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

Институт Информационных Технологий

Кафедра Вычислительной Техники

ПРАКТИЧЕСКАЯ РАБОТА №4

по дисциплине
«Проектирование интеллектуальных систем (часть 2/2)»

Студент группы:ИКБО-04-22 __Кликушин В.И._
(Ф. И.О. студента)

Преподаватель __Холмогоров В.В.__
(Ф.И.О. преподавателя)

Москва 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3
1 ПОСТАНОВКА ЗАДАЧИ 4
2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 5
2.1 Формализация задачи RL 5
2.1.1 Марковский процесс принятий решений 5
2.1.2 Политика агента 6
2.1.3 Траектория и распределение траекторий 7
2.1.4 Дисконтированная кумулятивная награда и функция ценности 7
2.1.5 Цель обучения 8
2.2 Классификация RL-алгоритмов 8
2.2.1 По наличию модели среды 8
2.2.2 По способу сбора и использования данных 9
2.2.3 По методу оптимизации политики 10
3 ПРАКТИЧЕСКАЯ ЧАСТЬ 11
3.1 Постановка практической задачи 11
3.2 Реализация среды взаимодействия 11
3.3 Архитектура нейронной сети 12
3.4 Реализация агента 12
3.5 Процесс обучения 12
ЗАКЛЮЧЕНИЕ 14
СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 15
ПРИЛОЖЕНИЯ 16

ВВЕДЕНИЕ

В современном мире информационных технологий искусственный интеллект и машинное обучение занимают одно из центральных мест, находя применение в самых разнообразных областях — от робототехники и автономных систем до анализа данных и создания адаптивных пользовательских интерфейсов. Среди множества парадигм машинного обучения особое место занимает обучение с подкреплением, в рамках которого агент учится оптимально действовать в окружающей среде методом проб и ошибок, максимизируя получаемую кумулятивную награду. Этот подход особенно эффективен в задачах, где заранее неизвестны правильные действия, но есть возможность оценить качество поведения через систему вознаграждений и штрафов.

1 ПОСТАНОВКА ЗАДАЧИ

Цель работы: приобрести навыки реализации генетических алгоритмов.

Задачи: создать программную реализацию генетического алгоритма, который способен решать задачу выбранного варианта (для этого она должна быть описана математически), либо находить решение одной из существующих задач или проблем математики, алгоритм должен иметь следующий минимальный стек:

– оптимизированные для решаемой задачи входные данные, образующие начальную популяцию;

– тренировочную и тестовые выборки (может и не быть, нужны для самостоятельного переобучения и отслеживания пользователем степени обучения);

– loss-функцию и метрику точности (может и не быть, если ожидаемые выходные данные неизвестны);

– фитнес-функцию и алгоритм отбора;

– случайное скрещивание и случайную мутацию.

В качестве дополнительных элементов генетического алгоритма реализовать:

– стратегии элитизма, рулетки и/или турнира (наиболее простые из существующих алгоритмов отбора);

– графики истории и результатов обучения;

– сравнительный анализ собственной стратегии из пункта 1 и стратегий из пункта 2 (в том числе между собой);

– графики полученных и ожидаемых (если они есть) результатов.

2 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Обучение с подкреплением представляет собой один из трёх основных парадигм машинного обучения наряду с обучением с учителем и обучением без учителя. В отличие от этих подходов, RL ориентирован на решение задач, в которых агент должен научиться действовать в окружающей среде, максимизируя награду, получаемую в процессе взаимодействия. Ключевая особенность RL заключается в том, что обучающийся агент не располагает готовыми примерами правильного поведения, а должен самостоятельно исследовать среду, комбинируя метод проб и ошибок с анализом долгосрочных последствий своих действий.

Этот подход находит применение в самых разнообразных областях — от игровых систем и робототехники до управления ресурсами и адаптивных пользовательских интерфейсов. Математической основой для формализации задач RL служит теория марковских процессов принятия решений.

2.1 Формализация задачи RL

2.1.1 Марковский процесс принятий решений

MDP является фундаментальной моделью, используемой для описания взаимодействия агента со средой в RL. Формально MDP задаётся кортежем из пяти элементов: (S, A, P, r, γ).

Пространство состояний S представляет собой множество всех возможных конфигураций среды. В зависимости от задачи, состояние может быть дискретным или непрерывным. Важным предположением является полная наблюдаемость: агенту доступно полное описание текущего состояния.

Пространство действий A включает все действия, которые агент может предпринять. Как и состояния, действия могут быть дискретными или непрерывными.

Функция переходов $P(s' \mid s, a)$ определяет динамику среды. Для каждого состояния s и действия a она задаёт распределение вероятностей над следующими состояниями s'. В детерминированных средах это распределение вырождено, и следующее состояние однозначно определяется парой (s, a).

Функция награды $r(s, a)$ присваивает скалярное значение каждому переходу, обозначающее немедленную «полезность» выполнения действия a в состоянии s. Награда может быть положительной (поощрение) или отрицательной (штраф). Дизайн функции награды — критический аспект постановки RL-задачи, поскольку именно она кодирует цель обучения.

Коэффициент дисконтирования $\gamma \in [0,1]$ вводится для обеспечения сходимости бесконечных сумм и моделирования предпочтения более ранних наград. При $\gamma = 1$ все будущие награды учитываются в полной мере; при $\gamma < 1$ награды, полученные в отдалённом будущем, дисконтируются.

Свойство Марковости утверждает, что следующее состояние и награда зависят только от текущего состояния и выбранного действия, а не от всей предыстории. Это свойство позволяет значительно упростить анализ и алгоритмы RL.

2.1.2 Политика агента

Политика π определяет поведение агента. В общем случае политика является стохастической и задаётся как условное распределение $\pi(a \mid s)$ над действиями для каждого состояния. Детерминированная политика представляет собой частный случай, где для каждого состояния выбирается одно конкретное действие.

Цель обучения с подкреплением состоит в нахождении оптимальной политики π^*, которая максимизирует ожидаемую кумулятивную награду. Поиск такой политики осложняется тем, что действия влияют не только на немедленное вознаграждение, но и на последующие состояния, а значит, и на все будущие награды.

2.1.3 Траектория и распределение траекторий

Взаимодействие агента со средой порождает траекторию — последовательность состояний, действий и наград (Формула 2.1).

$$T = (s_0, a_0, r_0, s_1, a_1, r_1, \dots) \tag{2.1}$$

Для заданной политики π и начального состояния s_0 распределение вероятностей над траекториями определяется по Формуле 2.2.

$$p(T) = \prod_{t \geq 0} \pi(a_t | s_t) \cdot p(s_{t+1} | s_t, a_t) \tag{2.2}$$

Это распределение учитывает как стохастичность политики, так и стохастичность среды.

2.1.4 Дисконтированная кумулятивная награда и функция ценности

Для оценки качества политики вводится понятие дисконтированной кумулятивной награды (также называемой возвратом) для траектории (Формула 2.3).

$$R(T) = \sum_{t=0}^{\infty} \gamma^t \, r(s_t, a_t) \tag{2.3}$$

В случае эпизодических задач (где траектория конечна) сумма берётся до момента достижения терминального состояния.

Функция ценности состояния $V^{\pi}(s)$ определяется как ожидаемый возврат при старте из состояния s и следовании политике π (Формула 2.4).

$$V^{\pi}(s) = \mathbb{E}_{T \sim \pi}\left[\sum_{k=0}^{\infty} \gamma^k \, r_{t+k} \mid s_t = s \right] \tag{2.4}$$

Функция ценности действия $Q^{\pi}(s,a)$ — это ожидаемый возврат после выполнения действия a в состоянии s и последующего следования политике π (Формула 2.5).

$$Q^{\pi}(s,a) = \mathbb{E}_{T\sim\pi}\left[\sum_{k=0}^{\infty}\gamma^{k} r_{t+k} \mid s_t = s, a_t = a\right] \quad (2.5)$$

Эти функции играют центральную роль во многих RL-алгоритмах, поскольку позволяют оценивать долгосрочные последствия действий без явного моделирования будущего.

2.1.5 Цель обучения

Формально задача RL состоит в нахождении политики π^{*}, максимизирующей ожидаемую дисконтированную награду (Формула 2.6).

$$J(\pi) = \mathbb{E}_{T\sim\pi}[R(T)] \rightarrow \max_{\pi}$$

В случае табличного MDP с конечными пространствами состояний и действий существует по крайней мере одна детерминированная оптимальная политика. Однако в общем случае поиск оптимальной политики представляет собой сложную задачу оптимизации в высокоразмерном пространстве.

2.2 Классификация RL-алгоритмов

Современные алгоритмы RL можно классифицировать по нескольким ключевым признакам, что отражает разнообразие подходов к решению задачи оптимизации политики.

2.2.1 По наличию модели среды

Model-based алгоритмы используют явную модель динамики среды $P(s' \mid s, a)$ и функции награды $r(s, a)$. Такая модель может быть задана априори или обучена в процессе взаимодействия со средой. Наличие модели позволяет агенту планировать — то есть моделировать возможные последовательности событий без реального взаимодействия со средой. Это может значительно повысить эффективность, но требует дополнительных вычислительных ресурсов и может быть подвержено ошибкам, если модель неточна.

Model-free алгоритмы не строят и не используют явную модель среды. Вместо этого они обучаются напрямую на опыте взаимодействия, оценивая функцию ценности и/или оптимизируя политику. Хотя такие алгоритмы обычно требуют больше данных, они проще в реализации и более устойчивы к неточностям модели. Большинство современных успешных применений RL, особенно с использованием глубоких нейронных сетей, основаны на model-free подходах.

2.2.2 По способу сбора и использования данных

On-policy алгоритмы требуют, чтобы данные для обучения собирались с использованием текущей оптимизируемой политики. Это означает, что политика, которую алгоритм пытается улучшить, также используется для взаимодействия со средой. Примерами являются алгоритмы SARSA и методы градиента политики, такие как REINFORCE и PPO. On-policy подходы обеспечивают согласованность между данными и оцениваемой политикой, но могут быть неэффективными с точки зрения использования данных, поскольку старый опыт становится бесполезным после обновления политики.

Off-policy алгоритмы способны обучаться на данных, собранных с использованием другой политики. Это позволяет более эффективно использовать данные, переиспользуя опыт, и осуществлять обучение в режиме, когда сбор данных и оптимизация политики происходят асинхронно. Яркими примерами являются Q-learning и его глубокие варианты (DQN), а также

алгоритмы, основанные на актор-критической архитектуре с буфером воспроизведения.

2.2.3 По методу оптимизации политики

Value-based подходы фокусируются на обучении функции ценности, а затем выводят оптимальную политику как жадную относительно этой функции. Основная идея заключается в использовании уравнений оптимальности Беллмана, которые связывают значения соседних состояний. Алгоритмы этого класса, такие как Q-learning и его глубокие расширения, доказали свою эффективность во многих задачах с дискретными действиями.

Policy-based подходы напрямую параметризуют политику и оптимизируют её параметры, используя оценку градиента целевой функции $J(\pi)$. Методы градиента политики, такие как REINFORCE, позволяют работать с непрерывными пространствами действий и могут обучать стохастические политики. Однако они часто страдают от высокой дисперсии оценок градиента.

Актор-критические методы объединяют достоинства value-based и policy-based подходов. В них используется две модели: актор (actor), который представляет политику, и критик (critic), который оценивает функцию ценности. Критик используется для уменьшения дисперсии оценок градиента актора, что ускоряет и стабилизирует обучение.

Мета-эвристики и эволюционные алгоритмы рассматривают задачу оптимизации политики как проблему чёрного ящика и используют методы, не требующие градиентов, такие как генетические алгоритмы или методы случайного поиска. Хотя они могут быть очень неэффективными по данным, они просты в реализации и иногда оказываются полезными в задачах с разрывными или очень шумными функциями награды.

3 ПРАКТИЧЕСКАЯ ЧАСТЬ

3.1 Постановка практической задачи

В рамках данной работы была реализована интеллектуальная система на основе обучения с подкреплением, способная обучаться игре «Змейка». Задача агента — научиться управлять змейкой на дискретном поле размером 10×10 клеток так, чтобы собирать еду, избегая столкновений с границами поля и собственным телом, максимизируя при этом итоговый счёт.

3.2 Реализация среды взаимодействия

Среда SnakeGame реализована в файле env.py (Приложение А) и предоставляет следующий функционал:

Инициализация игры. Змейка начинается с центра поля. Направление движения — вправо. Еда генерируется в случайной свободной клетке.

Представление состояния. Состояние среды кодируется в виде вектора из 12 нормализованных признаков: нормализованные координаты головы змейки; нормализованные координаты еды; нормализованные расстояния до еды по осям X и Y; one-hot кодирование текущего направления движения (4 признака); признак опасности впереди; нормализованная длина змейки.

Система наград:

- +10.0 за съедание еды;
- –0.05 за каждый шаг без еды;
- –10.0 за столкновение со стеной или с собой;
- +0.1 за приближение к еде;
- –5.0 и завершение эпизода при безрезультатном блуждании (>20 шагов без еды).

Визуализация. Реализован режим рендеринга с использованием библиотеки Pygame для отображения процесса обучения и тестирования.

3.3 Архитектура нейронной сети

В файле model.py (Приложение Б) определена архитектура Q-сети (QNetwork), используемой в DQN-агенте: входной слой принимает вектор состояния размерности 12; скрытые слои включают полносвязный слой на 128 нейронов с функцией активации ReLU, второй полносвязный слой на 128 нейронов с ReLU и третий полносвязный слой на 64 нейрона с ReLU; выходной слой содержит 3 нейрона (действия: вперед, направо, налево) без функции активации (линейный).

3.4 Реализация агента

Агент (DQNAgent), описанный в agent.py (Приложение В), реализует алгоритм Deep Q-Network с улучшениями:

Две нейронные сети:

- policy_net — для выбора действий;
- target_net — для расчёта целевых Q-значений.

3.5 Процесс обучения

Обучение осуществляется в файле train.py (Приложение Г): 500 эпизодов (по умолчанию); рендеринг каждого 50-го эпизода для наблюдения за прогрессом; сбор статистики — счёт за эпизод, средние потери, текущее значение ε; логирование — каждые 10 эпизодов выводится информация о прогрессе.

Визуализация процесса обучения представлена на Рисунке 3.1.

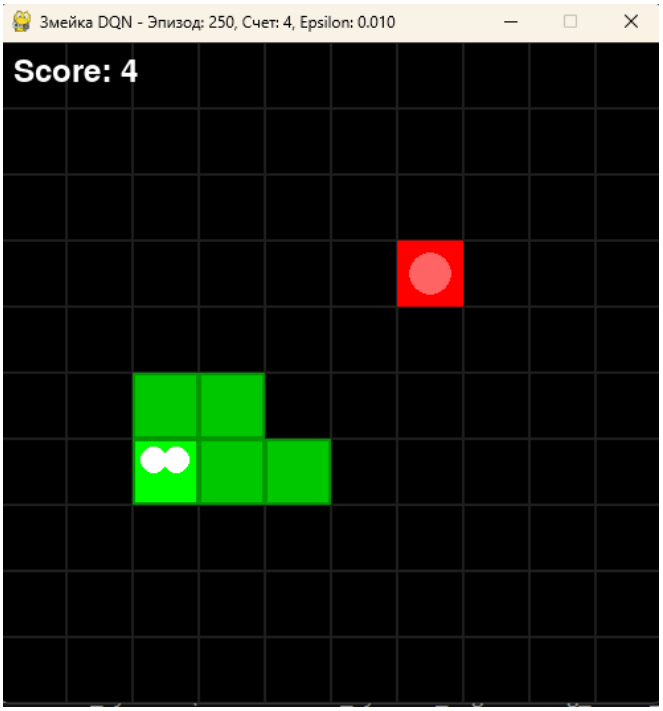

Рисунок 3.1 – Визуализация процесса обучения

Наблюдаемая динамика обучения объясняется особенностями алгоритма Deep Q-Learning и среды «Змейка». На начальных этапах обучения агент действует преимущественно случайно из-за высокого значения параметра ε, что приводит к низким значениям награды. По мере накопления опыта в буфере воспроизведения и уменьшения ε агент начинает извлекать полезные закономерности, что выражается в устойчивом росте среднего счёта. Колебания результатов отдельных эпизодов связаны со стохастичностью среды.

Логирование процесса обучения представлено на Рисунке 3.2.

```
Эпизод:   10/500 | Счет:  0 | Средний счет (10 эп.):  0.10 | Epsilon: 0.878 | Память: 155
Эпизод:   20/500 | Счет:  0 | Средний счет (10 эп.):  0.20 | Epsilon: 0.718 | Память: 299
Эпизод:   30/500 | Счет:  0 | Средний счет (10 эп.):  0.20 | Epsilon: 0.579 | Память: 451
Эпизод:   40/500 | Счет:  0 | Средний счет (10 эп.):  0.10 | Epsilon: 0.481 | Память: 586
Эпизод:   50/500 | Счет:  0 | Средний счет (10 эп.):  0.10 | Epsilon: 0.376 | Память: 761
Эпизод:   60/500 | Счет:  0 | Средний счет (10 эп.):  0.10 | Epsilon: 0.313 | Память: 891
Эпизод:   70/500 | Счет:  0 | Средний счет (10 эп.):  0.10 | Epsilon: 0.249 | Память: 1054
Эпизод:   80/500 | Счет:  0 | Средний счет (10 эп.):  0.10 | Epsilon: 0.192 | Память: 1236
Эпизод:   90/500 | Счет:  0 | Средний счет (10 эп.):  0.20 | Epsilon: 0.152 | Память: 1402
Эпизод:  100/500 | Счет:  0 | Средний счет (10 эп.):  0.10 | Epsilon: 0.123 | Память: 1545
Эпизод:  110/500 | Счет:  0 | Средний счет (10 эп.):  0.20 | Epsilon: 0.096 | Память: 1721
Эпизод:  120/500 | Счет:  0 | Средний счет (10 эп.):  0.60 | Epsilon: 0.068 | Память: 1963
Эпизод:  130/500 | Счет:  1 | Средний счет (10 эп.):  1.00 | Epsilon: 0.048 | Память: 2220
Эпизод:  140/500 | Счет:  0 | Средний счет (10 эп.):  0.70 | Epsilon: 0.033 | Память: 2493
Эпизод:  150/500 | Счет:  0 | Средний счет (10 эп.):  0.30 | Epsilon: 0.024 | Память: 2725
Эпизод:  160/500 | Счет:  0 | Средний счет (10 эп.):  1.10 | Epsilon: 0.016 | Память: 3061
Эпизод:  170/500 | Счет:  0 | Средний счет (10 эп.):  1.10 | Epsilon: 0.010 | Память: 3360
Эпизод:  180/500 | Счет:  1 | Средний счет (10 эп.):  0.50 | Epsilon: 0.010 | Память: 3618
Эпизод:  190/500 | Счет:  1 | Средний счет (10 эп.):  0.90 | Epsilon: 0.010 | Память: 3930
Эпизод:  200/500 | Счет:  3 | Средний счет (10 эп.):  1.80 | Epsilon: 0.010 | Память: 4285
Эпизод:  210/500 | Счет:  0 | Средний счет (10 эп.):  1.80 | Epsilon: 0.010 | Память: 4634
Эпизод:  220/500 | Счет:  6 | Средний счет (10 эп.):  1.00 | Epsilon: 0.010 | Память: 4929
Эпизод:  230/500 | Счет:  5 | Средний счет (10 эп.):  1.70 | Epsilon: 0.010 | Память: 5258
Эпизод:  240/500 | Счет:  4 | Средний счет (10 эп.):  1.90 | Epsilon: 0.010 | Память: 5619
Эпизод:  250/500 | Счет:  0 | Средний счет (10 эп.):  0.90 | Epsilon: 0.010 | Память: 5914
Эпизод:  260/500 | Счет:  0 | Средний счет (10 эп.):  1.80 | Epsilon: 0.010 | Память: 6232
Эпизод:  270/500 | Счет:  4 | Средний счет (10 эп.):  1.60 | Epsilon: 0.010 | Память: 6580
Эпизод:  280/500 | Счет:  4 | Средний счет (10 эп.):  1.60 | Epsilon: 0.010 | Память: 6946
```

Рисунок 3.2 – Логирование процесса обучения

ЗАКЛЮЧЕНИЕ

В ходе выполнения практической работы успешно реализована интеллектуальная система на основе обучения с подкреплением, способная автономно обучаться игре «Змейка». Работа включала теоретическое исследование основ генетических алгоритмов и методов RL, а также практическую реализацию среды взаимодействия, нейронной сети и агента с использованием алгоритма Deep Q-Network. В процессе обучения агент продемонстрировал устойчивый прогресс, научившись эффективно собирать еду и избегать столкновений, что подтвердило эффективность выбранного подхода для решения задач управления в дискретной среде.

Приобретённые навыки включают практическое применение теории обучения с подкреплением, разработку и отладку интерактивных сред, реализацию глубоких нейронных сетей с использованием PyTorch, а также анализ и визуализацию процесса обучения. Результаты работы имеют практическую значимость и могут служить основой для создания более сложных автономных систем в областях робототехники, управления ресурсами и адаптивного планирования.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Сорокин, А. Б. Безусловная оптимизация. [Электронный ресурс] : учебно-метод. пособие / А. Б. Сорокин, О. В. Платонова, Л. М. Железняк — М. РТУ МИРЭА , 2020.
2. Сорокин, А. Б. Введение в генетические алгоритмы: теория, расчеты и приложения. [Электронный ресурс] : учебно-метод. пособие / А. Б. Сорокин — М. МИРЭА , 2018.
3. Сорокин, А. Б. Введение в роевой интеллект: теория, расчеты и приложения [Электронный ресурс]: Учебно-методическое пособие / А. Б. Сорокин – Москва: Московский технологический университет (МИРЭА), 2019.
4. Генетические алгоритмы — математический аппарат. URL: https://loginom.ru/blog/ga-math (дата обращения: 10.12.2025).

ПРИЛОЖЕНИЯ

Приложение А — Код файла env.py.

Приложение Б — Код файла model.py.

Приложение В — Код файла agent.py.

Приложение Г — Код файла train.py.

Приложение А

Код файла env.py

Листинг А – Код файла env.py

```
import random
import numpy as np
import pygame

GRID_WIDTH = 10
GRID_HEIGHT = 10
BLOCK_SIZE = 50 # Размер одной клетки в пикселях
FPS = 60

# Направления движения
UP = (0, -1)
DOWN = (0, 1)
LEFT = (-1, 0)
RIGHT = (1, 0)

class SnakeGame:
    def __init__(self, render_mode=None):
        self.render_mode = render_mode # Режим отрисовки (None для обучения без графики и "human" для визуализации)
        if render_mode == "human":
            pygame.init()
            self.screen = pygame.display.set_mode(
                (GRID_WIDTH * BLOCK_SIZE, GRID_HEIGHT * BLOCK_SIZE)
            )
            self.clock = pygame.time.Clock() # Объект для контроля FPS
            self.font = pygame.font.Font(None, 36) # Шрифт для отображения счёта

        self.reset() # Инициализация начального состояния среды

    def reset(self):
        self.snake = [(GRID_WIDTH // 2, GRID_HEIGHT // 2)] # Змейка представлена списком координат сегментов, первый элемент — голова
        self.direction = RIGHT # Начальное направление движения
        self.food = self._generate_food() # Создание еды в случайной позиции
        self.score = 0
        self.done = False # терминальное состояние эпизода
        self.steps_without_food = 0 # контроль за зацикливанием
        return self.get_state()

    def _generate_food(self):
        while True: # Генерируем до тех пор, пока еда не окажется не на змейке.
            food = (
                random.randint(0, GRID_WIDTH - 1),
                random.randint(0, GRID_HEIGHT - 1),
            )
            if food not in self.snake:
                return food

    def _danger_ahead(self):
        '''
        Проверка опасности впереди

        Возвращает:
            1.0 — впереди столкновение;
            0.0 — безопасно.
        Используется как признак состояния
```

Продолжение Листинга А

```
        '''
        head_x, head_y = self.snake[0] # координаты головы
        nx = head_x + self.direction[0] # Координаты следующей клетки
        ny = head_y + self.direction[1] # Координаты следующей клетки

        if nx < 0 or nx >= GRID_WIDTH or ny < 0 or ny >= GRID_HEIGHT: #
Столкновение со стеной
            return 1.0
        if (nx, ny) in self.snake: # Столкновение с телом
            return 1.0
        return 0.0 # Безопасное движение

    def get_state(self):
        """Формирует вектор признаков состояния"""
        head_x, head_y = self.snake[0]
        food_x, food_y = self.food

        # Базовые признаки
        state = [
            head_x / GRID_WIDTH,  # Нормализованная позиция X головы
            head_y / GRID_HEIGHT,  # Нормализованная позиция Y головы
            food_x / GRID_WIDTH,  # Нормализованная позиция X еды
            food_y / GRID_HEIGHT,  # Нормализованная позиция Y еды
            # Расстояние до еды
            abs(head_x - food_x) / GRID_WIDTH,
            abs(head_y - food_y) / GRID_HEIGHT,
            # Направление движения (one-hot encoding)
            1 if self.direction == UP else 0,
            1 if self.direction == DOWN else 0,
            1 if self.direction == LEFT else 0,
            1 if self.direction == RIGHT else 0,
            # Опасность впереди
            self._danger_ahead(),
            # Длина змейки (нормализованная)
            len(self.snake) / (GRID_WIDTH * GRID_HEIGHT),
        ]
        return np.array(state, dtype=np.float32)

    def step(self, action):
        """Шаг среды"""
        self.steps_without_food += 1

        # Изменение направления
        if action == 0:  # Вперед (не менять направление)
            pass
        elif action == 1:  # Направо
            if self.direction == UP:
                self.direction = RIGHT
            elif self.direction == RIGHT:
                self.direction = DOWN
            elif self.direction == DOWN:
                self.direction = LEFT
            elif self.direction == LEFT:
                self.direction = UP
        elif action == 2:  # Налево
            if self.direction == UP:
                self.direction = LEFT
            elif self.direction == LEFT:
                self.direction = DOWN
            elif self.direction == DOWN:
                self.direction = RIGHT
```

Продолжение Листинга А

```
            elif self.direction == RIGHT:
                self.direction = UP

        # Движение
        head_x, head_y = self.snake[0]
        new_head = (head_x + self.direction[0], head_y + self.direction[1])

        # Проверка столкновений
        if (
            new_head[0] < 0
            or new_head[0] >= GRID_WIDTH
            or new_head[1] < 0
            or new_head[1] >= GRID_HEIGHT
        ):
            self.done = True
            return self.get_state(), -10.0, self.done, {"score": self.score}

        if new_head in self.snake:
            self.done = True
            return self.get_state(), -10.0, self.done, {"score": self.score}

        # Добавление новой головы
        self.snake.insert(0, new_head)

        # Проверка съедания еды
        reward = -0.05  # Маленький штраф за каждый шаг
        if new_head == self.food:
            self.score += 1
            self.food = self._generate_food()
            reward = 10.0  # Большая награда за еду
            self.steps_without_food = 0
        else:
            # Удаляем хвост только если не съели еду
            self.snake.pop()

        # Штраф за слишком долгое блуждание без еды
        if self.steps_without_food > 20:
            reward = -5.0
            self.done = True

        # Награда за приближение к еде
        old_head = (head_x, head_y)
        old_dist = abs(old_head[0] - self.food[0]) + abs(old_head[1] -
self.food[1])
        new_dist = abs(new_head[0] - self.food[0]) + abs(new_head[1] -
self.food[1])
        if new_dist < old_dist:
            reward += 0.1  # Небольшая награда за приближение к еде
        elif new_dist > old_dist:
            reward -= 0.1  # Небольшой штраф за удаление от еды

        return self.get_state(), reward, self.done, {"score": self.score}

    def render(self):
        """Визуализация игры"""
        if self.render_mode != "human":
            return

        self.screen.fill((0, 0, 0))

        # Рисуем сетку
```

Окончание Листинга А

```
        for x in range(GRID_WIDTH):
            for y in range(GRID_HEIGHT):
                rect = pygame.Rect(
                    x * BLOCK_SIZE, y * BLOCK_SIZE, BLOCK_SIZE, BLOCK_SIZE
                )
                pygame.draw.rect(self.screen, (30, 30, 30), rect, 1)

        # Рисуем змейку
        for i, (x, y) in enumerate(self.snake):
            color = (0, 255, 0) if i == 0 else (0, 200, 0)
            rect = pygame.Rect(x * BLOCK_SIZE, y * BLOCK_SIZE, BLOCK_SIZE,
BLOCK_SIZE)
            pygame.draw.rect(self.screen, color, rect)
            pygame.draw.rect(self.screen, (0, 150, 0), rect, 2)

            # Глаза у головы
            if i == 0:
                eye_size = BLOCK_SIZE // 5
                # Левый глаз
                eye_x = x * BLOCK_SIZE + BLOCK_SIZE // 3
                eye_y = y * BLOCK_SIZE + BLOCK_SIZE // 3
                pygame.draw.circle(
                    self.screen, (255, 255, 255), (eye_x, eye_y), eye_size
                )
                # Правый глаз
                eye_x = x * BLOCK_SIZE + 2 * BLOCK_SIZE // 3
                pygame.draw.circle(
                    self.screen, (255, 255, 255), (eye_x, eye_y), eye_size
                )

        # Рисуем еду
        food_rect = pygame.Rect(
            self.food[0] * BLOCK_SIZE,
            self.food[1] * BLOCK_SIZE,
            BLOCK_SIZE,
            BLOCK_SIZE,
        )
        pygame.draw.rect(self.screen, (255, 0, 0), food_rect)
        pygame.draw.circle(
            self.screen,
            (255, 100, 100),
            (
                self.food[0] * BLOCK_SIZE + BLOCK_SIZE // 2,
                self.food[1] * BLOCK_SIZE + BLOCK_SIZE // 2,
            ),
            BLOCK_SIZE // 3,
        )

        # Отображаем счет
        score_text = self.font.render(f"Score: {self.score}", True, (255, 255,
255))
        self.screen.blit(score_text, (10, 10))

        pygame.display.flip()
        self.clock.tick(FPS)
```

Приложение Б

Код файла model.py

Листинг Б – Код файла model.py

```
import torch.nn as nn

class QNetwork(nn.Module):
    def __init__(self, input_dim, output_dim):
        """
        Конструктор

        :param input_dim: размер вектора состояния
        :param output_dim: количество действий (вперед, вправо, влево)
        """
        super(QNetwork, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, output_dim),
        )

    def forward(self, state):  # прямой проход сети
        return self.fc(state)  # Прогон состояния через все слои
```

Приложение В

Код файла agent.py

Листинг В – Код файла agent.py

```
import random
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from collections import deque
from model import QNetwork

class DQNAgent:
    def __init__(self, state_dim, action_dim):
        '''
        Docstring для __init__

        :param state_dim: размер вектора состояния
        :param action_dim: количество возможных действий
        '''
        self.state_dim = state_dim
        self.action_dim = action_dim

        self.policy_net = QNetwork(state_dim, action_dim) # Основная сеть,
используется для выбора действий, веса обновляются при обучении
        self.target_net = QNetwork(state_dim, action_dim) # Целевая сеть,
используется для расчёта целевого значения Q_target, обеспечивает стабильность
обучения
        self.target_net.load_state_dict(self.policy_net.state_dict()) #
Копирование весов

        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=0.001) #
Оптимизатор Adam
        self.loss_fn = nn.MSELoss() # Функция потерь

        # Гиперпараметры обучения
        self.gamma = 0.99  # Коэффициент дисконтирования (учитывает будущие
награды)
        self.epsilon = 1.0  # Начальный epsilon (100% случайных действий,
стимулирует исследование среды)
        self.epsilon_min = 0.01 # гарантирует небольшую долю случайности
        self.epsilon_decay = 0.995 # коэффициент для экспоненциального
уменьшения epsilon
        self.batch_size = 64 # Размер мини-батча
        self.target_update_freq = 10  # Частота обновления целевой сети

        # Ограниченная память
        self.memory = deque(maxlen=20000) # Буфер опыта
        self.train_step_counter = 0 # Счётчик шагов обучения

    def choose_action(self, state, training=True):
        """Выбор действия с epsilon-greedy стратегией"""
        if training and np.random.random() < self.epsilon:
            return random.randint(0, self.action_dim - 1) # Случайное действие
из допустимых

        state_tensor = torch.FloatTensor(state).unsqueeze(0) # Преобразование
состояния
        with torch.no_grad(): # Отключение вычисления градиентов
            q_values = self.policy_net(state_tensor)
```

Окончание Листинга В

```
            return torch.argmax(q_values).item() # Выбираем действие с
максимальной ценностью

    def store_experience(self, state, action, reward, next_state, done):
        """Сохранение опыта в память"""
        self.memory.append((state, action, reward, next_state, done))

    def train(self):
        """Обучение на мини-батче из памяти"""
        if len(self.memory) < self.batch_size: # Обучение начинается только при
достаточном опыте
            return 0

        # Выбор случайного мини-батча
        batch = random.sample(self.memory, self.batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)

        # Конвертация в тензоры
        states = torch.FloatTensor(states)
        actions = torch.LongTensor(actions).unsqueeze(1)
        rewards = torch.FloatTensor(rewards)
        next_states = torch.FloatTensor(next_states)
        dones = torch.FloatTensor(dones)

        # Текущие Q-значения
        current_q = self.policy_net(states).gather(1, actions).squeeze()

        # Следующие Q-значения (Double DQN)
        with torch.no_grad():
            next_actions = self.policy_net(next_states).argmax(1, keepdim=True)
# Выбор действий по policy-сети
            next_q = self.target_net(next_states).gather(1,
next_actions).squeeze() # Оценка действий по target-сети
            target_q = rewards + (1 - dones) * self.gamma * next_q # Формула
Беллмана

        # Вычисление потерь
        loss = self.loss_fn(current_q, target_q)

        # Оптимизация
        self.optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(self.policy_net.parameters(), 1.0) #
Ограничение градиентов
        self.optimizer.step() # Обновление весов сети

        # Уменьшение epsilon
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

        # Обновление целевой сети
        self.train_step_counter += 1
        if self.train_step_counter % self.target_update_freq == 0:
            self.target_net.load_state_dict(self.policy_net.state_dict()) #
Копирование весов

        return loss.item()
```

Приложение Г

Код файла train.py

Листинг Г – Код файла train.py

```
import numpy as np
import pygame
from env import SnakeGame
from agent import DQNAgent

def train_agent(episodes=1000, render_every=50):
    """Функция обучения с визуализацией"""
    env = SnakeGame(render_mode="human")
    state_dim = len(env.get_state())
    action_dim = 3  # Вперед, направо, налево
    agent = DQNAgent(state_dim, action_dim)

    scores = []  # итоговый счёт за эпизод
    losses = []  # средняя функция потерь
    epsilons = []  # значение ε

    print("Начало обучения...")
    print(f"Размер состояния: {state_dim}")
    print(f"Количество действий: {action_dim}")

    for episode in range(
        episodes
    ):  # Эпизод — один полный запуск игры от reset до done=True
        state = env.reset()  # Сброс среды
        total_reward = 0  # суммарная награда
        episode_loss = 0  # накопленная ошибка
        steps = 0  # число шагов в эпизоде

        # Обработка событий Pygame
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                return

        while True:
            # Выбор действия
            action = agent.choose_action(state)

            # Выполнение действия: next_state — новое состояние, reward —
полученная награда, done — флаг завершения эпизода, info - счет
            next_state, reward, done, info = env.step(action)

            # Сохранение опыта
            agent.store_experience(state, action, reward, next_state, done)

            # Обучение (не на каждом шаге)
            if steps % 4 == 0:
                loss = agent.train()
                if loss:
                    episode_loss += loss

            # Обновление состояния
            state = next_state
            total_reward += reward
            steps += 1
```

Окончание Листинга Г

```
            # Визуализация
            if episode % render_every == 0:  # Рендеринг только некоторых
эпизодов:
                env.render()
                pygame.display.set_caption(
                    f"Змейка DQN - Эпизод: {episode}, "
                    f"Счет: {info['score']}, "
                    f"Epsilon: {agent.epsilon:.3f}"
                )

            if done:
                break

        # Сбор статистики
        scores.append(info["score"])  # Сохранение итогового счёта
        losses.append(episode_loss / max(steps, 1))  # Средняя ошибка за эпизод
        epsilons.append(agent.epsilon)  # Запоминаем значение ε

        # Вывод прогресса
        if (episode + 1) % 10 == 0:
            avg_score = np.mean(scores[-10:])  # Средний счёт за последние 10
эпизодов
            print(
                f"Эпизод: {episode+1:4d}/{episodes} | "
                f"Счет: {info['score']:2d} | "
                f"Средний счет (10 эп.): {avg_score:5.2f} | "
                f"Epsilon: {agent.epsilon:.3f} | "
                f"Память: {len(agent.memory)}"
            )

    return agent, scores, losses, epsilons
```