7. ЛАБОРАТОРНАЯ РАБОТА № 7 «ИССЛЕДОВАНИЕ ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ»

7.1. Цель работы

Исследование методов недетерминированного поиска решений задач, приобретение навыков программирования интеллектуальных агентов, функционирующих в недетерминированных средах, исследование способов построения агентов, обучаемых на основе алгоритмов обучения с подкреплением.

7.2. Краткие теоретические сведения

7.2.1. Недетерминированный поиск

Рассмотрим методы поиска в условиях, когда действие агента, выполненное в некотором состоянии, может приводить к нескольким новым состояним с определенной долей вероятности. Такие задачи поиска, в которых поведение агента характеризуется долей неопределенности, относятся к **недетерминированным задачам поиска**. Они могут быть решены с помощью моделей, известных **как марковские процессы принятия решений (Markov Decision Processes – MDP).**

7.2.2. Марковские процессы принятия решений

Марковский процесс принятия решений определяется следующим набором множеств и функций [3, 7, 8]:

1**. Множество состояний** $s \in S$. Состояния в MDP представляются также, как и состояния при поиске решений задач в пространстве состояний;

2. **Множество действий** $a \in A$. Действия в MDP также представляются аналогично действиям в ранее рассмотренных методах поиска решений задач;

3. **Функция перехода** $T(s,a,s')$ **(transition function),** представляется вероятностями перехода агента из состояния s в состояние s' посредством выполнения действия a, т.е. $P(s'|\, s, a)$;

4. **Функция вознаграждения** $R(s,a,s')$ **(reward function)** - определяет *награду*, получаемую агентом при переходе из состояния s в состояние s' посредством выполнения действия a;

5. **Коэффициент дисконтирования** γ **(discount factor) –** фактор определяющий степень важности текущих и будущих наград.

Формально марковский процесс принятия решений можно представить в виде множества

$$\mathbf{MDP=\{}\, S, A, T, R, \gamma \,\mathbf{\}.}$$

При определении MDP также указывают н**ачальное состояние** и, возможно, **конечное состояние**. Награда может быть положительной или отрицательной в зависимости от того, приносят ли действия пользу агенту.

Для MDP характерно выполнение **марковского свойства**: следующее состояние определяется только текущим состоянием. Это подобно поиску, в котором

функция-приемник использует только предшествующее состояние (а не историю состояний).

Решение задачи, представляемой в виде MDP, сводится к построению функции **оптимальной политики:**

$$\pi^*: S \rightarrow A.$$

Политика – это функция **π**, определяющая для каждого состояния *s* действие *a* . Оптимальная политика, обозначаемая как π*, максимизирует ожидаемое накопленное вознаграждение, если ей следовать. В заданном состоянии *s* агент, следующий политике π, выберет действие $a=\pi(s)$.

Смену состояний агента можно представить в виде последовательности:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

Цель функционирования агента – максимизация суммарного вознаграждения, которое можно представить в виде **функции полезности**

$$U([s_0, a_0, s_1, a_1, s_2, \dots]) = R(s_0, a_0, s_1) + R(s_1, a_1, s_2) + R(s_2, a_2, s_3) + \dots \quad (7.1)$$

Если число шагов конечное, то это **эпизодическая задача** и награды суммируются по эпизоду (до терминального состояния). Сокращенно это можно записать в виде

$$U([r_0, r_1, r_2, \dots]) = r_0 + r_1 + r_2 + \cdots, \quad (7.2)$$

где r_0, r_1,… – награды на каждом шаге. В моделях **конечного горизонта** эпизод прерывается после фиксированного числа шагов (аналогично ограничению глубины в деревьях поиска).

Для **продолжающихся задач**, у которых нет завершающего состояния (в отличие от эпизодических задач) вводится понятие коэффициента дисконтирования ***γ***. Определим функцию полезности **с коэффициентом дисконтирования** в следующем виде:

$$U([r_0, r_1, r_2, \dots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \cdots. \quad (7.3)$$

Коэффициент *γ* определяет относительную важность будущих и немедленных наград. Его значение лежит в диапазоне от 0 до 1; $\gamma = 0$ означает, что немедленные награды более важны, а $\gamma=1$ означает, что будущие награды важнее немедленных. Для модели бесконечного горизонта значение функции полезности определяется выражением ($\gamma<1$):

$$U([r_0, \dots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \le R_{\max}/(1-\gamma). \quad (7.4)$$

Функция ценности состояния (state value function) или просто «функция ценности» указывает, насколько хорошо для агента пребывание в конкретном состоянии *s* при следовании политике π. Эта функция равна ожидаемой кумулятивной награде при следовании политике π в состоянии *s*:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t \ge 0} \gamma^t r_t | s_0 = s, \pi\right], \quad (7.5)$$

где $\mathbb{E}$ – символ математического ожидания.

Q-функция ценности состояния-действия (*s*, *a*) равна ожидаемой кумулятивной награде при выборе действия *a* в состоянии *s* и при следовании политике π:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t\geq 0}\gamma^t r_t | s_0 = s, a_0 = a, \pi\right]. \quad (7.6)$$

Q-функция, в отличие от функции ценности состояния, скорее *определяет полезность действия в состоянии*, а не полезность самого состояния.

Мы хотим найти **оптимальную политику π***, которая максимизирует накопленную награду. *Формально оптимальная политика определяется выражением*

$$\pi^* = \arg\max_{\pi}\mathbb{E}\left[\sum_{t\geq 0}\gamma^t r_t | \pi\right]. \quad (7.7)$$

7.2.3. MDP дерево поиска и уравнение Беллмана

Марковский процесс принятия решений, также как поиск в пространстве состояний, можно представить в виде дерева поиска (рисунок 7.1). Неопределенность моделируется в этих деревьях с помощью **Q-состояний**, также известных как узлы **состояния-действия** (*s*, *a*), по существу идентичных узлам жеребьевки в Expectimax деревьях; **Q-состояние** представляется в виде действия *a* в состоянии *s* и обозначается как кортеж (*s*, *a*); Q-состояния используют вероятности для моделирования неопределенности того, что агент, выполнив действие *a*, перейдет в состояние *s*'.

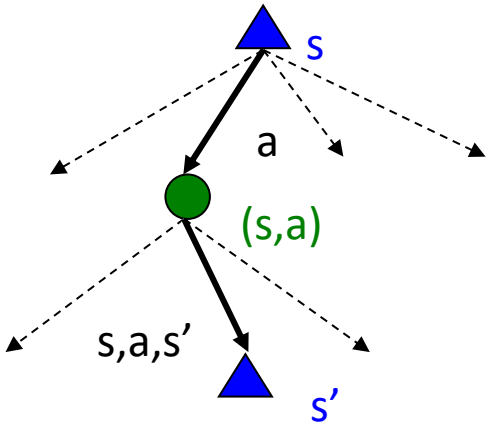

Рисунок 7.1. – MDP дерево поиска

Для решения задачи MDP используют уравнение Беллмана. Когда мы ищем решение задачи, представленной в виде MDP, интерес представляет нахождение оптимальных функций ценности и политики.

Оптимальной функцией ценности $V^*(s)$ называется функция ценности, которая обеспечивает максимальную ценность при следовании определенной политике [7]:

$$V^*(s) = \max_{\pi} V^{\pi}(s). \quad (7.8)$$

Оптимальную функцию ценности состояния вычисляют как максимум *Q*-функции по действиям *a*:

$$V^*(s) = \max_a Q^*(s, a). \quad (7.9)$$

Ценность *q*-состояния (*s*,*a*) можно определить выражением (см. рисунок 7.1):

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[\, R(s,a,s') + \gamma\, V^*(s')\,] \quad . \quad (7.10)$$

Подставив (7.10) в (7.9), получим **уравнение Беллмана**

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \quad , \quad (7.11)$$

где $T(s,a,s')$ – вероятность перехода из (s,a) в s', а $R(s,a,s')+\gamma V^*(s')$ – ценность действия a в состоянии s при переходе в s'. Уравнение Беллмана устанавливает *рекуррентную связь* между ценностью состояния s и ценностью следующего состояния s' при выполнении наилучшего действия a.

7.2.4. Итерации по значениям ценности состояний

Итерации по значениям **ценности состояний** $V(s)$ – это алгоритм динамического программирования, который обеспечивает итеративное вычисления ценности состояний $V(s)$, пока не выполнится условие сходимости:

$$\forall s, V_{k+1}(s) = V_k(s)). \quad (7.12)$$

Алгоритм предусматривает следующие шаги:

1. Положить $V_0(s) = 0$: отсутствие действий на шаге 0 означает, что ожидаемая сумма наград равна нулю;
2. Для каждого из состояний повторять вычисление (обновлять) ценности состояния в соответствии с выражением, пока значения не сойдутся

$$\forall s \in S,\ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')] \quad . \quad (7.13)$$

Временная сложность каждой итерации алгоритма соответствует $O(S^2A)$. Алгоритм обеспечивает схождение ценности каждого состояния к оптимальному значению. При этом следует отметить, что политика может сходиться задолго до того, как сойдутся ценности состояний.

7.2.5. Извлечение политики

Цель решения MDP – определение оптимальной политики. Предположим, что имеются оптимальные значения ценности состояний $V^*(s)$. Каким образом следует при этом действовать в каждом состоянии? Это можно определить, применив метод, который называется **извлечением политики (**policy extraction**)**. Если агент находится в состоянии *s*, то следует выбрать действие *a*, обеспечивающее получение максимальной ожидаемой суммарной награды:

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \quad . \qquad (7.14)$$

Не удивительно, что *a* является действием, которое приводит к *q*-состоянию с максимальным значением *q*-ценности, которое формально соответствует определению оптимальной политики:

$$\pi^*(s) = \arg\max_a Q^*(s,a) \quad . \qquad (7.15)$$

Таким образом, ***проще выбирать действия по q-ценностям, чем по ценностям состояний!***

7.2.6. Итерации по политикам

Итерации по значениям функций ценности могут быть очень медленными. Поэтому, когда требуется определить политику, можно использовать альтернативный подход, основанный на **итерациях по политикам**. В соответствии с этим подходом, предусматриваются следующие шаги поиска оптимальной политики:

1. Определить первоначальную политику π. Такая политика может быть произвольной;
2. Выполнить оценку текущей политики, используя метод **оценки политики** (policy evaluation). Для текущей политики π оценка политики означает вычисление $V^{\pi}(s)$ для всех состояний:

$$V^{\pi}(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V^{\pi}(s')] \quad . \qquad (7.16)$$

   Вычисление $V^{\pi}(s)$ можно выполнить *двумя способами*: либо решить систему линейных уравнений (7.16) относительно $V^{\pi}(s)$ (например, в Matlab); либо вычислить оценки состояний итерационно, используя пошаговые обновления:

$$V_0^{\pi}(s) = 0,$$

$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^{\pi}(s')] \quad . \qquad (7.17)$$

   Обозначить текущую политику как π_i . Временная сложность итерационной оценки политики соответствует $O(S^2)$ на 1 итерацию. Тем не менее, второй способ оценки политики значительно медленнее на практике.
3. После того как выполнена оценка текущей политики π_i, выполняется **улучшение политики** (policy improvement) на основе одношагового метода извлечения политики**:**

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^{\pi_i}(s') \right] . \quad (7.18)$$

Если $\pi_{i+1} = \pi_i$, то алгоритм останавливают и полагают $\pi_{i+1} = \pi_i = \pi^*$, иначе переходят к п.2.

Важно отметить, что политика в этом случае сходится к оптимальной политике. Кроме этого, схождение политики происходит быстрее схождения значений ценности состояний.

7.2.7. Обучение с подкреплением

Обучение с подкреплением (**RL**, reinforcement learning) — область машинного обучения, в которой обучение осуществляется посредством взаимодействия агента с окружающей средой (рисунок 7.2) [7, 8]. В среде RL вы не указываете агенту, что и как он должен делать, вместо этого **агент получает награду за каждое выполненное действие**. В итоге агент начинает выполнять действия, которые максимизируют вознаграждение.

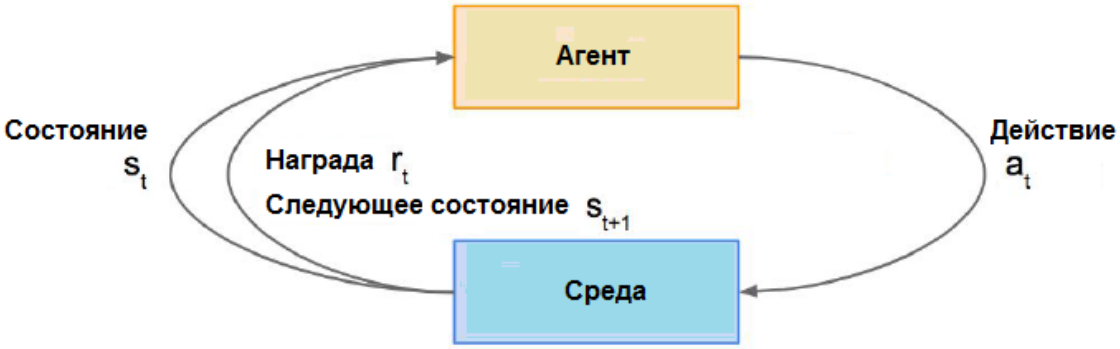

Рисунок 7.2 – Обучение с подкреплением

Предполагается что среда описывается марковским процессом принятия решений (MDP). Но если в методах итерации по значениям и итерации по политикам нам были известны функции переходов ***T*** и вознаграждений ***R,*** то при обучении с подкреплением эти функции неизвестны.

При обучении с подкреплением агент предпринимает попытки **исследования** (**exploration**) среды, совершая действия и получая обратную связь в форме новых состояний и наград. Агент использует эту информацию для определения оптимальной политики в ходе процесса, называемого обучением на основе подкрепления, прежде, чем начнет **эксплуатировать** (**expoloitation**) полученную политику.

Последовательность взаимодействия агента со средой (*s, a, s', r*) на одном шаге называют **выборкой.** Коллекция выборок, которая приводит к терминальному состоянию, называется **эпизодом.**

Агент обычно выполняет много эпизодов в ходе исследования для того, чтобы собрать достаточно данных для обучения.

Модель является представлением среды с точки зрения агента. Различают два типа обучения с подкреплением: **основанное на модели и без модели**.

7.2.8. Обучение с подкреплением на основе модели

В процессе обучения, основанного на модели, агент предпринимает попытки оценивания вероятностей переходов ***T*** и вознаграждений ***R*** по выборкам, получаемым во время исследования, перед тем как использовать эти оценки для нахождения MDP решения.

Шаг 1: **Эмпирическое обучение MDP модели:**

- Подсчёт числа исходов s' для каждой пары (s, a);
- Нормализация числа исходов для получения оценки $T(s, a, s')$;
- Оценка наград для каждого перехода (s, a, s').

Шаг 2: **Получение решения MDP:**

После схождения оценок T, обучение завершается генерацией политики $\pi(s)$ на основе алгоритмов итерации по значениям или политикам.

Обучение на основе модели интуитивно простое, но характеризуется большой пространственной сложностью (из-за необходимости хранения значений счетчиков переходов (s, a, s')).

7.2.9. Обучение с подкреплением без модели

При обучении с подкреплением без модели используют три алгоритма, которые разделяются на две группы:

1. **Алгоритмы пассивного обучения:**
   - Алгоритм прямого оценивания;
   - Обучение на основе временных различий;
2. **Алгоритмы активного обучения:**
   - Q-обучение.

В случае пассивного обучения агент следует заданной политике и обучается ценностям состояний на основе накопления выборочных значений из эпизодов, что в общем, соответствует оцениванию политики при решении задачи MDP, когда ***T*** и ***R*** известны.

В случае активного обучения агент использует обратную связь для итеративного обновления его политики, пока не построит оптимальную политику после достаточного объема исследований.

7.2.10. Алгоритм прямого оценивания (обучение без модели)

Цель: вычисление ценности каждого состояния при следовании политике π.

Идея: Усреднять наблюдаемые выборки значений ценности.

Алгоритм:

- Действовать в соответствии с политикой π:
  - каждый раз при посещении состояния, подсчитывать и аккумулировать ценности состояния и число посещений состояния;
  - найти среднюю ценность состояния.

Положительные свойства прямого оценивания: простота; не требует знаний *T, R*; вычисляет средние значения ценностей, используя просто выборки переходов.

Основным недостатком алгоритма являются значительные временные затраты.

7.2.11. Обучение на основе временных различий (TD-обучение)

Основная идея TD–обучения (TD-temporal difference) заключается в том, чтобы обучаться на основе выборок при выполнении каждого действия [7, 8]:

- обновлять $V(s)$ каждый раз, при совершении перехода (s, a, s', r);
- более вероятные переходы будут вносить вклад в обновления более часто.

В ходе TD–обучения выполняется оценка ценности состояния при фиксированной политике. При этом ценность состояний вычисляется путем аппроксимации матожидания в уравнении Беллмана *экспоненциальным скользящим средним* выборочных значений $V(s)$:

- выборочное значение $V(s)$:

$$sample = R(s,\pi(s),s') + \gamma V^{\pi}(s') \tag{7.19}$$

- скользящее среднее выборок:

$$V^{\pi}(s) \leftarrow (1 - \alpha)V^{\pi}(s) + (\alpha)\, sample \tag{7.20}$$

или

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha\,(sample - V^{\pi}(s)). \tag{7.21}$$

Правило обновления ценности состояния (7.21) называют **правилом *обновления на основе временных различий.*** Различие равно разности между выборочной наградой $(R + \gamma V(s'))$ и ожидаемой наградой $V(s)$, умноженной на скорость обучения α. Фактически эта разность есть погрешность, которую называют TD-погрешностью.

Последовательность шагов **алгоритма TD-обучения** выглядит так:

1. Инициализировать $V(s)$ нулями или произвольными значениями;
2. Запустить эпизод, для каждого шага в эпизоде выполнить действие a в состоянии s, получить награду r и перейти в следующее состояние (s');
3. Обновить ценности состояний по правилу TD-обновления (7.20);
4. Повторять шаги 2 и 3, пока не будет достигнуто схождение ценности состояний.

Алгоритм обеспечивает более быстрое схождение ценности состояний по сравнению с алгоритмом прямого оценивания.

7.2.12. Q-обучение

TD-обучение – подход к оцениванию политики без модели, моделирующий обновление Беллмана с помощью он-лайн усреднения выборок. Однако, если необ-

ходимо будет преобразовать значения $V^{\pi}(s)$ в новую политику возникнут сложности, так как для этого в соответствии с (7.15) необходимо оперировать значениями *q*-ценностей.

В *q*-обучении используется правило обновления, известное как правило итераций по *q*-ценностям [7]:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]. \quad (7.22)$$

Это правило обновления следует из (7.10) и (7.9). Алгоритм q-обучения строится по схеме, аналогичной алгоритму TD-обучения, с использованием *выборок значений Q-состояний*

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a') \quad (7.23)$$

и их скользящем усреднении

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot sample. \quad (7.24)$$

или

$$Q(s, a) \leftarrow Q(s, a) + \alpha(sample - Q(s, a)) = Q(s, a) + \alpha\, difference,$$

где разность $difference = sample - Q(s, a)$ рассматривается как отличие выборочной оценки *Q*-ценности *sample* от ожидаемой оценки $Q(s, a)$.

При достаточном объеме исследований и снижении в ходе обучения скорости обучения α оценки *Q*-ценностей, получаемые с помощью (7.24), будут сходиться к оптимальным значениям для каждого Q-состояния. В отличие от TD-обучения, которое требует применения дополнительных технологий извлечения политики, *q*-обучение формирует оптимальную политику непосредственно при выборе субоптимальных или случайных действий.

7.12.13. Алгоритм двойного Q-обучения

Q-обучение может завышать оценки ценности действий, главным образом из-за компоненты $\max_a Q(s', a')$. Алгоритм двойного *Q*-обучения призван преодолеть этот недостаток посредством использования двух *Q*-функций, которые обозначим *Q1* и *Q2*. На каждом шаге обновляется одна функция. Если выбрана *Q1*, то обновление производится по формуле:

$$a^* = \operatorname{argmax}_a Q1(s', a'); \quad (7.25)$$

$$Q1(s,a) \leftarrow Q1(s,a) + \alpha[R(s,a,s') + \gamma Q2(s', a^*) - Q1(s,a)], \quad (7.26)$$

а если *Q2*, то по формуле

$$a^* = \operatorname{argmax}_a Q2(s', a'); \quad (7.27)$$

$$Q2(s,a) \leftarrow Q2(s,a) + \alpha[R(s,a,s') + \gamma Q1(s', a^*) - Q2(s,a)] \quad (7.28)$$

Это значит, что в обновлении каждой Q-функции участвует другая функция. При этом применяется жадный поиск, что уменьшает степень завышения оценки ценности действий по сравнению с одной Q-функцией.

7.2.14. ε-жадная стратегия

Жадная стратегия выбирает оптимальное действие **среди уже исследованных**. Что лучше искать: новое лучшее действие или действие, лучшее из всех исследованных действий? Это называется *дилеммой между исследованием и эксплуатацией*.

Чтобы разрешить дилемму, вводится ***ε-жадная стратегия:*** действие выбирается среди уже исследованных на основе текущей политики с вероятностью 1 – ε и с вероятностью ε будет выполняться опробывание новых случайных действий (**исследование**). Значение ε должно уменьшаться со временем, поскольку заниматься исследованиями до бесконечности незачем. Таким образом, со временем политика переходит **на эксплуатацию** «хороших» действий.

7.2.15. Q-обучение с линейной аппроксимацией

Обычное Q-обучение требует вычисления значений *Q(s, a)* по всем возможным парам состояние-действие. Но представьте среду, в которой число состояний очень велико, а в каждом состоянии доступно множество действий. Перебор всех действий в каждом состоянии занял бы слишком много времени.

Возможное решение – аппроксимировать функцию *Q*(*s*, *a*). Для этого представляют ценность состояний с помощью *вектора признаков.* **Признаки** - это функции, которые фиксируют важные свойства состояний. Примеры признаков состояний для игры Пакман:

- расстояние до ближайшего призрака;
- расстояние до ближайшей гранулы;
- число призраков;
- 1/ (расстояние до призрака)^2;
- находится ли Пакман в ловушке? (0/1);
- и др..

Используя вектор признаков, мы можем аппроксимировать Q-функцию ценности состояния, например, в виде линейной комбинации признаков с весами:

$$Q(s,a) \quad = \quad w_1 \cdot f_1(s,a) + w_2 \cdot f_2(s,a) + \ldots + w_n \cdot f_n(s,a) = \vec{w} \cdot \vec{f}(s,a) \quad (7.29)$$

где $\vec{f}(s,a) = \begin{bmatrix} f_1(s,a) & f_2(s,a) & \ldots & f_n(s,a) \end{bmatrix}^T$ – вектор признаков состояния (*s*, *a*), а $\vec{w} = \begin{bmatrix} w_1 & w_2 & \ldots & w_n \end{bmatrix}$ – вектор весов.

Основные шаги **алгоритма Q-обучения с аппроксимацией**:

1) выполнить действие в состоянии *s* в соответствии с ε-жадной стратегией и получить выборку (*s*, *a*, *s'*, *r*);

2) вычислить разность (*difference*) между выборочным значением *Q*-ценности $[R(s,a,s') + \gamma \max_{a'} Q(s',a')]$ и текущим значением $Q(s, a)$

$$difference = [R(s,a,s') + \gamma \max_{a'} Q(s',a')] - Q(s,a) :$$

3) обновить веса признаков состояний в соответствии с простым *дельта правилом*

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a). \quad (7.30)$$

и вычислить новые значения ценности $Q(s, a)$ в соответствии с (7.29);

4) повторять шаги 1)-3) до схождения Q-ценностей.

Вместо того, чтобы хранить Q-ценности для каждого состояния **Q-обучение с аппроксимацией** позволяет хранить только один весовой вектор. В результате это даёт более эффективную версию алгоритма с точки зрения использования памяти.

7.2.16. Функция разведки (исследования)

При использовании рассмотренного алгоритма *q*-обучения с аппроксимацией требуется вручную управлять коэффициентом ε. Этого можно избежать, если применить **функцию разведки (исследования)**, которая использует модифицированное правило обновления ценности *q*-состояний, предоставляющее предпочтения тем состояниям, которые посещаются реже:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} f(s',a')], \quad (7.31)$$

где f – функция разведки. Обычно полагают, что

$$f(s,a) = Q(s,a) + \frac{k}{N(s,a)}, \quad (7.32)$$

где k – некоторая предопределенная константа, $N(s,a)$ – число посещений *q*-состояния (s, a). Агент в состоянии s всегда выбирает действие с максимальным значением $f(s,a)$ и, соответственно, не принимает вероятностных решений относительно исследований и эксплуатации, так как решение в пользу исследования новых выборок автоматически регулируется в (7.32) членом $k/N(s, a)$, который будет иметь большие значения для редко выполняемых действий. По мере увеличения числа итераций, счетчики числа посещений состояний растут и $k/N(s, a)$ стремится к нулю, а $f(s,a)$ стремится к $Q(s,a)$. Таким образом, исследования выборок становятся все более и более редкими.

7.2.17. Алгоритм двойной глубокой *Q*-сети (DDQN)

Если в качестве аппроксиматора $Q(s, a)$ используется глубокая нейронная сеть (DQN – deep *Q*-network), то говорят **о глубоком *Q*-обучении**[8]. В **алгоритме двойной DQN** (Double DQN) для оценивания целевых *Q*-значений используется

отдельная сеть, которая имеет такую же структуру, как сеть, выполняющая аппроксимацию $Q(s, a)$. Но ее веса изменяются только после каждых T эпизодов (T – настраиваемый гиперпараметр). Обновление сводится просто к копированию весов аппроксимирующей (предсказательной) сети. Таким образом, целевая Q-функция в течение некоторого времени остается неизменной, что повышает устойчивость процесса обучения.

Математически обучение двойной DQN означает минимизацию в ходе обучения функции потерь в виде среднего квадрата следующей ошибки:

$$difference=[\ R(s,a,s')+\gamma\ \max_a Q_T(s',a')\ ] - Q(s,a), \qquad (7.33)$$

где $Q_T(s',a')$ – функция ценности целевой сети, а $Q(s,a)$ – функция ценности предсказательной сети.

Соответственно, правило обновления весов нейросети в этом случае будет отличаться от (7.30) и базироваться на алгоритме обратного распространения ошибки, используемого при обучении нейронных сетей.

7.2.18. DQN с буфером воспроизведения опыта

Аппроксимация значений Q-функций нейронной сетью, обучаемой на одном примере за раз, ведет себя не устойчиво. Для повышения устойчивости используют **буфер воспроизведения опыта**.

Идея заключается в том, чтобы сохранять в памяти опыт агента в виде выборок (s, a, s', r), полученных на протяжении эпизодов в сеансе обучения. Обучение с буфером воспроизведения опыта состоит из 2-х этапов: накопление опыта и обновление модели (ей) на миниблоке из случайных выборок прошлого опыта.

Воспроизведение опыта может стабилизировать процесс обучения, обеспечив набор слабо коррелированных примеров.

7.3. Задания для выполнения

Задание 1. Итерации по значениям

Реализуйте агента, осуществляющего итерации по значениям в соответствии с выражением (7.13), в классе **ValueIterationAgent** (класс частично определен в файле **valueIterationAgents.py**). Агент **ValueIterationAgent** получает на вход MDP при вызове конструктора класса и выполняет итерации по значениям для заданного количества итераций (опция **-i**) до выхода из конструктора.

При итерации по значениям вычисляются k-шаговые оценки оптимальных значений V_k. Дополнительно к **runValueIteration**, реализуйте следующие методы для класса **ValueIterationAgent**, используя значения V_k.:

computeActionFromValues(state) – определяет лучшее действие в состоянии (политику) с учетом значений ценности состояний, хранящихся в словаре **self.values**;

computeQValueFromValues(state, action) возвращает q-ценность пары **(state, action**) с учетом значений ценности состояний, хранящихся в словаре **self.values**

(данный метод реализует вычисления с учетом уравнения Беллмана для *q*-ценностей);

Вычисленные значения отображаются в графическом интерфейсе пользователя (см. рисунки ниже): ценности состояний представляются числами в квадратах, значения *q*-ценностей отображаются числами в четвертях квадратов, а политики - это стрелки, исходящие из каждого квадрата.

Важно: используйте «пакетную» версию итерации по значениям, где каждый вектор ценности состояний V_k вычисляется на основе предыдущих значений вектора V_{k-1}, а не на основе «онлайн» версии, где один и тот же вектор обновляется по месту расположения. Это означает, что при обновлении значения состояния на итерации *k* на основе значений его состояний-преемников, значения состояния-преемника, используемые в вычислении обновления, должны быть значениями из итерации *k*-1 (даже если некоторые из состояний-преемников уже были обновлены на итерации *k*).

Примечание: политика, сформированная по значениям глубины *k*, фактически будет соответствовать следующему значению накопленной награды (т.е. вы вернете π_{k+1}). Точно так же *q*-ценности дают следующее значение награды (т.е. вы вернете Q_{k+1}). Вы должны вернуть политику π_{k+1} .

Подсказка: при желании вы можете использовать класс **util.Counter** в **util.py**, который представляет собой словарь ценностей состояний со значением по умолчанию, равным нулю. Однако будьте осторожны с **argMax**: фактический argmax, который вам необходим, может не быть ключом!

Примечание. Обязательно обработайте случай, когда состояние не имеет доступных действий в MDP (подумайте, что это означает для будущих вознаграждений).

Подсказка: в среде BookGrid, используемой по умолчанию, при выполнении 5 итераций по значениям должен получиться результат, изображенный на рисунке 7.3:

```
python gridworld.py -a value -i 5
```

Оценивание: ваш агент, использующий итерации по значениям, будет оцениваться с использованием иной схемы клеточного мира. Будут проверяться ценности состояний, *q*-ценности и политики после заданного количества итераций, а также с учетом выполнения условия сходимости (например, после 100 итераций).

Задание 2. Реализация политик

Рассмотрим схему среды **DiscountGrid**, изображенную на рисунке 7.4. Эта среда имеет два терминальных состояния с положительной наградой (в средней строке): закрытый выход с наградой +1 и дальний выход с наградой +10. Нижняя строка схемы состоит из конечных состояний с отрицательными наградами -10 (показаны красным). Начальное состояние – желтый квадрат. Различают два типа путей: (1) пути, которые проходят по границе «обрыва» вдоль нижней строки схемы, эти пути короче, но характеризуются большими отрицательными наградами (они

обозначены красной стрелкой на рисунке 7.4); (2) пути, которые «избегают обрыва» и проходят по верхней строке схемы. Эти пути длиннее, но они с меньшей вероятностью принесут отрицательные результаты. Эти пути обозначены зеленой стрелкой на рисунке 7.4.

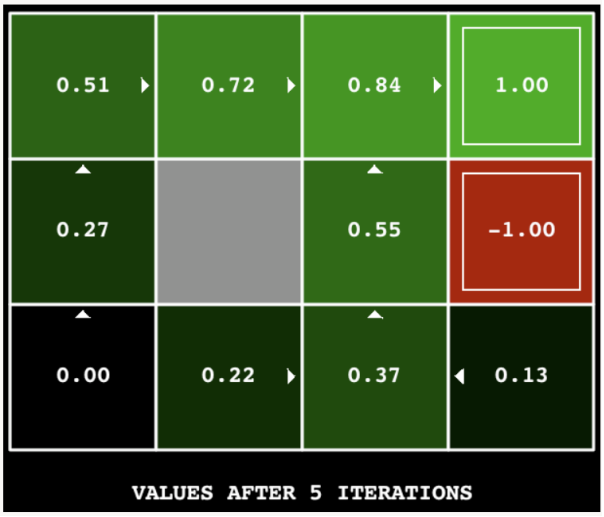

Рисунок 7.3 – Состояние среды BookGrid после 5 итераций

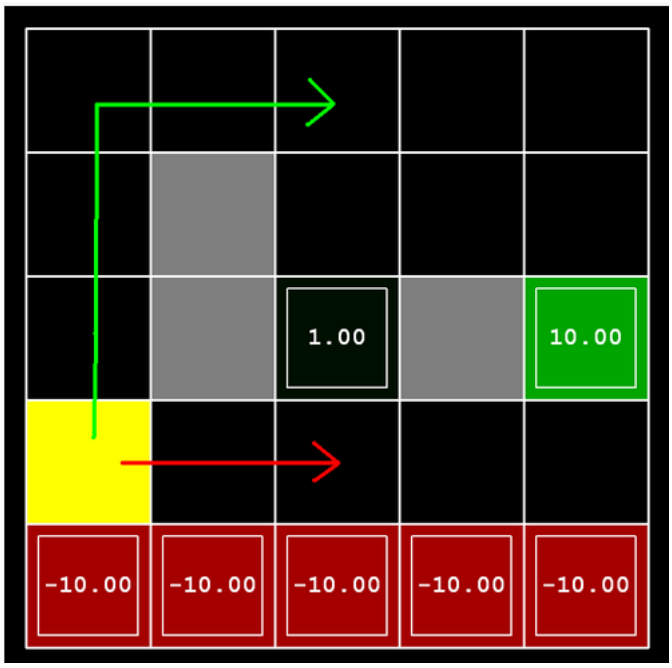

Рисунок 7.4 – Состояния среды DiscountGrid

В этом задании необходимо подобрать значения коэффициента дисконтирования, уровня шума и текущей награды для схемы **DiscountGrid**, чтобы сформировать оптимальные политики нескольких различных типов. Ваш выбор значений для указанных параметров должен обладать свойством, которое заключается в том, что если бы ваш агент следовал своей оптимальной политике, не подвергаясь никакому шуму, то он демонстрировал бы требуемое поведение. Если определенное поведение не достигается ни для одной настройки параметров, необходимо сообщить, что политика невозможна, вернув строку «НЕВОЗМОЖНО».

Ниже указаны оптимальные типы политик, которые вы должны попытаться реализовать:
1) агент предпочитает близкий выход (+1), перемещаясь вдоль обрыва (-10);
2) агент предпочитает близкий выход (+1), но избегает обрыва (-10);
3) агент предпочитает дальний выход (+10), перемещаясь вдоль обрыва (-10);
4) агент предпочитает дальний выход (+10), избегает обрыва (-10);
5) агент предпочитает избегать оба выхода и обрыва (так что эпизод никогда не должен заканчиваться).

Внесите необходимые значения параметров в функции от **question2a()** до **question2e()** в файле **analysis.py**. Эти функции возвращают кортеж из трех элементов (дисконт, шум, награда).

Задание 3. Q-обучение

Обратите внимание, что ваш агент итераций по значениям фактически не обучается на собственном опыте. Он скорее, изучает модель MDP, чтобы сформировать полную политику, прежде чем начнет взаимодействовать с реальной средой. Когда он действительно взаимодействует со средой, он просто следует предварительно вычисленной политике. Это различие может быть незаметным в моделируемой среде, такой как Gridword, но очень важно в реальном мире, когда полное описание MDP отсутствует.

В этом задании необходимо реализовать агента с *q*-обучением, который мало занимается конструированием планов, но вместо этого учится методом проб и ошибок, взаимодействуя со средой с помощью метода **update(state, action, nextState, reward)**. Шаблон *q*-обучения приведен в классе **QLearningAgent** в файле **qlearningAgents.py,** обращаться к нему можно из командной строки с параметрами **'-a q'**. Для этого задания необходимо реализовать методы **update, computeValueFromQValues, getQValue и computeActionFromQValues.**

Примечание. Для метода **computeActionFromQValues**, который возвращает лучшее действие в состоянии, при наличии нескольких действий с одинаковой *q*-ценностью, необходимо выполнить случайный выбор с помощью функции **random.choice()**. В некоторых состояниях действия, которые агент ранее не встречал, могут иметь значение *q*-ценности, равное нулю, и если все действия, которые ваш агент встречал раньше, имеют отрицательное значение ***q***-ценности, то действие, которое не встречалось может быть оптимальным.

Важно: убедитесь, что в функциях **computeValueFromQValues** и **computeActionFromQValues** вы получаете доступ к значениям *q*-ценности, вызывая **getQValue**. Эта абстракция будет полезна для задания 6, когда вы переопределите **getQValue** для работы с признаками пар «состояние-действие».

Задание 4. Эпсилон-жадная стратегия

Завершите реализацию агента с *q*-обучением, дописав эпсилон-жадную стратегию выбора действий в методе **getAction** класса **QLearningAgent**.Метод должен обеспечивать выбор случайного действия с вероятностью **epsilon** и с вероятностью

1- epsilon выбирать действие с лучшим значением *q*-ценности. Обратите внимание, что выбор случайного действия может привести к выбору наилучшего действия, то есть вам следует выбирать не случайное субоптимальное действие, а любое случайное допустимое действие.

Вы можете выбрать действие из списка случайным образом, вызвав функцию **random.choice**. Для моделирования двоичной случайной переменной используйте вызов **util.flipCoin(epsilon)**, который возвращает **True** с вероятностью **epsilon** и **False** с вероятностью **1- epsilon**.

Задание 5. Q-обучение и Пакман

Пора поиграть в Пакман! Игра будет проводиться в два этапа. На первом этапе обучения Пакман будет обучаться значениям ценности позиций и действиям. Поскольку получение точных значений *q*-ценностей даже для крошечных полей игры занимает очень много времени, обучение Пакмана по умолчанию выполняется без отображения графического интерфейса (или консоли). После завершения обучения Pacman перейдет в режим тестирования. При тестировании для параметров **self.epsilon** и **self.alpha** будут установлены нулевые значения, что фактически остановит *q*-обучение и отключит режим исследования, чтобы позволить Пакману использовать сформированную входе обучения политику. Тестовая игра отображается в графическом интерфейсе. Запустите *q*-обучение Пакмана для маленького поля игры **smallGrid**:

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

Обратите внимание, что **PacmanQAgent** уже определен для вас в терминах написанного агента **QLearningAgent**. **PacmanQAgent** отличается только тем, что он имеет параметры обучения по умолчанию, которые более эффективны для игры Пакман (**epsilon=0.05, alpha=0.2, gamma=0.8**). Вы получите максимальную оценку за выполнение этой части задания, если агент будет выигрывать не менее, чем в 80% случаев. Автооцениватель проведет 10 тестовых игр после 2000 тренировочных игр.

Подсказка: если агент **QLearningAgent** успешно работает с **gridworld.py** и **crawler.py**, но при этом не обучается хорошей политике для игры Пакман на поле **smallGrid,** то это может быть связано с тем, что методы **getAction** и/или **computeActionFromQValues** в некоторых случаях не учитывают должным образом ненаблюдаемые действия. В частности, поскольку ненаблюдаемые действия по определению имеют нулевое значение *q*-ценности, а все действия, которые были исследованы, могли иметь отрицательные значения *q*-ценности, то ненаблюдаемое действие может стать оптимальным. Остерегайтесь применения функции argmax класса **util.Counter.**

Чтобы оценить задание 5, выполните:

```
python autograder.py -q q5
```

Примечание. Если вы хотите поэкспериментировать с параметрами обучения, вы можете использовать параметр **-a,** например **-a epsilon=0.1, alpha=0.3,**

gamma=0.7 Затем эти значения будут доступны агенту как **self.epsilon**, **self.gamma** и **self.alpha**.

Примечание. Хотя всего будет сыграно 2010 игр, первые 2000 игр не будут отображаться из-за опции **-x 2000**, которая обозначает, что первые 2000 обучающих игр не отображаются. Таким образом, вы увидите, как Pacman играет только в последние 10 игр. Количество обучающих игр может передаваться вашему агенту также в виде опции **numTraining**.

Статистика обучения будет отображаться после каждых 100 игр. Так как параметр **epsilon** имеет положительные значения во время обучения, то Пакман играет плохо даже после того, как усвоит хорошую политику. Это происходит потому, что он иногда совершает случайный исследовательский ход в сторону призрака. Для сравнения: должно пройти от 1000 до 1400 игр, прежде чем Пакман получит положительное вознаграждение за сегмент из 100 эпизодов, что свидетельствует о том, что он начал больше выигрывать, чем проигрывать. К концу обучения вознаграждение должно оставаться положительным и быть достаточно высоким (от 100 до 350).

После того, как Пакман закончит обучение, он должен надежно выигрывать в тестовых играх (по крайней мере, в 90% случаев), поскольку теперь он использует обученную политику.

Тем не менее вы обнаружите, что обучение того же агента, казалось бы, на простой среде **mediumGrid** не работает. При этом среднее вознаграждение Пакмана остается отрицательным на протяжении всего обучения. Во время теста он играет плохо, вероятно, проигрывая все свои тестовые партии. Тренировки также займут много времени.

Пакман не может победить на больших игровых полях, потому что каждая конфигурация игры имеет специфические состояния с уникальными значениями *q*-ценности. У Пакмана нет возможности обобщить случаи столкновения с призраком и понять, что это плохо для всех позиций. Очевидно, такой вариант *q*-обучения не масштабирует большое число состояний.

Задание 6. Q-обучение с аппроксимацией

Реализуйте *q*-обучение с аппроксимацией, которое обеспечивает обучение весов признаков состояний. Дополните описания класса **ApproximateQAgent** в **qlearningAgents.py**, который является подклассом **PacmanQAgent.**

Q-обучение с аппроксимацией предполагает существование признаковой функции $f(s, a)$ от пары состояние-действие, которая возвращает вектор $[f1(s, a), \ldots fi(s, a), \ldots, fn(s, a)]$ из значений признаков. Вам предоставляются для этого возможности модуля **featureExtractors.py.** Вектор признаков – это объект **util.Counter** (подобен словарю), содержащий пары признаков и значений; все пропущенные признаки будут иметь нулевые значения.

Задание 7. Глубокое Q-обучение

В этом итоговом задании объединяются идеи Q-обучения и машинного обучения из предыдущей лабораторной работы. Вам необходимо в файле **model.py** реализовать класс **DeepQNetwork**, который представляет собой глубокую нейронную сеть, предсказывающую Q-ценности для всех возможных пар (s, a). В ходе выполнения задания вам необходимо определить конструктор нейросети **__init__** , реализовать её методы прямого распространения **forward,** вычисления потерь **get_loss,** одношагового обучения **gradient_update**.

7.4. Порядок выполнения лабораторной работы

7.4.1. Изучить по лекционному материалу и учебным пособиям [1-3, 7] методы недетерминированного поиска, основанные на использовании модели марковского процесса приятия решений, включая алгоритмы итерации по значениям и политикам, а также алгоритмы обучения подкреплением с моделями и без моделей: алгоритм прямого обучения, алгоритм TD-обучения, алгоритм Q-обучения.

7.4.2. Использовать для выполнения лабораторной работы файлы из архива **МиСИИ_лаб7_2024.zip.** Разверните программный код в новой папке и не смешивайте с файлами предыдущих лабораторных работ.

7.4.3. В этой лабораторной работе необходимо реализовать алгоритмы итераций по значениям и Q-обучение, включая глубокое Q-обучение.

Для автооценивания всех заданий лабораторной работы следует выполнить команду:

```
python autograder.py
```

Для оценки конкретного задания, например, задания 2, автооцениватель вызывается с параметром **q2,** где 2 – номер задания:

```
python autograder.py -q q2
```

Для проверки отдельного теста в пределах задания используйте команды вида:

```
python autograder.py -t test_cases/q2/1-bridge-grid
```

Код лабораторной работы содержит файлы, указанные в таблице ниже.

Файлы для редактирования:	
valueIterationAgents.py	Агенты, выполняющие итерацию по значениям для решения известного MDP.
qlearningAgents.py	Агенты, использующие Q-обучения, для классов Gridworld, Crawler и Pacman.
analysis.py	Файл для размещения Ваших ответов на вопросы заданий лабораторной работы.
model.py	Файл, содержащий класс DeepQNetwork
Файлы, которые необходимо просмотреть:	

mdp.py	Определяет методы для общих MDP.
learningAgents.py	Определяет базовые классы ValueEstimationAgent и QLearningAgent, которые Ваши агенты должны расширить.
util.py	Утилиты, включающие util.Counter, которые полезны для Q-обучения.
gridworld.py	Реализация класса клеточного мира - Gridworld.
featureExtractors.py	Классы для извлечения признаков пар (состояние, действие). Используются в Q-обучении с аппроксимацией (в qlearningAgents.py).
Поддерживающие файлы, которые можно игнорировать:	
environment.py	Абстрактный класс для общей среды обучения с подкреплением. Используется в gridworld.py.
graphicsGridworldDisplay.py	Gridworld графика.
graphicsUtils.py	Графические утилиты.
textGridworldDisplay.py	Плагин для текстового интерфейса Gridworld.
crawler.py	Код робота Crawler и тест-комплект.
graphicsCrawlerDisplay.py	GUI для краулера.
autograder.py	Автоооцениватель для лабораторной работы
testParser.py	Парсер тестов автооценивателя и файлы решений
testClasses.py	Общие классы автооценивателя
test_cases/	Папка, содержащая тесты для каждого из заданий (вопросов)
reinforcementTestClasses.py	Особые тестовые классы авооценивателя

7.4.4. Для начала запустите среду **Gridworld** в ручном режиме, в котором для управления используются клавиши со стрелками:

python gridworld.py -m

Вы увидите среду **Gridworld** в виде клеточного мира размером 4x3 с двумя терминальными состояниями. Синяя точка – это агент. Обратите внимание, когда вы нажимаете клавишу «вверх», агент фактически перемещается на Север только в 80% случаев. Это свойство агента в среде **Gridworld.** Вы можете контролировать многие опции среды **Gridworld**. Полный список опций можно получить по команде:

python gridworld.py -h

Агент по умолчанию перемещается по клеткам случайным образом. При выполнении команды

python gridworld.py -g MazeGrid

Вы увидите, как агент случайно перемещается по клеткам, пока не попадет в клетку с терминальным состоянием.

Примечание: среда **Gridworld** такова, что агент сначала должен войти в предтерминальное состояние (поля с двойными линиями, показанные в графическом интерфейсе), а затем выполнить специальное действие «выход» до фактического завершения эпизода (в истинном терминальном состоянии, называемом TERMINAL_STATE, которое не отображается в графическом интерфейсе). Если вы выполните выход вручную, накопленное вознаграждение может быть меньше, чем вы ожидаете, из-за дисконтирования (опция **-d;** по умолчанию коэффициент дисконтирования равен 0,9). Просмотрите результаты, которые выводятся в консоли. Вы увидите сведения о каждом переходе, выполняемом агентом.

Как и в Pacman, позиции представлены декартовыми координатами **(x, y),** а любые массивы индексируются **[x] [y]**, где «север» является направлением увеличения **y** . По умолчанию большинство переходов получают нулевую награду. Это можно изменить с помощью опции **–r**, которая управляет текущей наградой.

7.4.5. В задании 1 требуется реализовать следующие методы класса **ValueIterationAgent(ValueEstimationAgent):**

- **runValueIteration(self);**
- **computeQValueFromValues(self, state, action);**
- **computeActionFromValues(self, state).**

Метод **runValueIteration(self)** в циклах для каждой итерации и для каждого состояния **state** осуществляет выбор действий в состоянии (с помощью **action in self.mdp.getPossibleActions(state)**) и для каждой пары (**state, action**) вычисляет значения *q*-ценностей и складирует их в некотором списке **q_state** с помощью вызова **q_state.append(self.getQValue(state, action))**. Вычисление ценности каждого состояния реализуется на основе (7.9) путем вызова **updatedValues[state] = max(q_state)**, где **updatedValues** – временный словарь, содержащий обновляемые ценности состояний **state** на каждой итерации. После вычисления ценности всех состояний для заданной итерации, они сохраняются в словаре значений ценности состояний путем копирования **self.values = updatedValues**.

Метод **computeQValueFromValues(self, state, action)** вычисляет ценности *q*-состояний в соответствии с (7.10). Для каждого следующего состояния **nextstate** после **state** вычисления реализуются с помощью вызова:

```
Qvalue+=prob*(self.mdp.getReward(state, action, nextstate) + self.dis-
count*self.getValue(nextstate)),
```

где **nextstate** и **prob** извлекаются с помощью **in** из множества **self.mdp.getTransitionStatesAndProbs(state, action)**.

Метод **computeActionFromValues(self, state)** определяет лучшее действие в состоянии **state.** Для этого он перебирает все доступные действия **action** в состоянии **state**, получаемые с помощью вызова **self.mdp.getPossibleActions(state)** и для каждого действия **action** вычисляет и запоминает *q*-ценности в словаре **policy** путем вызова **policy[action] = self.getQValue(state, action)**. В заключение метод возвращает

лучшее действие путем вызова **policy.argMax()**, что соответствует извлечению политики согласно (7.15).

Протестируйте вашу реализацию и внесите результаты тестирования в отчет:

python autograder.py -q q1

7.4.6. В задании 2 необходимо подобрать значения коэффициента дисконтирования (**answerDiscount**), уровня шума (**answerNoise**) и текущей награды (**answerLivingReward**) для среды **DiscountGrid** (рисунок 7.4). Внесите вместо **None** необходимые значения параметров в функции от **question2a()** до **question2e()** в файле **analysis.py**.

Помните, что с помощью значений текущей награды можно управлять степенью риска: большие положительные награды заставляют агента избегать выхода, умеренные положительные награды исключают риск, умеренные отрицательные допускают риск. Снижение уровня шума понижает вероятность отклонения от выбранного направления движения. Коэффициент дисконтирования определяет важность ближайших наград по отношению к будущим наградам.

Протестируйте ваши ответы и внесите результаты тестирования в отчет:

python autograder.py -q q2

Примечание. Вы можете проверить свои политики в графическом интерфейсе: **python gridworld.py -a value -i 100 -g DiscountGrid.**

На некоторых машинах стрелка может не отображаться. В этом случае нажмите кнопку на клавиатуре, чтобы переключиться на дисплей со значениями **qValue,** и мысленно вычислите политику, взяв **argmax** от **qValue** для каждого состояния.

7.4.7. В задании 3 необходимо реализовать для класса **QLearningAgent** методы **update, computeValueFromQValues, getQValue и computeActionFromQValues.**

Реализация метода **getQValue(self, state, action)** тривиальна. Метод просто обращается к свойству класса **self.values[(state, action)]** и получает значения $Q(s,a)$.

При реализации метода **computeValueFromQValues(self, state)** необходимо для всех легальных действий **action** в состоянии **state** вычислить ценности q-состояний с помощью **getQValue(state, action)** и вернуть максимальную ценность.

Реализация метода **computeActionFromQValue(self, state)** использует вызов **bestQ=computeValueFromQValues(self, state)** для вычисления лучшей ценности **bestQ** состояния **state**. Затем в цикле перебираются все допустимые действия для **state**, вычисляются с помощью **getQValue(state, action)** ценности q-состояний и находятся действия **action** с ценностью, равной **bestQ.** Найденные лучшие действия накапливаются в некотором списке. Метод осуществляет случайный выбор лучшего действия среди найденных лучщих действий.

В методе **update(self, state, action, nextState, reward)** необходимо реализовать q-обучение в соответствии с (7.23) и (7.24). Для вычисления ценности состояния s' – **nextState** используйте вызов **getValue(nextState)**, который реализует (7.9).

Можно наблюдать за тем, как агент обучается, используя клавиатуру:

python gridworld.py -a q -k 5 -m

Напомним, что параметр **-k** контролирует количество эпизодов, которые агент использует для обучения.

Подсказка: чтобы упростить отладку, вы можете отключить шум с помощью параметра **--noise 0.0** (хотя это делает *q*-обучение менее интересным).

Если вы вручную направите Pacman на север, а затем на восток по оптимальному пути для четырех эпизодов, вы должны увидеть значения *q*-ценностей, указанные на рисунке 7.5.

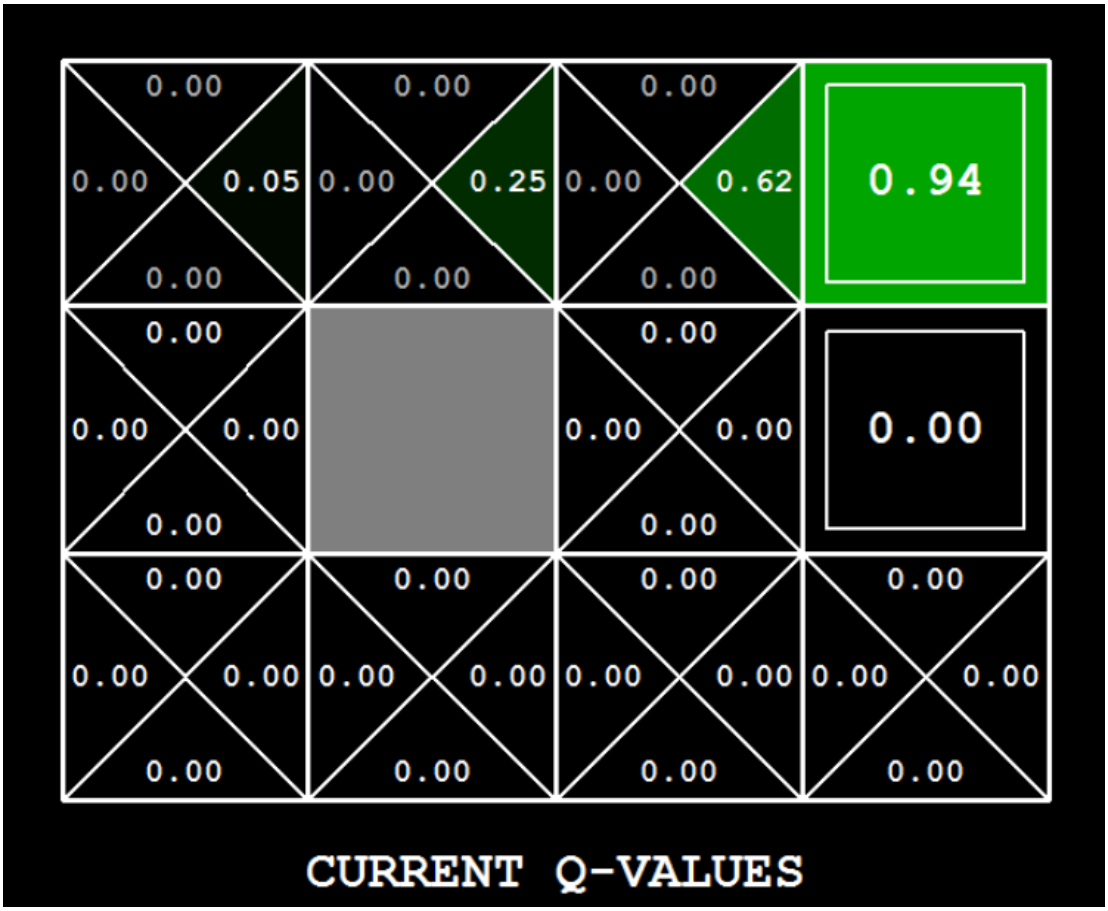

Рисунок 7.5. – Результаты *q*-обучения

В ходе автоценивания будет проверяться, обучается ли агент тем же значениям *q*-ценностей и политике, что и эталонная реализация на одном и тот же наборе примеров. Чтобы оценить вашу реализацию, запустите автооцениватель и внесите результаты в отчет:

python autograder.py -q q3

7.4.8. В задании 4 необходимо реализовать эпсилон-жадную стратегию в методе **getAction(self, state).** Реализация метода предполагает, что подбрасывается монетка с помощью вызова метода **util.flipCoin(self.epsilon)** и с вероятностью **epsilon**

возвращается случайное из допустимых действий в состоянии **state,** иначе возвращается лучшее действие, определяемое с помощью **computeActionFromQValue self,state).**

После реализации метода **getAction** проанализируйте поведение агента в **gridworld** (с **epsilon = 0.3**).

python gridworld.py -a q -k 100

Ваши окончательные значения *q*-ценностей должны будут соответствовать значениям, получаемым при итерации по значениям, особенно на проторенных путях. Однако среднее вознаграждение будет ниже, чем предсказывают *q*-ценности из-за случайных действий и начальной фазы обучения.

Проанализируйте поведение агента при разных значениях эпсилон (параметр **–e)**. Соответствует ли такое поведение агента вашим ожиданиям?

python gridworld.py -a q -k 100 --noise 0.0 -e 0.1

python gridworld.py -a q -k 100 --noise 0.0 -e 0.9

Чтобы протестировать вашу реализацию, запустите автооцениватель внесите результаты в отчет:

python autograder.py -q q4

Теперь без дополнительного кодирования вы сможете запустить *q*-обучение для робота **crawler** (рисунок 7.6**)**:

python crawler.py

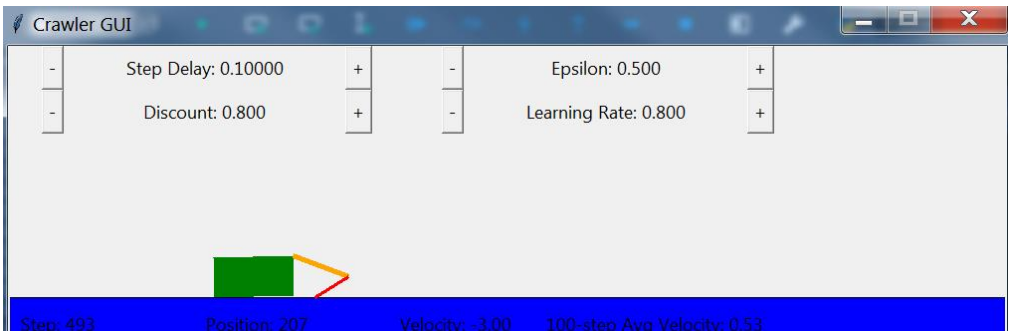

Рисунок 7.6. – Робот Crawler

Если вызов не сработает, то вероятно, вы написали слишком специфичный код для задачи GridWorld, и вам следует сделать его более общим для всех MDP.

При корректной работе на экране появится ползущий робот, управляемый из класса **QLearningAgent**. Испытайте различные параметры обучения, чтобы увидеть, как они влияют на действия агента. Обратите внимание, что параметр **step delay** (задержка шага) является параметром моделирования, тогда как скорость обучения и эпсилон являются параметрами алгоритма обучения, а коэффициент дисконтирования является свойством среды.

7.4.9. При выполнении задания 5 следуйте указаниям, приведенным в самом задании. Проведите все необходимые эксперименты, указанные в задании. Результаты внесите в отчет.

7.4.10. В задании 6 необходимо реализовать *q*-обучение с аппроксимацией, дописав методы класса **ApproximateQAgent** .

При написании кода метода **getQValue(self, state, action)**, возращающего аппроксимированное значение $Q(s,a)$, необходимо получить вектор признаков *q*-состояний с помощью вызова **self.features = self.featExtractor.getFeatures(state, action)**. А затем для всех признаков **i** из **self.features[i]** найти взвешенную сумму признаков в соответствии с (7.25).

Реализация метода **update(self, state, action, nextState, reward)** должна обеспечивать вычисление обновления весов признаков. Для этого вычисляется значение *difference* – разности выборочной и ожидаемой оценок $Q(s, a)$, затем для каждого признака **self.features[i]** вычисляются обновления весов **self.weights[i]** в соответствии с (7.26).

По умолчанию **ApproximateQAgent** использует функцию **IdentityExtractor**, которая для каждой пары **(state, action)** – состояние-действие создает отдельный признак. С такой функцией извлечения признаков агент, осуществляющий *q*-обучение с аппроксимацией, будет работать аналогично **PacmanQAgent**. Проверьте это с помощью следующей команды:

python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid

Важно: **ApproximateQAgent** является подклассом **QLearningAgent** и поэтому имеет с ним несколько общих методов, например, **getAction**. Убедитесь, что методы в **QLearningAgent** вызывают **getQValue** вместо прямого доступа к *q*-ценностям, чтобы при переопределении **getQValue** в вашем агенте использовались новые аппроксимированные *q*-ценности для определения действий.

Убедившись, что агент, основанный на аппроксимации состояний, правильно работает, запустите его с использованием **SimpleExtractor** для извлечения пользовательских признаков, который с легкостью научится побеждать:

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid

SimpleExtractor возвращает простые признаки:

- есть ли возможность съесть еду в следующей позиции;
- как далеко находится следующая еда;
- неизбежно ли столкновение с призраком;
- находится ли призрак в шаге от агента.

Даже более сложные игровые поля не должны стать проблемой для **ApproximateQAgent** с **SimpleExtractor** (предупреждение: обучение может занять несколько минут):

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic

Если не будет ошибок, то агент с *q*-обучением и аппроксимацией должен будет почти каждый раз выигрывать при использовании этих простых признаков, даже если у вас всего 50 обучающих игр.

В ходе оценивания будет проверяться, обучается ли ваш агень тем же значениям *q*-ценностей и весам признаков, что и эталонная реализация. Чтобы оценить вашу реализацию, запустите автооцениватель и внесите результаты всех экспериментов в отчет:

```
python autograder.py -q q6
```

7.4.11. В задании 6 необходимо в файле **model.py** реализовать в классе **DeepQNetwork** глубокую нейронную сеть, вычисляющую целевые значения Q-ценностей для всех возможных пар (*s, a*), и использовать её в алгоритмах двойного Q-обучения или DDQN, рассмотренных выше.

При реализации контсруктора класса **DeepQNetwork** в методе **__init__()** инициализируйте следующие обязательные параметры и переменные:

self.learning_rate: скорость обучения нейросети, которая будет использоваться оптимизатором;

self.numTrainingGames: число обучающих игр, которые будут использоваться, чтобы собрать выборки (*s, a, s', r*) и провести Q-обучение; обратите внимание, что это число должно быть больше 1000, так как примерно первые 1000 игр используются для исследования и не используются для обновления Q-ценностей.

self.batch_size: размер миниблока данных из буфера воспроизведения опыта, который используется на каждом шаге обновления параметров нейросети на основе алгоритма обратного распространения; этот параметр будет использовать автооценивателем, вам не нужно будет обращаться к нему непрсоредственно.

Определите также следующие методы класса **DeepQNetwork:**

forward(self, states): возвращает результат прямого распространения миниблока данных **states** размером **(batch_size, state_dim)** по нейросети, где **state_dim** – размерность состояния игры; метод возвращает тензор размером **(batch_size, num_actions)**, который содержит прогнозируемые нейросетью Q значения для всех возможных действий в каждом состоянии из **states**;

get_loss(self, states, Q_target): возвращает средний квадрат ошибки между прогнозируемыми нейросетью значениями Q, которые возвращает метод **forward()** и целевыми значениями **Q_targets** (которые рассматриваются как истинные);

gradient_update(): выполняет одну итерацию обновления параметров нейросети **self.parameters()** с использованием одного из оптимизаторов машинного обучения, например, **SGD, Adam**; метод должен выполнять только одно обновление каждого параметра, т.к. цикличность обновления будет обеспечиваться автооценивателем. Чтобы не создавались копии оптимизаторов при каждом вызове **gradient_update(),** рекомендуется не включать вызов конструктора оптимизатора в код метода **gradient_update()**, вместо этого разумно определить оптимизатор в конструкторе класса **DeepQNetwork.** В таком случае экземпляр оптимизатора будет создавться один раз при инициалиации **DeepQNetwork.**

Непосредственно алгоритмы глубокого Q обучения Пакмана реализованы в классе **PacmanDeepQAgent**. Класс позволяет реализовать алгоритм двойного Q-обучения (см. п. 7.2.13, параметр **doubleQ=True**) или двойного DQN (см. п. 7.2.17, параметр **doubleQ=False**) с использованием буфера воспроизведения опыта (см. п. 7.2.18). Просмотрите код класса, который содержит комментарии для лучшего понимания.

Поскольку вам необходимо разработать подходящую архитектуру глубокой нейросети, то познакомьтесь с методом get_features, который возвращает вектор, описывающий состояние игры state. Видно, что состояние игры представляется вектором, содержащим позицию Пакмана (**pacman_state**), позиции приведений (**ghost_state**), позиции капсул с едой (**food_locations**). Размерность этого вектора вычисляется при вызове метода **get_state_dim()**, который вернет размер входа нейросети **state_dim**. В ходе обучения глубокая нейросеть должна по сути по входным состояниям игры научиться формировать скрытые признаки, подходящие для аппроксимации Q-функции. Ваша задача подобрать архитектуру сети, обеспечивающую удовлетворительное прогнозирование значений Q(*s, a*). Начните с простой сети, содержащей один скрытый слой, и постепенно её усложняйте.

После завершения разработки сети выполните автооценивание. В ходе автооценивания агент **DeepQPacman** будет тестироваться на 10 играх после того, как он обучится на **self.numTrainingGames** играх. Если агент выиграет не менее 6 из 10 игр, то тест бужет пройден. Обратите внимание, что алгоритмы DQN не отличается стабильностью. Количество игр, в которых выиграет агент, может меняться для каждого запуска.

Запустите автооцениватель следующей командой и внесите результаты в отчет:

python autograder.py -q q7

7.5. Содержание отчета

Цель работы, описание основных понятий MDP, уравнение Беллмана, описание алгоритмов итераций по значениям и политикам, описание задачи обучения с подкреплением, описание алгоритма RL, основанного на модели, описание алгоритмов TD-обучения и Q-обучения, описание алгоритма Q-обучения с аппроксимацией, алгоритмы двойного Q-обучения и DDQN, код реализованных агентов и функций с комментариями в соответствии с заданиями 1-7, результаты игр на разных полях игры, их анализ, результаты автооценивания заданий, выводы по проведенным экспериментам с разными алгоритмами обучения с подкреплением.

7.6. Контрольные вопросы

7.6.1 Объясните, что понимают под недетерминированными задачами поиска?

7.6.2. Объясните основные понятия Марковского процесса принятия решений.

7.6.3. Что понимается под функцией политики?

7.6.4. Что понимается под функцией полезности?

7.6.5. Какая задача называется эпизодической?

7.6.7. Запишите выражение для вычисления функции полезности для продолжающихся задач.

7.6.8. Как определяется функция ценности состояния?

7.6.9. Как определяется *q*-функция ценности состояния-действия?

7.6.10. Объясните, как строится MDP дерево поиска?

7.6.11. Запишите и объясните уравнение Беллмана для функции ценности состояния.

7.6.12. Сформулируйте и объясните алгоритм итерации по значениям ценности состояний.

7.6.13. Сформулируйте и объясните метод извлечения политики.

7.6.14. Сформулируйте и объясните алгоритм итерации по политикам.

7.6.15. Объясните основные понятия обучения с подкреплением.

7.6.16. Что понимается под исследованиями и эксплуатацией при RL обучении?

7.6.17. Объясните обучение с подкреплением на основе модели.

7.6.18. Как классифицируются алгоритмы обучения с подкреплением без модели?

7.6.19. Объясните алгоритм прямого оценивания (обучение без модели).

7.6.20. Объясните алгоритм обучения на основе временных различий.

7.6.21. Объясните алгоритм *Q*-обучения.

7.6.22. Объясните, что понимают под ε-жадная стратегией.

7.6.23. Объясните алгоритм *Q*-обучения с аппроксимацией.

7.6.24. Что понимают под функции разведки (исследования)?

7.6.25. Объясните алгоритм двойного *Q*-обучения.

7.6.26. Объясните алгоритм двойной глубокой *Q*-сети.

7.6.27. Объясните механизм использования буфера воспризведения опыта.