return temp_board;
}

public:
    Game(int size = 4, int seed = 42, double negative_reward = -10.0,
std::string reward_mode = "log2", double cell_move_penalty = 0.1)
        : board_dim(size), state_size(size* size), action_size(4),
negative_reward(negative_reward), reward_mode(reward_mode),
cell_move_penalty(cell_move_penalty),
        score(0.0), reward(0.0), current_cell_move_penalty(0.0), done(false),
steps(0), moved(false), step_penalty(0.0), rng(seed) {}

    void reset(int init_fields = 2, double step_penalty = 0.0) {
        game_board = std::vector<std::vector<double>>(board_dim,
std::vector<double>(board_dim, 0.0));

        for (int i = 0; i < init_fields; ++i) {
            fill_random_empty_cell();
        }

        score = std::accumulate(game_board.begin(), game_board.end(), 0.0,
[] (double sum, const std::vector<double>& row) {
            return sum + std::accumulate(row.begin(), row.end(), 0.0);
        });
        reward = 0.0;
        current_cell_move_penalty = 0.0;
        done = false;
        steps = 0;
        rewards_list.clear();
        scores_list.clear();
        step_penalty = step_penalty;
        history.clear();

        history.push_back({
            -1,
            nullptr,
            game_board,
            std::vector<std::vector<double>>(),
            score,
            reward
        });
    }

    py::array_t<double> current_state() {
        std::vector<double> state;
        for (const auto& row : game_board) {
            state.insert(state.end(), row.begin(), row.end());
        }
        return py::array_t<double>(state.size(), state.data());
    }

    py::tuple step(int action, const py::array_t<double>& action_values) {
        std::vector<std::vector<double>> old_board = game_board;
        std::vector<std::vector<double>> temp_board = process_action(action,
game_board);

```

```

        if (game_board != temp_board) {
            game_board = temp_board;
            fill_random_empty_cell();
            reward = reward - current_cell_move_penalty;
            score = std::accumulate(game_board.begin(), game_board.end(),
0.0, [](double sum, const std::vector<double>& row) {
                return sum + std::accumulate(row.begin(), row.end(), 0.0);
            });
            done = check_is_done();
            moved = true;
        }
        else {
            reward = negative_reward;
            moved = false;
        }
        steps += 1;
        rewards_list.push_back(reward);

        history.push_back({
            action,
            action_values.data(),
            old_board,
            game_board,
            score,
            reward
        });

        return py::make_tuple(game_board, reward, done);
    }

    bool check_is_done() {
        return check_is_done(game_board);
    }

    bool check_is_done(const std::vector<std::vector<double>>& board) {
        for (const auto& row : board) {
            if (std::find(row.begin(), row.end(), 0.0) != row.end()) {
                return false;
            }
        }

        for (const auto& row : board) {
            for (size_t i = 0; i < row.size() - 1; ++i) {
                if (row[i] == row[i + 1]) {
                    return false;
                }
            }
        }

        for (size_t i = 0; i < board.size(); ++i) {
            for (size_t j = 0; j < board[i].size() - 1; ++j) {
                if (board[j][i] == board[j + 1][i]) {
                    return false;
                }
            }
        }

        return true;
    }

    void fill_random_empty_cell() {
        std::vector<std::pair<int, int>> empty_cells;
        for (int i = 0; i < board_dim; ++i) {

```

```

        for (int j = 0; j < board_dim; ++j) {
            if (game_board[i][j] == 0.0) {
                empty_cells.emplace_back(i, j);
            }
        }

        if (empty_cells.empty()) {
            return;
        }

        std::uniform_int_distribution<int> dist(0, empty_cells.size() - 1);
        int index = dist(rng);
        int x = empty_cells[index].first;
        int y = empty_cells[index].second;

        game_board[x][y] = (std::uniform_real_distribution<double>(0.0,
1.0)(rng) < 0.9) ? 2.0 : 4.0;
    }

    std::vector<std::tuple<int, py::array_t<double>,
std::vector<std::vector<double>>, std::vector<std::vector<double>>, double,
double>> get_history() const {
        std::vector<std::tuple<int, py::array_t<double>,
std::vector<std::vector<double>>, std::vector<std::vector<double>>, double,
double>> history_copy;
        for (const auto& entry : history) {
            const double* action_values_ptr = std::get<1>(entry);
            py::array_t<double> action_values(action_size,
action_values_ptr);

            history_copy.emplace_back(
                std::get<0>(entry),
                action_values,
                std::get<2>(entry),
                std::get<3>(entry),
                std::get<4>(entry),
                std::get<5>(entry)
            );
        }
        return history_copy;
    }

    double get_score() const {
        return score;
    }

    double get_reward() const {
        return reward;
    }

    double get_negative_reward() const {
        return negative_reward;
    }

    bool get_done() const {
        return done;
    }

    bool get_moved() const {
        return moved;
    }

```

```

int get_steps() const {
    return steps;
}

void set_moved(bool value) {
    moved = value;
}

std::string get_reward_mode() const {
    return reward_mode;
}

void draw_board(const std::vector<std::vector<double>>& board, const
std::string& title) {
    py::module plt = py::module::import("matplotlib.pyplot");
    py::dict cell_colors = py::dict(
        py::arg("0") = "#FFFFFF",
        py::arg("2") = "#EEE4DA",
        py::arg("4") = "#ECE0C8",
        py::arg("8") = "#ECB280",
        py::arg("16") = "#EC8D53",
        py::arg("32") = "#F57C5F",
        py::arg("64") = "#E95937",
        py::arg("128") = "#F3D96B",
        py::arg("256") = "#F2D04A",
        py::arg("512") = "#E5BF2E",
        py::arg("1024") = "#E2B814",
        py::arg("2048") = "#EBC502",
        py::arg("4096") = "#00A2D8",
        py::arg("8192") = "#9ED682"
    );

    int ncols = board.size();
    int nrows = board.size();

    plt.attr("figure")(py::arg("figsize") = py::make_tuple(3, 3));
    plt.attr("suptitle")(title);
    py::list axes;
    for (int r = 0; r < nrows; ++r) {
        for (int c = 1; c <= ncols; ++c) {
            axes.append(plt.attr("subplot")(nrows, ncols, r * ncols +
c));
        }
    }

    std::vector<double> v;
    for (const auto& row : board) {
        v.insert(v.end(), row.begin(), row.end());
    }

    for (size_t i = 0; i < axes.size(); ++i) {
        py::object ax = axes[i];
        ax.attr("text")(0.5, 0.5, std::to_string(static_cast<int>(v[i])),
            py::arg("horizontalalignment") = "center",
            py::arg("verticalalignment") = "center");

        // Èñîïëüçóâì py::str äëý âîñòóìà ê ýëâìáíòàì ñëìââöý
    }

    ax.attr("set_facecolor")(cell_colors[py::str(std::to_string(static_cast<int>(
v[i])))]));
}

// Óâëðâââì ìâðèè ìñâé

```



```

        for (const auto& ax : axes) {
            ax.attr("set_xticks") (py::list());
            ax.attr("set_yticks") (py::list());
        }

        plt.attr("show") ();
    }
};

PYBIND11_MODULE(game, m) {
    py::class_ <Game>(m, "Game")
        .def(py::init<int, int, double, std::string, double>(),
            py::arg("size") = 4,
            py::arg("seed") = 42,
            py::arg("negative_reward") = -10.0,
            py::arg("reward_mode") = "log2",
            py::arg("cell_move_penalty") = 0.1)
        .def("reset", &Game::reset,
            py::arg("init_fields") = 2,
            py::arg("step_penalty") = 0.0)
        .def("current_state", &Game::current_state)
        .def("step", &Game::step)
        .def("check_is_done", py::overload_cast<const
std::vector<std::vector<double>>>&>(&Game::check_is_done))
        .def("check_is_done", py::overload_cast<>(&Game::check_is_done))
        .def("fill_random_empty_cell", &Game::fill_random_empty_cell)
        .def("get_score", &Game::get_score)
        .def("get_reward", &Game::get_reward)
        .def("get_negative_reward", &Game::get_negative_reward)
        .def("get_done", &Game::get_done)
        .def("get_moved", &Game::get_moved)
        .def("get_steps", &Game::get_steps)
        .def("set_moved", &Game::set_moved)
        .def("get_reward_mode", &Game::get_reward_mode)
        .def("get_history", &Game::get_history)
        .def("draw_board", &Game::draw_board, py::arg("board"),
py::arg("title") = "Current game");
}

```

Код game.cpp

```

import numpy as np
import random
import pickle
from collections import namedtuple, deque
import torch
import torch.nn.functional as F
import torch.optim as optim
from model import QNetwork

BUFFER_SIZE = 100000    # replay buffer size
BATCH_SIZE = 1024       # minibatch size
LR = 0.00005            # learning rate
TAU = 0.001             # for soft update of target parameters

base_dir = './data/'

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class Agent():
    """ Interacts with and learns from the environment """

```

```

def __init__(self, state_size, action_size, seed, fcl_units=256,
fc2_units=256, fc3_units=256,
            buffer_size=BUFFER_SIZE, batch_size=BATCH_SIZE, lr=LR,
use_expected_rewards=True, predict_steps=2):
    """Initialize an Agent object.

    Params
    =====
        state_size (int): dimension of each state
        action_size (int): dimension of each action
        seed (int): random seed
        fc*_units (int): size of the respective layer
        buffer_size (int): number of steps to save in replay buffer
        batch_size (int): self-explanatory
        lr (float): learning rate
        use_expected_rewards (bool): whether to predict the weighted sum
of future rewards or just for current step
        predict_steps (int): for how many steps to predict the expected
rewards
    """
    self.state_size = state_size
    self.action_size = action_size
    self.seed = seed
    random.seed(seed)
    np.random.seed(seed)
    self.batch_size = batch_size
    self.losses = []
    self.use_expected_rewards = use_expected_rewards
    self.current_iteration = 0

    # Game scores
    self.scores_list = []
    self.last_n_scores = deque(maxlen=50)
    self.mean_scores = []
    self.max_score = 0
    self.min_score = 1000
    self.best_score_board = []

    # Rewards
    self.total_rewards_list = []
    self.last_n_total_rewards = deque(maxlen=50)
    self.mean_total_rewards = []
    self.max_total_reward = 0

    # Max cell value on game board
    self.max_vals_list = []
    self.last_n_vals = deque(maxlen=50)
    self.mean_vals = []
    self.max_val = 0
    self.best_val_board = []

    # Number of steps per episode
    self.max_steps_list = []
    self.last_n_steps = deque(maxlen=50)
    self.mean_steps = []
    self.max_steps = 0
    self.total_steps = 0

    self.actions_avg_list = []
    self.actions_deque = {
        0: deque(maxlen=50),
        1: deque(maxlen=50),
        2: deque(maxlen=50),

```

```

        3:deque(maxlen=50)
    }

    # Q-Network
    self.qnetwork_local = QNetwork(state_size, action_size, seed,
    fc1_units=fc1_units, fc2_units=fc2_units, fc3_units=fc3_units).to(device)
    self.qnetwork_target = QNetwork(state_size, action_size, seed,
    fc1_units=fc1_units, fc2_units=fc2_units, fc3_units=fc3_units).to(device)
    self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=lr)
    self.lr_decay = optim.lr_scheduler.StepLR(self.optimizer, 1000,
0.9999)

    # Replay buffer
    self.memory = ReplayBuffer(action_size, buffer_size, batch_size,
seed)

    # Initialize time step
    self.t_step = 0
    self.steps_ahead = predict_steps

    def save(self, name):
        """Saves the state of the model and stats

        Params
        =====
            name (str): name of the agent version used in dqn function
        """
        torch.save(self.qnetwork_local.state_dict(),
base_dir+'/network_local_%s.pth' % name)
        torch.save(self.qnetwork_target.state_dict(),
base_dir+'/network_target_%s.pth' % name)
        torch.save(self.optimizer.state_dict(), base_dir+'/optimizer_%s.pth'
% name)
        torch.save(self.lr_decay.state_dict(), base_dir+'/lr_schd_%s.pth' %
name)

        state = {
            'state_size': self.state_size,
            'action_size': self.action_size,
            'seed': self.seed,
            'batch_size': self.batch_size,
            'losses': self.losses,
            'use_expected_rewards': self.use_expected_rewards,
            'current_iteration': self.current_iteration,

            # Game scores
            'scores_list': self.scores_list,
            'last_n_scores': self.last_n_scores,
            'mean_scores': self.mean_scores,
            'max_score': self.max_score,
            'min_score': self.min_score,
            'best_score_board': self.best_score_board,

            # Rewards
            'total_rewards_list': self.total_rewards_list,
            'last_n_total_rewards': self.last_n_total_rewards,
            'mean_total_rewards': self.mean_total_rewards,
            'max_total_reward': self.max_total_reward,

            # Max cell value on game board
            'max_vals_list': self.max_vals_list,
            'last_n_vals': self.last_n_vals,
            'mean_vals': self.mean_vals,
            'max_val': self.max_val,

```

```

        'best_val_board': self.best_val_board,

# Number of steps per episode
        'max_steps_list': self.max_steps_list,
        'last_n_steps': self.last_n_steps,
        'mean_steps': self.mean_steps,
        'max_steps': self.max_steps,
        'total_steps': self.total_steps,

        'actions_avg_list': self.actions_avg_list,
        'actions_deque': self.actions_deque,
# Replay buffer
        'memory': self.memory.dump(),
# Initialize time step
        't_step': self.t_step,
        'steps_ahead': self.steps_ahead
    }

    with open(base_dir+'agent_state_%s.pkl' % name, 'wb') as f:
        pickle.dump(state, f)

    def step(self, state, action, reward, next_state, done, error,
action_dist):
        # Save experience in replay memory
        self.memory.add(state, action, reward, next_state, done, error,
action_dist, None)

    def act(self, state, eps=0.):
        """Returns actions for given state as per current policy.

        Params
        =====
            state (array_like): current state
            eps (float): epsilon, for epsilon-greedy action selection
        """
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        return action_values.cpu().data.numpy()

    def learn(self, learn_iterations, mode='board_max', save_loss=True):

        if self.use_expected_rewards:
            self.memory.calc_expected_rewards(self.steps_ahead)

        self.memory.add_episode_experiences()

        losses = []

        if len(self.memory) > self.batch_size:
            for i in range(learn_iterations):

                states, actions, rewards, next_states, dones =
self.memory.sample(mode=mode)

                # Get expected Q values from local model
                Q_expected = self.qnetwork_local(states).gather(1, actions)

                # Compute loss
                loss = F.mse_loss(Q_expected, rewards)

```

```

        losses.append(loss.detach().numpy())

        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    self.lr_decay.step()

    if save_loss:
        self.losses.append(np.mean(losses))
    else:
        self.losses.append(0)

class ReplayBuffer:
    """Fixed-size buffer to store experience."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.

        Params
        =====
        action_size (int): dimension of each action
        buffer_size (int): maximum size of buffer
        batch_size (int): size of each training batch
        seed (int): random seed
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)

        self.episode_memory = []
        self.batch_size = batch_size

        self.seed = random.seed(seed)
        self.experience = namedtuple("Experience", field_names=["state",
"action", "reward", "next_state", "done", "error", "action_dist", "weight"])

    def dump(self):
        # Saves the buffer into dict object
        d = {
            'action_size': self.action_size,
            'batch_size': self.batch_size,
            'seed': self.seed
        }

        d['memory'] = [d._asdict() for d in self.memory]
        return d

    def load(self, d):
        # creates a new buffer from dict
        self.action_size = d['action_size']
        self.batch_size = d['batch_size']
        self.seed = d['seed']

        for e in d['memory']:
            self.memory.append(self.experience(**e))

    def add(self, state, action, reward, next_state, done, error,
action_dist, weight=None):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done, error,
action_dist, weight)

```

```

        self.episode_memory.append(e)

    def add_episode_experiences(self):
        self.memory.extend(self.episode_memory)
        self.episode_memory = []

    def calc_expected_rewards(self, steps_ahead=1):
        rewards = [e.reward for e in self.episode_memory if e is not None]
        exp_rewards = [np.sum(rewards[i:i+steps_ahead]) for i in
range(len(rewards) - steps_ahead)]

        temp_memory = []

        for i, e in enumerate(self.episode_memory[:-steps_ahead]):
            t_e = self.experience(e.state, e.action, exp_rewards[i],
e.next_state, e.done, e.error, e.action_dist, None)
            temp_memory.append(t_e)

        self.episode_memory = temp_memory

    def sample(self, mode='board_max'):
        """Randomly sample a batch of experiences from memory."""

        if mode == 'random':
            experiences = random.sample(self.memory, k=self.batch_size)
        elif mode == 'board_max':
            probs = np.array([e.state.max() for e in self.memory])
            probs = probs / probs.sum()
            idx = np.random.choice(len(self.memory), size=self.batch_size,
p=probs)

            experiences = deque(maxlen=self.batch_size)
            for i in idx:
                experiences.append(self.memory[i])

            states = torch.from_numpy(np.vstack([e.state for e in experiences if
e is not None])).float().to(device)
            actions = torch.from_numpy(np.vstack([e.action for e in experiences
if e is not None])).long().to(device)
            rewards = torch.from_numpy(np.vstack([e.reward for e in experiences
if e is not None])).float().to(device)
            next_states = torch.from_numpy(np.vstack([e.next_state for e in
experiences if e is not None])).float().to(device)
            dones = torch.from_numpy(np.vstack([e.done for e in experiences if e
is not None])).astype(np.uint8).float().to(device)

            return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)

```

Код agent.py

```

import torch
import torch.nn as nn
import torch.nn.functional as F

class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=64,
fc2_units=64, fc3_units = 64):

```

```

"""Initialize parameters and build model.
Params
=====
state_size (int): Dimension of each state
action_size (int): Dimension of each action
seed (int): Random seed
fc1_units (int): Number of nodes in first hidden layer
fc2_units (int): Number of nodes in second hidden layer
fc3_units (int): Number of nodes in third hidden layer

"""
super(QNetwork, self).__init__()
self.seed = torch.manual_seed(seed)

self.fc1 = nn.Linear(state_size, fc1_units)
self.bn1 = nn.BatchNorm1d(fc1_units)
self.act1 = nn.ReLU()
self.fc2 = nn.Linear(fc1_units, fc2_units)
self.bn2 = nn.BatchNorm1d(fc2_units)
self.act2 = nn.ReLU()
self.fc3 = nn.Linear(fc2_units, fc3_units)
self.bn3 = nn.BatchNorm1d(fc3_units)
self.act3 = nn.ReLU()
self.fc4 = nn.Linear(fc3_units, action_size)

def forward(self, state):
    """Build a network that maps state -> action values."""

    x = self.fc1(state)
    x = self.act1(x)
    x = self.bn1(x)
    x = self.fc2(x)
    x = self.act2(x)
    x = self.bn2(x)
    x = self.fc3(x)
    x = self.act3(x)
    return self.fc4(x)

```

Код model.py

```

import numpy as np
import matplotlib.pyplot as plt
from game_cpp import Game # Импортируем библиотеку из game.cpp
from collections import deque
from agent import Agent
import time
import torch
import datetime
import random
from IPython.display import clear_output
import pickle
import os

# Deep Q-Learning function

def transform_state(state, mode='plain'):
    """ Returns the (log2 / 17) of the values in the state array """

    if mode == 'plain':
        return np.reshape(state, -1)

    elif mode == 'plain_hw':

```

```

        return np.concatenate([np.reshape(state, -1),
np.reshape(np.transpose(state), -1)])

    elif mode == 'log2':
        state = np.reshape(state, -1)
        state[state == 0] = 1
        return np.log2(state) / 17

    elif mode == 'one_hot':
        state = np.reshape(state, -1)
        state[state == 0] = 1
        state = np.log2(state)
        state = state.astype(int)
        new_state = np.reshape(np.eye(18)[state], -1)
        return new_state
    else:
        return state

def dqn(n_episodes=100, eps_start=0.05, eps_end=0.001, eps_decay=0.995,
step_penalty=0, sample_mode='error',
start_learn_iterations=20):
    """Deep Q-Learning.

    Params
    =====
        n_episodes (int): maximum number of training episodes
        eps_start (float): starting value of epsilon, for epsilon-greedy
action selection
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing
epsilon
        step_penalty (int): if we want to deduct some points from the reward
for taking a step, put it here
        sample_mode (str): how to assign sample probabilities for experiences
in the replay buffer
        learn_iterations (int): number of learning iterations after each
episode
    """

    eps = eps_start
    starting_iteration = agent.current_iteration
    best_game_history, worst_game_history = load_game_history(version)
    learn_iterations = start_learn_iterations

    # main loop
    for i_episode in range(1, n_episodes + 1):
        agent.current_iteration = agent.current_iteration + 1
        time_start = time.time()

        # keep track of different actions taken per episode
        actions = np.array([0, 0, 0, 0])

        # Starting with a fresh board
        env.reset(2, step_penalty) # reset environment

        state = transform_state(env.current_state(), mode='one_hot') # get
the current state
        reward = env.get_reward() # get the reward
        total_rewards = reward # initialize total rewards
        score = env.get_score() # initialize the score
        agent.total_steps = 0

        # keep playing

```



```

while not env.get_done():
    reward = env.get_negative_reward()

    action_values = agent.act(state) # select an action

    actions_sorted = [(i, v) for i, v in enumerate(action_values[0])]
    actions_sorted = sorted(actions_sorted, key=lambda x: x[1],
reverse=True)

    random_action = random.choice(np.arange(agent.action_size))
    action_idx = 0
    env.set_moved(False)
    while not env.get_moved():

        if random.random() < eps:
            action_elem = actions_sorted[random_action]
        else:
            action_elem = actions_sorted[action_idx]
            action_idx += 1

        action = np.int64(action_elem[0])
        actions[action] += 1
        _, reward, done = env.step(action, action_values) # send the
action to environment
        next_state = transform_state(env.current_state(),
mode='one_hot') # get the current state
        reward = env.get_reward() # get the reward

        # save the results of the step

        error = np.abs(reward - action_elem[1]) ** 2
        score = env.get_score()
        done = env.get_done() # check if episode has finished

        # learning step
        if len(agent.actions_avg_list) > 0:
            actions_dist = [np.mean(agent.actions_deque[i]) for i in
range(4)][action]
        else:
            actions_dist = (actions / np.sum(actions))[action]

        # Save this experience
        agent.step(state, action, reward, next_state, done, error,
actions_dist)

        state = next_state

        agent.total_steps += 1
        total_rewards += reward

        if done:
            break

    # Do the actual learning
    agent.learn(learn_iterations, mode=sample_mode, save_loss=True,
weight=env.get_score())

    # Calculate action stats
    actions = actions / env.get_steps()

    agent.actions_deque[0].append(actions[0])
    agent.actions_deque[1].append(actions[1])
    agent.actions_deque[2].append(actions[2])

```

```

agent.actions_deque[3].append(actions[3])

agent.actions_avg_list.append([np.mean(agent.actions_deque[i]) for i
in range(4)])

# Here we keep track of the learning progress and save the best
values
if total_rewards > agent.max_total_reward:
    agent.max_total_reward = total_rewards

if score > agent.max_score:
    agent.max_score = score
    agent.best_score_board = env.current_state().copy()
    best_game_history = env.get_history().copy()

if score < agent.min_score:
    agent.min_score = score
    worst_game_history = env.get_history().copy()

if env.get_score() > agent.max_val:
    agent.max_val = env.get_score()
    agent.best_val_board = env.current_state().copy()

if env.get_steps() > agent.max_steps:
    agent.max_steps = env.get_steps()
    agent.best_steps_board = env.current_state().copy()

agent.total_rewards_list.append(total_rewards)
agent.scores_list.append(score) # save most recent score to total
agent.max_vals_list.append(env.get_score())
agent.max_steps_list.append(env.get_steps())

agent.last_n_scores.append(score)
agent.last_n_steps.append(env.get_steps())
agent.last_n_vals.append(env.get_score())
agent.last_n_total_rewards.append(total_rewards)

agent.mean_scores.append(np.mean(agent.last_n_scores))
agent.mean_steps.append(np.mean(agent.last_n_steps))
agent.mean_vals.append(np.mean(agent.last_n_vals))
agent.mean_total_rewards.append(np.mean(agent.last_n_total_rewards))

time_end = time.time()

# Increasing the epsilon every N episodes in order to allow for some
exploration
if agent.current_iteration % 5000 == 0:
    eps = eps * 2
else:
    eps = max(eps_end, eps_decay * eps) # decrease epsilon

# Display training stats
if agent.current_iteration % 100 == 0:
    clear_output()

    # Training metrics
    fig, ax1 = plt.subplots()
    fig.set_size_inches(16, 6)
    ax1.plot(agent.max_vals_list + [None for i in range(10000 -
len(agent.scores_list))],
            label='Max cell value seen on board', alpha=0.3,
color='g')

```

```

        ax1.plot(agent.mean_steps + [None for i in range(10000 -
len(agent.scores_list))],
                label='Mean steps over last 50 episodes', color='b')

        ax2 = ax1.twinx() # instantiate a second axes that shares the
same x-axis

        ax2.plot(agent.mean_total_rewards + [None for i in range(10000 -
len(agent.scores_list))],
                label='Mean total rewards over last 50 episodes',
color='r')
        fig.tight_layout() # otherwise the right y-label is slightly
clipped

        plt.xlabel('Episode #')
        handles, labels = [(a + b) for a, b in
zip(ax1.get_legend_handles_labels(), ax2.get_legend_handles_labels())]
        plt.legend(handles, labels)
        plt.show()

        plt.figure(figsize=(16, 6))
        plt.title('Loss')
        plt.plot(agent.losses)
        plt.yscale('log')
        plt.show()

        # Averaged actions stats
        plt.figure(figsize=(16, 6))
        plt.title('Averaged actions distribution per game')
        a_list = np.array(agent.actions_avg_list).T

        plt.stackplot([i for i in range(1, len(agent.actions_avg_list) +
1)], a_list[0], a_list[1], a_list[2],
                    a_list[3],
                    labels=['Up %0.2f' % (agent.actions_avg_list[-1][0]
* 100),
                        'Down %0.2f' % (agent.actions_avg_list[-
1][1] * 100),
                        'Left %0.2f' % (agent.actions_avg_list[-
1][2] * 100),
                        'Right %0.2f' % (agent.actions_avg_list[-
1][3] * 100)])
        plt.legend()
        plt.show()

        # Display the board with the best score
        env.draw_board(agent.best_score_board, 'Best score board')

        # Save the model and the game history
        save_state(version, eps)
        save_game_history(version, best_game_history, worst_game_history)

        s = '%d/%d | %0.2fs | Sc:%d | AvgSc:%d | TR:%d | AvgTR:%d |
G1MaxVal:%d' % \
            (agent.current_iteration, starting_iteration + n_episodes,
time_end - time_start, score,
            np.mean(agent.last_n_scores), total_rewards,
            np.mean(agent.last_n_total_rewards),
            np.max(agent.max_vals_list))

        s = s + ' ' * (120 - len(s))
        print(s, end='\r')

base_dir = './data'

```

```

def save_state(name, eps):
    with open(base_dir + '/game_%s.pkl' % name, 'wb') as f:
        state = {
            'env': env,
            'last_eps': eps
        }
        pickle.dump(state, f)
    agent.save(name)

def save_game_history(name, best_history, worst_history):
    with open(base_dir + '/best_game_history_%s.pkl' % name, 'wb') as f:
        pickle.dump(best_history, f)
    with open(base_dir + '/worst_game_history_%s.pkl' % name, 'wb') as f:
        pickle.dump(worst_history, f)

def load_game_history(name):
    best_history = []
    worst_history = []
    if os.path.exists(base_dir + '/best_game_history_%s.pkl' % name):
        with open(base_dir + '/best_game_history_%s.pkl' % name, 'rb') as f:
            best_history = pickle.load(f)

    if os.path.exists(base_dir + '/worst_game_history_%s.pkl' % name):
        with open(base_dir + '/worst_game_history_%s.pkl' % name, 'rb') as f:
            worst_history = pickle.load(f)

    return best_history, worst_history

# Create the environment with 4x4 board
version = 'ml_model_2048_ohe_2step_penalty_logreward_512x3_random'
env = Game(4, reward_mode='log2', negative_reward=-3, cell_move_penalty=0.1)
eps = 0.5

# Create the agent, duplicating default values for visibility
state_size = (env.get_size()) ** 2 # total number of cells
action_size = 4 # number of available actions

agent = Agent(state_size=state_size * 18, action_size=action_size,
              seed=42, fcl_units=1024, fc2_units=1024, fc3_units=1024,
              buffer_size=10000, batch_size=1024, lr=0.004,
              use_expected_rewards=True, predict_steps=2,
              gamma=0., tau=0.001)

# Run the training
dqn(n_episodes=100000,
    eps_start=eps or 0.05,
    eps_end=0.00001,
    eps_decay=0.999,
    step_penalty=0,
    sample_mode='random',
    start_learn_iterations=10)

```

Код training.py