ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ БЮДЖЕТНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
**«ФИНАНСОВЫЙ УНИВЕРСИТЕТ ПРИ ПРАВИТЕЛЬСТВЕ
РОССИЙСКОЙ ФЕДЕРАЦИИ»
(ФИНАНСОВЫЙ УНИВЕРСИТЕТ)**

Департамент анализа данных
и машинного обучения

Дисциплина: «Технологии анализа данных и машинного обучения»
Направление подготовки: «Прикладная математика и информатика»
Профиль: «Анализ данных и принятие решений в экономике и финансах»
Факультет информационных технологий и анализа больших данных
Форма обучения очная
Учебный 2022/2023 год, 6 семестр

Курсовая работа на тему:

«Разработка диалоговой системы с применением обучения с подкреплением»

Выполнил:

студент группы ПМ20-1

Кудряшов Н.А.

Научный руководитель:

ассистент Блохин Н.В.

Москва 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время диалоговые системы становятся все более популярными и востребованными в различных сферах деятельности, от бизнеса до медицины и образования. Реализация чат-ботов для компаний позволяет существенно автоматизировать процесс взаимодействия с клиентами. Помимо этого, применение методов глубокого обучения позволяет улучшить качество работы диалоговых систем и повысить их эффективность. В частности, использование нейронных сетей позволяет создавать более сложные и гибкие модели, которые могут адаптироваться к различным ситуациям и контекстам.

Актуальность работы заключатся в потребности реализации удобных диалоговых систем, позволяющих улучшить и автоматизировать процесс взаимодействия с человеком.

Целью этой курсовой работы является изучение и реализация диалоговой системы с применением обучения с подкреплением. Данное исследование может дать толчок к разработке более совершенной модели по взаимодействию с клиентом.

Объектом исследования является алгоритм реализации диалога между агентом диалоговой системы и пользователем.

Предметом исследования является алгоритм реализации менеджера диалогов на основе DQN (Deep Q-Network) и DDQN (Double Deep Q-Network).

ГЛАВА 1. ОБУЧЕНИЕ ДИАЛОГОВОЙ СИСТЕМЫ С ИСПОЛЬЗОВАНИЕМ ГЛУБОКОГО ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ.

1.1 Основные понятия в области обучения с подкреплением.

Для правильного понимания терминологии необходимо дать четкие определения некоторых основных объектов обучения с подкреплением:

- Агент: сущность, которая действует в среде и принимает решения, чтобы максимизировать награду.
- Среда: сценарий или окружение, с которым должен столкнуться агент.
- Награда: оценка, предоставляемая агенту, после выполнения определенного действия или задачи.
- Политика (policy): стратегия, применяемая агентом для принятия решения о следующем действии на основе текущего состояния.

Благодаря развитию теории обучения с подкреплением стало возможным сформировать четкое понимание о глубоко укоренившихся психологических и нейробиологических взглядах о животном поведении в открытой среде. Наблюдаемый объект (агент) пытается на основе имеющихся у него данных оптимизировать контроль над окружающей средой. Однако, чтобы успешно использовать обучение с подкреплением в ситуациях, приближенных к сложностям реального мира, агентам необходимо получать эффективное представление об окружающей среде из многомерных входных данных и использовать его для обобщения опыта в новых ситуациях.

Таким образом, основой машинного обучения с подкреплением является взаимодействие агента и некоторой среды, в которой он находится. Целью же агента является выбор правильного действия, следствием чего является числовая мера качества, называемая наградой, которая определяет, насколько хорошо агент выполнил поставленную перед ним задачу. Максимизация этого

значения позволяет формировать более оптимальные действия, адаптируясь тем самым в среде.

1.2 Метод Q-learning.

Для того, чтобы обучить агента принимать оптимальные решения в условиях неопределенности и неизвестности существует множество различных алгоритмов. В представленной работе будет рассмотрен один из них – метод Q-learning (предложен Кристофером Уоткинсоном в 1989 году[1]).

Определим функцию $Q(s, a)$ такую, что для текущего состояния s и действия a она возвращает оценку общего вознаграждения, которого достигнет агент с этого состояния, выполняя последующее действие, следуя некоторой политики. Среди этих возможных стратегий существует несколько оптимальных, определяющих выбор наиболее выгодного действия. Обозначим функцию Q для таких оптимальных политик как Q^*.

В случае, если бы нам была известная истинная функция Q^*, решение было бы крайне простым. Следовало применить «жадную политику», означающую выбор такого действие a из состояния s, которое максимизировало бы значение функции (1.1):

$$Q^* - argmax_a Q^*(s, a) \tag{1.1}$$

Таким образом, задача сводится к поиску хорошей оценки функции Q^* и применению к ней жадной политики.

С этой целью запишем функцию в символьном виде, как сумму наград r за каждое действие (1.2):

$$Q^*(s, a) = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \cdots \tag{1.2}$$

[1] Watkins Ch. - Learning from delayed rewards, PhD. thesis, Cambridge University, 1989.

В таком случае необходимо ввести коэффициент дисконтирования ($\gamma < 1$), гарантирующий, что сумма в формуле конечна. В ином же случае значимость каждого члена в формуле экспоненциально уменьшалась по мере увеличения их количества и в пределе бы становилась нулем. Таким образом, коэффициент дисконтирования γ определяет, насколько функция Q в состоянии s зависит от будущего (определяет, насколько сильно агент предпочитает мгновенную награду в настоящем по сравнению с будущими наградами). Теперь запишем предыдущее уравнение в рекурсивной форме (1.3):

$$Q^*(s,a) = r_0 + \gamma(r_1 + r_2 + r_3 + \cdots) = r_0 + \gamma \max_a Q^*(s', a) \qquad (1.3)$$

Получившаяся формула (1.3) называется уравнением Беллмана. Оно лежит в основе алгоритма Q-обучения (Q-learning). Было доказано, что уравнение сходится к желаемому значению Q^* при условии, что существует конечное число состояний и каждое из пары состояние-действие представлено неоднократно.

```
Set values for learning rate α、 discount rate γ、 reward matrix R
Initialize Q(s,a) to zeros
Repeat for each episode,do
        Select state s randomly
        Repeat for each step of episode,do
                Choose a from s using ε-greedy policy or Boltzmann policy
                Take action a obtain reward r from R, and next state s'
                Update Q(s,a)← Q(s,a) + α[r + γ max_a Q(s',a') - Q(s,a)]
                Set s = s'
                Until s is the terminal state
         End do
End do
```

Рис. 1. Псевдокод, описывающий алгоритм Q-learning.

1.3 Deep Q Networks (DQN).

В 2013 году исследовали из «Google DeepMind Technologies» предложили реализацию алгоритма Q-learning в формате нейронной сети для обучения агента семи играм на Atari 2600[2] без каких-либо корректировок в архитектуре системы консоли[3]. Данная работа положила начало применению нейронных сетей для реализации глубокого обучения с подкреплением.

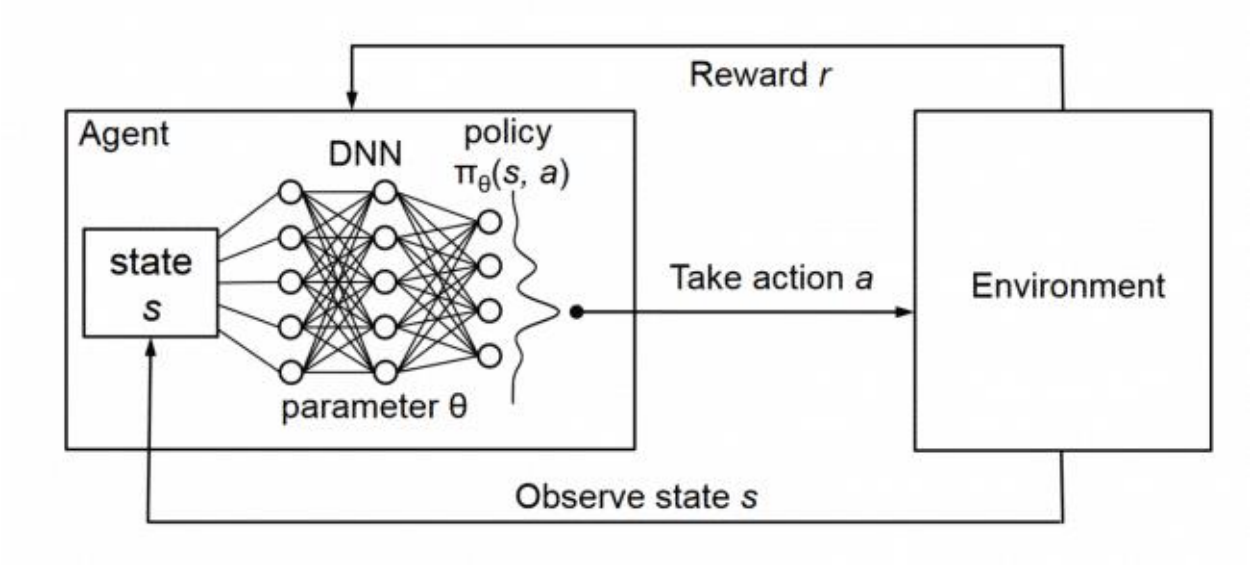

Рис. 2. Структура алгоритма глубокого обучения с подкреплением на основе Deep Q-network.

Deep Q-network (DQN) это многослойная нейронная сеть, которая для текущего состояния s возвращает вектор действий $Q(s,.;\theta)$, где θ – параметры нейронной сети. Для n-мерного пространства состояний и

[2] Atari 2600 - одна из первых игровых приставок, созданная американской компанией Atari в 1977 году. Обучение нейронной сети происходило на основе игр «Pong», «Breakout», «Space Invaders», «Seaquest» и «Beam Rider».

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. – «Playing Atari with Deep Reinforcement Learning», 2013 г.

пространства действий, содержащего m действий, нейронная сеть является функцией от $\mathbb{R}^n$ к $\mathbb{R}^m$.

Двумя важными компонентами алгоритма DQN (предложенных в работе «Human-level control through deep reinforcement»[4] от 2015 года), дополняющих исследования предшественников, стало использование целевой нейронной сети (target network) и системы воспроизведения опыта (experience replay). Целевая система с параметрами θ^- дополнилась параметром τ, отвечающим за периодичность копирование шагов, так что $\theta^- = \theta_t$ и остаются в истории на всех остальных шагах:

$$Q(s, a; \theta) \equiv r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t^-)\,. \qquad (1.4)$$

Для воспроизведения опыта наблюдаемые изменения сохраняются в течение некоторого времени и равномерно выбираются из этого массива памяти для обновления сети. Подобный подход позволил существенно улучшить производительность алгоритма DQN.

1.4 Double Q-learning и Double Deep Q-Network (DDQN).

Оператор max в формулах для стандартного Q-learning (1.3) и DQN (1.4) использует одни и те же значения как для принятия решения, так и для оценки действия. Это повышает вероятность выбора завышенных значений, что может привести к чрезмерно оптимистичным оценкам результатов. Чтобы

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg & Demis Hassabis. «Human-level control through deep reinforcement learning», 2015 г.

предотвратить это, необходимо отделить выбор от произведения оценки. В этом и заключается идея Double Q-learning[5].

Решение включает использование сразу двух функций Q, одна из которых используется для выбора действия (Q), а другая для оценки этого действия (Q') (1.5):

$$Q^*(s,a) = r_t + \gamma Q\big(s_{t+1}, argmax_{a'} Q'(s_t, a_t)\big) \qquad (1.5)$$

Algorithm 1 Double Q-learning

Initialize Q^A, Q^B, s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s',a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s',a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

Рис. 3. Псевдокод, описывающий алгоритм Double Q-learning.

При реализации Double Deep Q-Network (DDQN) существуют сразу две отдельные нейронные сети с разными весами – одна из которых занимается выбором действия, а другая оценкой этого действия:

$$Q^*(s,a;\theta) \equiv r_{t+1} + \gamma Q(s_{t+1}, argmax_{a'} Q(s_{t+1}, a; \theta_t); \theta'_t). \qquad (1.6)$$

Следует обратить внимание, что выбор действия в операторе $argmax$ по-прежнему зависит от весов θ_t. Это означает, что мы производим выбор действия на основе «жадной политики» в соответствии с текущими

[5] Hado van Hasselt, Arthur Guez, David Silver. «Deep Reinforcement Learning with Double Q-learning», 2015 г.

значениями, определенными в θ_t. Однако также используется второй набор весов θ'_t, производящий оценку значения выбранной стратегии.

ГЛАВА 2. РЕАЛИЗАЦИЯ ДИАЛОГОВОЙ СИСТЕМЫ.

Далее более подробно рассмотрим практическую реализацию диалоговой системы на основе разобранной модели.

2.1 Основные этапы разработки и обучения.

Разработку целевого (goal-oriented) чат-бота можно разделить на три основных подзадачи:

- Менеджер диалогов (The Dialogue Manager (DM)) – основная часть алгоритма, которая состоит из трекера состояния диалога (Dialogue State Tracker (DST)) и самого агента обучения с подкреплением.
- Модуль восприятия естественного языка (Natural Language Understanding (NLU)). Его главной задачей является преобразование введённого пользователем сообщения на естественном языке в объекты, с которыми может работать алгоритм.
- Модуль генерации естественного языка (Natural Language Generator (NLG)). Обратная NLU задача генерации ответа на естественном языке для пользователя.

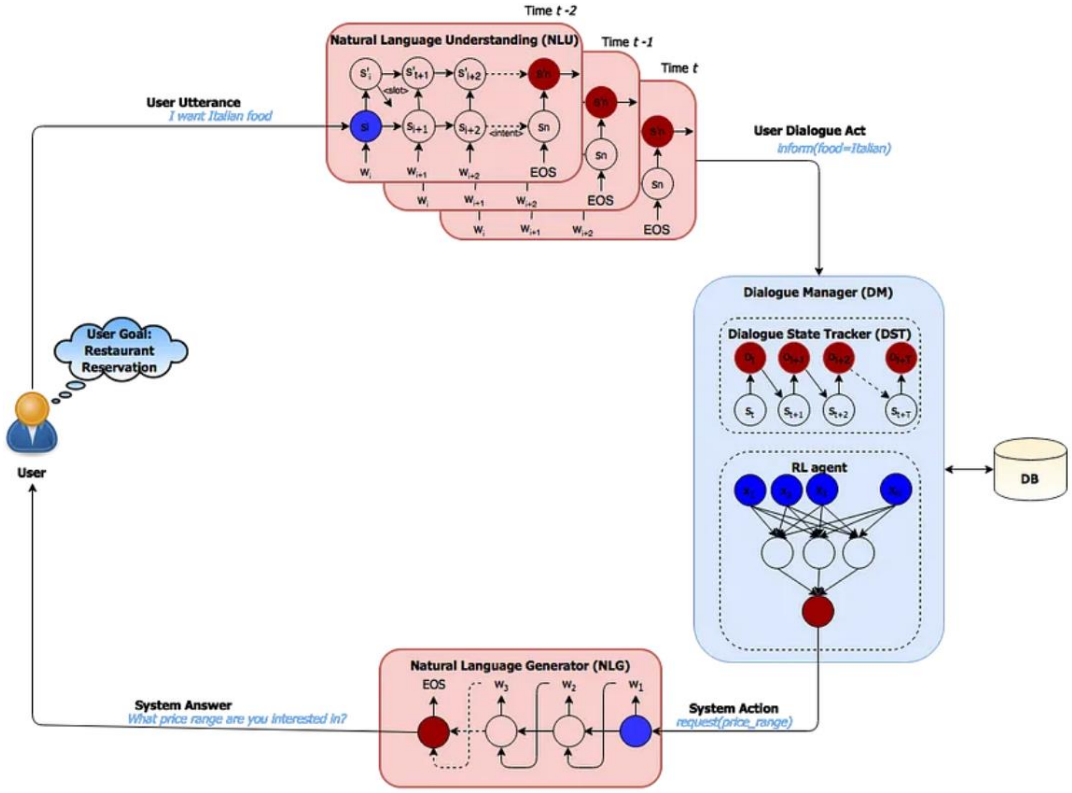

Рис. 4. Схема диалоговой системы целевого чат-бота.

На представленном рисунке описана схема взаимодействия пользователя с диалоговой средой. Запрос человека обрабатывается компонентом NLU в семантический фрейм, который далее отдается агенту. Переданная пользователем информация сохраняется в историю диалога трекером состояния и далее выступает в качестве входных данных для нейронной сети чат-бота. Во время обработки агентом запроса, также возможно обращение к базе данных, в которой хранится дополнительная информация для чат-бота с целью использования в формировании удовлетворительного ответа. После созданный агентом ответ обрабатывается компонентом NLG в естественный язык для правильного восприятия пользователем.

В представленной разработке будет в подробностях рассмотрен этап менеджера диалогов.

Для правильного восприятия алгоритма введем некоторые обозначения:

- Эпизодом при обучении агента будет называться одна беседа из цикла.

- Раунд – часть беседы с одним запросом пользователя и ответом от системы.
- Слот – пара ключ-значения, в котором ключом выступает определенный параметр рассматриваемой цели, а значение – возможный вариант этого параметра.

Далее подробнее рассмотрим каждый из элементов диалоговой системы.

2.2 Используемые данные.

В качестве данных для реализации диалоговой системы был составлен список из различных московских ресторанов. Задачей агента будет подобрать подходящий пользователю стол для бронирования. Каждый возможный вариант обладает следующими параметрами:

- «кухня» - определяет стиль или национальную принадлежность блюд, подаваемых в ресторане.
- «район» - район города Москвы, в котором находится ресторан.
- «название» - уникальное имя, которое используется для идентификации конкретного ресторана.
- «яндекс_карты» - рейтинг ресторана в соответствии с оценками пользователей в приложении «Яндекс: Карты».
- «гугл_карты» - рейтинг ресторана в соответсвии с оценками пользователей в приложжении «Google: Карты».
- «дата» - день недели, на которую будет оформлена бронь.
- «количество_человек» - количество гостей, которые будут присутствовать в ресторане.
- «количество_детей» - количество детей, которые будут присутствовать в ресторане.

На основе этих параметров было составлено три датасета для обучения модели:

- «res_db.pkl» - датасет, в котором хранятся все возможные значения каждого из параметров в формате словаря. Эта база данных будет использоваться агентом для подбора подходящего пользователю варианта по конкретному критерию.
- «res_dict.pkl» - датасет, в котором собраны все варианты доступных столов для бронирования в каждом ресторане.
- «res_user_goals.pkl» - случайно сгенерированные на основе имеющихся данных возможные запросы людей, которые будут применяться в пользовательской симуляции.

2.3 Агент на базе Deep Q-Network и обучение.

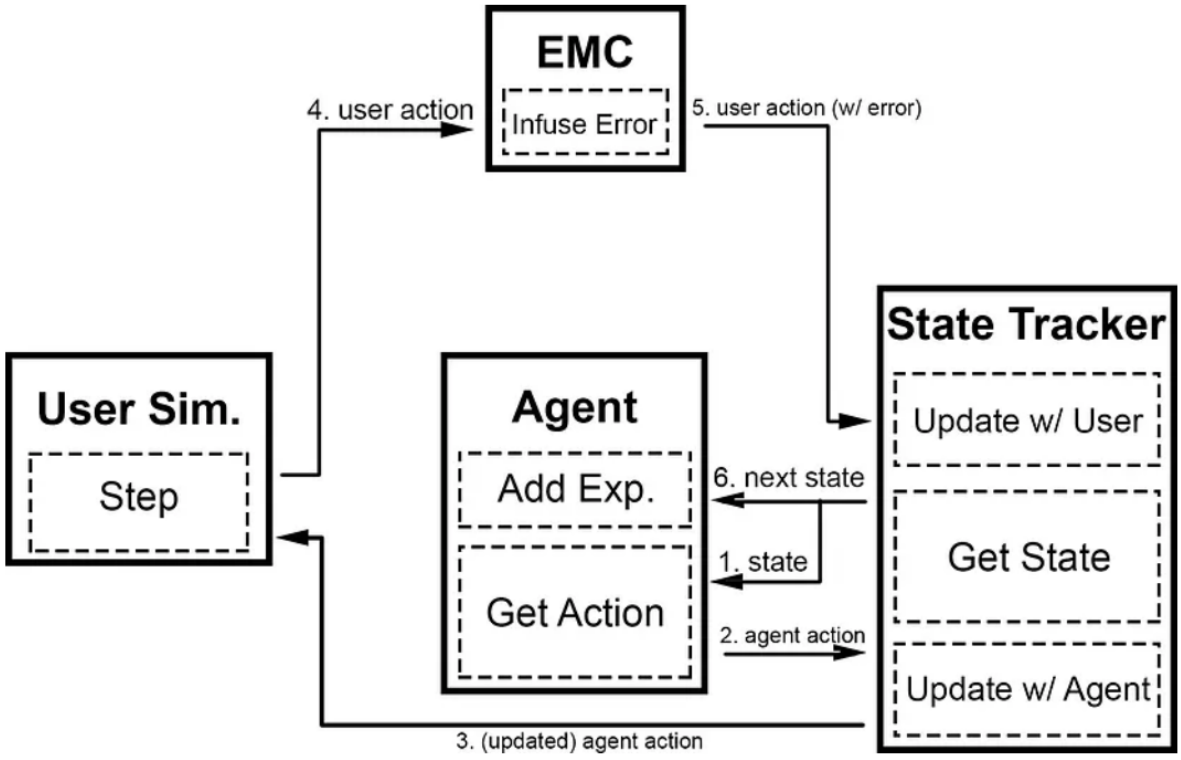

Рис. 5. Цикл обучения диалоговой системы.

На представленной диаграмме изображен один раунд полного цикла тренировки. Эта система состоит из четырёх основных блоков: агента, трекера состояния диалога, пользователя или симуляции пользователя, а также модуля

контроля ошибок. Далее более подробно опишем каждый шаг, описанный на рисунке:

- Трекеру состояния необходимо подготовить информацию о текущем состоянии диалога из истории прошлого раунда или инициализировать новое, если это начало нового диалога, и отправить эту информацию в обработчик агента.
- Обновленное информацией из базы данных и состоянием из истории эпизода формируется действие агента в текущем раунде и далее фиксируется трекером состояния, сохраняя его в историю.
- Действие агента считывается методом пользовательской симуляции. На основе определенных правил генерируется ответ и награда.
- Действие пользователя подвергаются воздействию модуля контроля ошибок для того, чтобы добавить случайную неточность в ответ.
- Действие пользователя с ошибкой также сохраняется трекером состояния.
- Оценка, выданная в текущем раунде, также фиксируется и начинается новый этап цикла, где входящими данными для агента будет информация из ответа пользователя в предыдущем раунде.

Каждое действие пользователя и агента определяется их намерением (типом действия, которое воспроизводит сторона):

- «inform» используется с целью проинформировать собеседника о желаемом значении конкретного параметра (ввести по нему ограничение).
- «request» используется с целью запроса от собеседника информации по определенному параметру.
- «thanks» - выражение благодарности, используемое пользователем, чтобы указать агенту, что он сделал что-то хорошее или что человек хочет завершить диалог.

- «match found» - используется только агентом для того, чтобы проинформировать пользователя о найденном совпадении с его целью.
- «reject» - используется только пользователем на действие агента с намерением «match found», чтобы указать, что совпадение не соответствует желаниям (ограничениям) пользователя.
- «done» - используется только агентом, чтобы проверить достиг ли он цели пользователя по завершению диалога.

Главной задачей для goal-oriented чат-бота является умение общается с реальными пользователями для достижения поставленной ими цели. Иными словами, агенту необходимо правильно определить текущее состояние диалога (посредством истории, сохраняемой трекером состояния) и воспроизвести действие, близкое к оптимальному.

Для реализации модели агента используется библиотека языка «Python» - «Keras». Сама же модель представляет собой однослойную нейронную сеть со скрытым слоем, основанную на жадном алгоритме в соответствии уравнению Беллмана. Помимо этого, реализована возможность создания, как и DQN модели, так и DDQN нейронную сеть.

2.4 Трекер состояния диалога.

Для того, чтобы у чат-бота была возможность сделать правильный выбор при формировании действия, ему необходимо четко понимать в каком состоянии находится текущий разговор. Для этого существует трекер состояния. Он обновляет историю диалога, собирая действия пользователя и агента по мере их выполнения. Помимо этого, трекер состояния также фиксирует все заполненные слоты информации, которые содержались в любых действиях агента и пользователя до сих пор в текущем эпизоде.

В зависимости от того, каким было намерение пользователя в предыдущем шаге, трекер состояния может по-разному предопределить

формирование ответа от агента. В частности, необходимо рассмотреть последовательность формирования действия агента при намерениях «inform», «request» и «match found».

В случае намерения «inform» агенту необходимо заполнить слот с информацией конкретным значением, с целью проинформировать пользователя о новом ограничении в поиске. Для достижения этого трекер состояния обращается к базе данных, в которой хранятся возможные варианты его заполнения. Найдя подходящую опцию, чат-бот добавляет ее к уже имеющимся ограничениям и далее сообщает пользователю.

При намерении «request» агент осуществляет запрос к пользователю с целью заполнить определенный слот с ограничением (от пользователя ожидается намерение «inform»).

В случае намерения «match found» агент также обращается к базе данных с целью найти полное совпадение со всеми имеющимися значениями слотов ограничений. Если же такого варианта нет, чат-бот сообщает о неудаче. В ином же случае агент информирует пользователя об успешно подобранном для него варианте.

Помимо полезной информации о предыдущих действиях сторон, которая передается трекером состояний, агент также информируется и о порядке раунда. Это делается для того, чтобы сам диалог не шел слишком долго. Так если номер текущего раунда близок к максимальному значению, агент с большей вероятностью предпримет действие с намерением «match found».

2.5 Симуляция пользователя.

Пользовательская симуляция необходима, чтобы предварительно обучить модель для дальнейшего взаимодействия с обычным человеком. Симуляция пользователя будет строиться на основе agenda-based системы (системы "на основе повестки дня"). Это означает, что у пользователя есть цель

в диалоге, и он предпринимает действия в соответствии с этой целью, отслеживая при этом текущее состояние разговора, чтобы в дальнейшем предпринимать обоснованные действия. На каждом этапе диалога действие пользователя создается в ответ на действие агента с использованием в основном детерминированных правил, а также нескольких стохастических правил для создания разнообразия ответов.

Как уже было сказано, пользовательская симуляция отслеживает историю диалога с целью создания действия на каждом этапе диалога. В частности, сохраняемая информация о состоянии представляет собой список из четырех различных словарей:

- «rest_slots» - слоты информирования и запросов, которые еще не были использованы ни агентом, ни пользователем. В случае успешного завершения диалога этот словарь должен оказаться полностью пустым.
- «history_slots» - слоты информации и запросов, которые уже были использованы агентом и пользователем. В случае успешного завершения диалога этот словарь должен быть полностью заполнен значениями из «rest_slots».
- «request_slots» - слоты запросов, которые пользователь хочет использовать в ближайших или будущих действиях.
- «inform_slots» - слоты информации (или ограничений), которые пользователь собирается сообщить в блажащих или будущих действиях.
- «intent» - намерение действия, сформированного на текущем шаге.

Анализируя историю диалога и предпринятое агентом действие на текущем шаге, симуляция пользователя формирует ответное действие, а также оценивает выбор чат-бота. Алгоритм формирования ответа различен в зависимости от намерения агента на предыдущем шаге.

Если намерением агента является «request», то существуют четыре случая генерация ответа:

- Если агент запрашивает что-то, что находится в слотах информации цели пользовательской симуляции, и оно не было проинформировано до текущего момента, то необходимо передать нужное значение из самой цели.
- Если агент запрашивает что-то, что находится в слотах запроса цели пользовательской симуляции, и оно уже было сообщено до текущего момента, то необходимо передать нужное значение из массива истории («history_slots»).
- Если агент запрашивает что-то, что находится в слотах запроса цели пользовательской симуляции, и оно еще не было сообщено до текущего момента, то необходимо запросить этот слот с дополнительным ограничением.
- В иных случаях в качестве запрашиваемого слота передается значение «anything», обозначающее любой возможный вариант.

Если намерением агента является «inform», то существуют два случая генерация ответа:

- Если агент сообщает что-то, что находится в слотах информации цели пользовательской симуляции, и значение, которое он передал, не совпадает, то возвращается правильное значение.
- В иных случаях выбирается какой-либо слот для запроса или информирования.

Если намерением агента является «match found», необходимо провести проверку по следующим трем пунктам:

- «default_slot» должно хранить актуальное значение номера бронирования, в не «no match available», означающее отсутствие подходящего пользователю варианта.

- Все ограничения из цели пользовательской симуляции должны содержаться в действии агента, а их значения должны быть одинаковыми.

При выполнении этих условий формируется ответ с намерением «thanks», в ином же случае инициализируется намерение «reject».

2.6 Контроль ошибок.

После того, как действие пользователя получено, в него с некоторой вероятностью вноситься ошибка. В статье «End-to-End Task-Completion Neural Dialogue Systems»[6] было обнаружено, что использование модуля контроля ошибок (EMC) позволяет агенту быстрее обучаться и учитывать возможные ошибки обычных пользователей.

Модуль контроля ошибок может внести следующие типы ошибок:

- Замена значения случайным в каком-либо слоте (в независимости от его типа).
- Замена всего слота: выбирается случайный ключ и значение для этого слота (в независимости от его типа).
- Удаления одного из слотов (в независимости от его типа).

Тем самым мы добиваемся более качественного обучения модели и адаптируем ее к реальным условиям.

2.7 Запуск диалоговой системы.

Перед запуском диалоговой системы необходимо ознакомиться с гиперпараметрами модели, собранными в один файл (см. «constants.json»). Из основных следует выделить:

[6] Xiujun Li, Yun-Nung Chen, Lihong Li, Jianfeng Gao, Asli Celikyilmaz: «End-to-End Task-Completion Neural Dialogue Systems». – 2018 г.

- «learning_rate» - размер шага на каждой итерации, с которым алгоритм оптимизации приближается к минимуму функции потерь (по умолчанию равен 0,001).
- «dqn_hidden_size» - размерность скрытого слоя (по умолчанию равен 80)
- «gamma» - коэффициент дисконтирования γ, используемый в уравнениях DQN (1.4) и DDQN (1.6) (по умолчанию равен 0.9).
- «slot_error_mode» - определяет тип ошибки, которая будет использоваться в EMC (по умолчанию равен 0 – ошибка добавляется на уровне значения слота)
- «slot_error_prob» - вероятность внесения ошибки на уровне значения слота (по умолчанию равно 0.1)
- «intent_error_prob» - вероятность внесения ошибки на уровне изменения всего слота (по умолчанию равно 0)

Обучение модели происходило в несколько подходов (модель подвергалась дообучению на основе сохраненных значений весов). Вероятность успешного диалога, по результатам 15300 проведенных диалогов, достигла значения 81% для моделей DQN и DDQN.

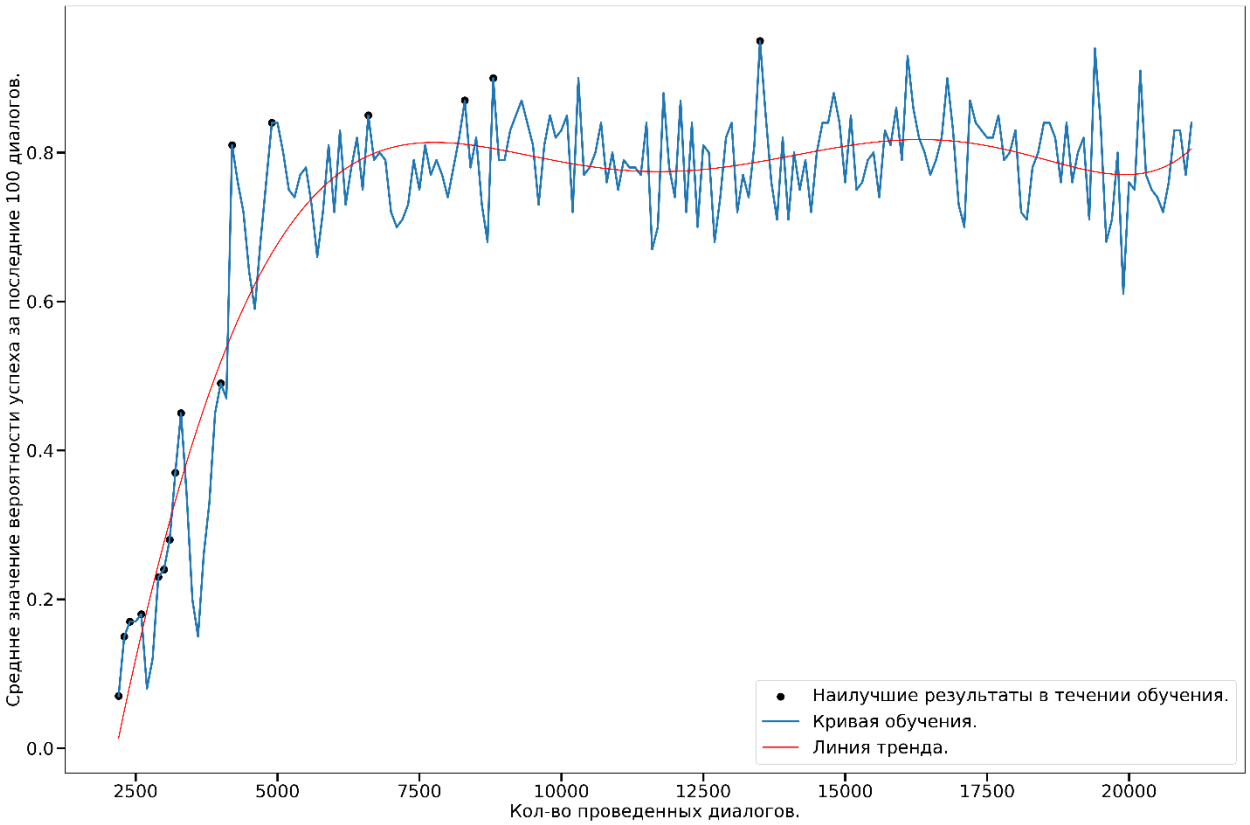

Рис. 6. Кривая обучения алгоритма DQN.

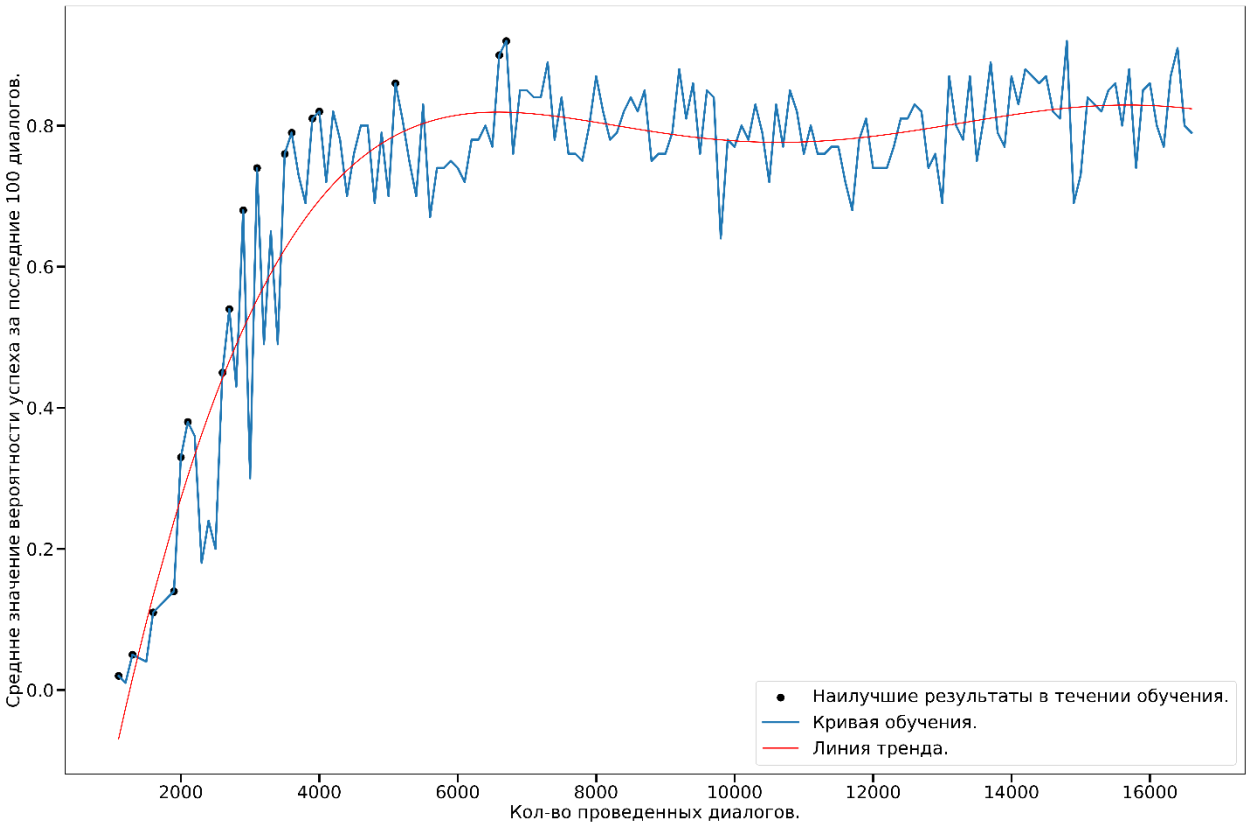

Рис. 7. Кривая обучения алгоритма DDQN.

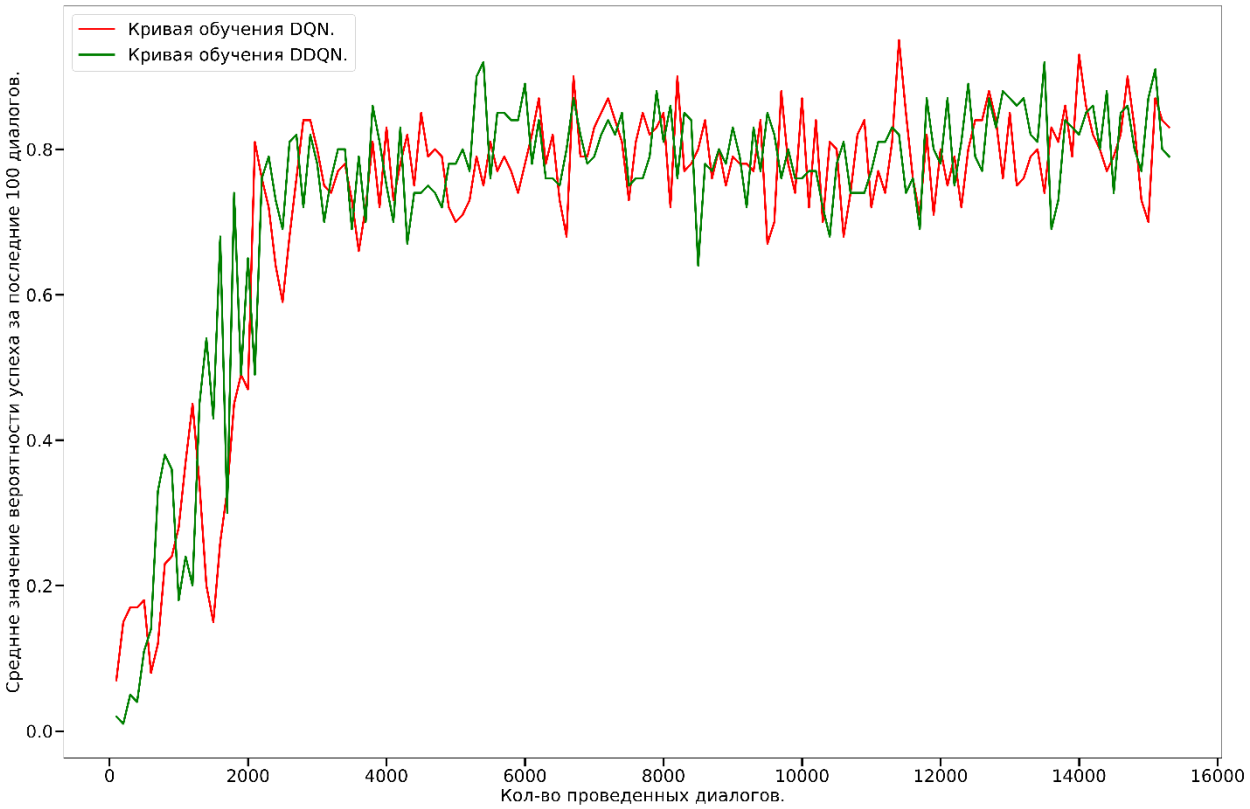

Рис. 8. Сравнение кривых обучений для алгоритмов DQN и DDQN.

Награда за раунд: 40

{'intent': 'inform', 'inform_slots': {'время': '18:30'}, 'request_slots': {}, 'round': 1, 'спикер': 'Агент'}

{'intent': 'inform', 'request_slots': {}, 'inform_slots': {'дата': 'завтра'}, 'round': 1, 'сприкер': 'Пользователь'}

{'intent': 'request', 'inform_slots': {}, 'request_slots': {'количество_человек': 'UNK'}, 'round': 2, 'спикер': 'Агент'}

{'intent': 'inform', 'request_slots': {}, 'inform_slots': {'количество_человек': '3'}, 'round': 2, 'сприкер': 'Пользователь'}

{'intent': 'inform', 'inform_slots': {'район': 'текстильщики'}, 'request_slots': {}, 'round': 3, 'спикер': 'Агент'}

{'intent': 'inform', 'request_slots': {}, 'inform_slots': {'город': 'москва'}, 'round': 3, 'сприкер': 'Пользователь'}

{'intent': 'match_found', 'inform_slots': {'город': 'москва', 'область': 'москва', 'кухня': 'итальянская', 'район': 'текстильщики', 'название': 'da pino', 'яндекс_карты': '4.4', 'гугл_карты': '4.4', 'дата': 'завтра', 'время': '18:30', 'количество_человек': '3', 'бронь': '25'}, 'request_slots': {}, 'round': 6, 'спикер': 'Агент'}

{'intent': 'thanks', 'request_slots': {}, 'inform_slots': {}, 'round': 6, 'сприкер': 'Пользователь'}

{'intent': 'done', 'inform_slots': {}, 'request_slots': {}, 'round': 7, 'спикер': 'Агент'}

{'intent': 'done', 'request_slots': {}, 'inform_slots': {}, 'round': 7, 'сприкер': 'Пользователь'}

Рис. 9. Пример симулированного диалога.

ЗАКЛЮЧЕНИЕ

Целью этой курсовой работы являлось изучение и реализация диалоговой системы с применением машинного обучения с подкреплением на примере бронирования места в ресторане. В результате работы мне удалось реализовать обучение на основе алгоритмов DQN и DDQN. Высокая вероятность успешного завершения диалога позволяет с уверенностью сказать, что модель можно использовать для общения с реальными пользователями. Дополнительные исследования в области генерации и понимания естественных текстов для чат-бота, на мой взгляд, позволят использовать подобные диалоговые системы в реальных условиях, что сильно облегчит взаимодействие компаний со своей клиентской базой.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.

1. Xiujun Li, Yun-Nung Chen, Lihong Li, Jianfeng Gao, Asli Celikyilmaz: «End-to-End Task-Completion Neural Dialogue Systems». – 2018 г.
2. Xiujun Li, Zachary C. Lipton, Bhuwan Dhingra, Lihong Li, Jianfeng Gao, Yun-Nung Chen: «A User Simulator for Task-Completion Dialogues». 2017 г.
3. Jaromír Janisch: «Let’s make a DQN» [Электронный ресурс]. - URL: https://jaromiru.com/2016/09/27/lets-make-a-dqn-theory/ – 2016 г.
4. И. Б. Широков, С. В. Колесова, В. А. Кучеренко, М. Ю. Серебряков: «Анализ технологий глубокого обучения с подкреплением для систем машинного зрения» // Известия ТулГУ. – 2022 г. – С. 118–120.
5. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, Demis Hassabis: «Human-level control through deep reinforcement learning» // NATURE. – 2015 г.
6. Max Brenner: «Training a Goal-Oriented Chatbot with Deep Reinforcement Learning» // Towards Data Science [Электронный ресурс]. - URL: https://towardsdatascience.com/training-a-goal-oriented-chatbot-with-deep-reinforcement-learning-part-i-introduction-and-dce3af21d383 – 2018 г.
7. Hado van Hasselt, Arthur Guez, David Silver: «Deep Reinforcement Learning with Double Q-learning» // Google DeepMind. – 2015 г.
8. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. – «Playing Atari with Deep Reinforcement Learning» // DeepMind Technologies. – 2013 г.
9. Chris Yoon: «Double Deep Q Networks» // Towards Data Science [Электронный ресурс]. - URL: https://towardsdatascience.com/double-deep-q-networks-905dd8325412 – 2019 г.

10. Christopher Watkins: «Learning from delayed rewards» // «King's College». – 1989 г.

ПРИЛОЖЕНИЕ А. ОПИСАНИЕ ПРОЦЕССОРА.

Тип процессора: Intel(R) Core (TM) i7-10700K CPU @ 3.80GHz 3.79 GHz.

Память: 32,0 ГБ (доступно: 31,9 ГБ).

Кэш L2: 2048 КБ.

Кэш L3: 16384 КБ.

ПРИЛОЖЕНИЕ Б. ПРОГРАММНЫЙ КОД.

```
from collections import defaultdict
import random, copy
import numpy as np
import re
import pickle
import json
import copy
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from keras.layers import Dropout
import random
import time

class DQNAgent:
    def __init__(self, state_size, constants):
        self.C = constants['agent']
        self.memory = []
        self.memory_index = 0
        self.max_memory_size = self.C['max_mem_size']
        self.eps = self.C['epsilon_init']
        self.vanilla = self.C['vanilla']
        self.lr = self.C['learning_rate']
        self.gamma = self.C['gamma']
        self.batch_size = self.C['batch_size']
        self.hidden_size = self.C['dqn_hidden_size']
        self.load_weights_file_path = self.C['load_weights_file_path']
        self.save_weights_file_path = self.C['save_weights_file_path']
        if self.max_memory_size < self.batch_size:
            raise ValueError('Максимальный размер памяти должен быть не
меньше размера батча!')
        self.state_size = state_size
        self.possible_actions = agent_actions
        self.num_actions = len(self.possible_actions)
        self.rule_request_set = rule_requests
        self.beh_model = self._build_model()
        self.tar_model = self._build_model()
        self._load_weights()
        self.reset()
    def _build_model(self):
        model = Sequential()
        model.add(Dense(self.hidden_size, input_dim=self.state_size,
activation='relu'))
```

```
        model.add(Dense(self.num_actions, activation='linear'))
        model.compile(loss='mse', optimizer=Adam(learning_rate=self.lr))
        return model
    def reset(self):
        self.rule_current_slot_index = 0
        self.rule_phase = 'not done'
    def get_action(self, state, use_rule=False):
        if self.eps > random.random():
            index = random.randint(0, self.num_actions - 1)
            action = self._map_index_to_action(index)
            return index, action
        else:
            if use_rule:
                return self._rule_action()
            else:
                return self._dqn_action(state)

    def _rule_action(self):
        if self.rule_current_slot_index < len(self.rule_request_set):
            slot = self.rule_request_set[self.rule_current_slot_index]
            self.rule_current_slot_index += 1
            rule_response = {'intent': 'request', 'inform_slots': {}, 'request_slots': {slot:
'UNK'}}
        elif self.rule_phase == 'not done':
            rule_response = {'intent': 'match_found', 'inform_slots': {}, 'request_slots':
{}}
            self.rule_phase = 'done'
        elif self.rule_phase == 'done':
            rule_response = {'intent': 'done', 'inform_slots': {}, 'request_slots': {}}
        index = self._map_action_to_index(rule_response)
        return index, rule_response

    def _map_action_to_index(self, response):
        for (i, action) in enumerate(self.possible_actions):
            if response == action:
                return i
        raise ValueError(f'Ответ: {response} не найден в возможных действиях')

    def _dqn_action(self, state):
        index = np.argmax(self._dqn_predict_one(state))
        action = self._map_index_to_action(index)
        return index, action

    def _dqn_predict_one(self, state, target=False):
```

```
        return self._dqn_predict(state.reshape(1, self.state_size),
target=target).flatten()

    def _map_index_to_action(self, index):
        for (i, action) in enumerate(self.possible_actions):
            if index == i:
                return copy.deepcopy(action)
        raise ValueError(f'Индекс: {index} не входит в круг возможных действий')

    def _dqn_predict(self, states, target=False):
        if target:
            return self.tar_model.predict(states, verbose=0)
        else:
            return self.beh_model.predict(states, verbose=0)

    def add_experience(self, state, action, reward, next_state, done):
        if len(self.memory) < self.max_memory_size:
            self.memory.append(None)
        self.memory[self.memory_index] = (state, action, reward, next_state, done)
        self.memory_index = (self.memory_index + 1) % self.max_memory_size

    def empty_memory(self):
        self.memory = []
        self.memory_index = 0

    def is_memory_full(self):
        return len(self.memory) == self.max_memory_size

    def train(self):
        num_batches = len(self.memory) // self.batch_size
        for b in range(num_batches):
            batch = random.sample(self.memory, self.batch_size)
            states = np.array([sample[0] for sample in batch])
            next_states = np.array([sample[3] for sample in batch])
            assert states.shape == (self.batch_size, self.state_size), f'Размерность
состояния: {states.shape}'
            assert next_states.shape == states.shape
            beh_state_preds = self._dqn_predict(states)
            if not self.vanilla:
                beh_next_states_preds = self._dqn_predict(next_states)
            tar_next_state_preds = self._dqn_predict(next_states, target=True)
            inputs = np.zeros((self.batch_size, self.state_size))
            targets = np.zeros((self.batch_size, self.num_actions))
            for i, (s, a, r, s_, d) in enumerate(batch):
```

```
            t = beh_state_preds[i]
            if not self.vanilla:
                t[a] = r + self.gamma *
tar_next_state_preds[i][np.argmax(beh_next_states_preds[i])] * (not d)
            else:
                t[a] = r + self.gamma * np.amax(tar_next_state_preds[i]) * (not d)
            inputs[i] = s
            targets[i] = t
        self.beh_model.fit(inputs, targets, epochs=1, verbose=0)

    def copy(self):
        self.tar_model.set_weights(self.beh_model.get_weights())

    def save_weights(self):
        if not self.save_weights_file_path:
            return
        beh_save_file_path = re.sub(r'\.h5', r'_beh.h5', self.load_weights_file_path)
        self.beh_model.save_weights(filepath=beh_save_file_path,save_format='h5')
        tar_save_file_path = re.sub(r'\.h5', r'_tar.h5', self.load_weights_file_path)
        self.tar_model.save_weights(filepath=tar_save_file_path,save_format='h5')

    def _load_weights(self):
        if not self.load_weights_file_path:
            return
        beh_load_file_path = re.sub(r'\.h5', r'_beh.h5', self.load_weights_file_path)
        self.beh_model.load_weights(beh_load_file_path)
        tar_load_file_path = re.sub(r'\.h5', r'_tar.h5', self.load_weights_file_path)
        self.tar_model.load_weights(tar_load_file_path)

class StateTracker:
    def __init__(self, database, constants):
        self.db_helper = DBQuery(database)
        self.match_key = usersim_default_key
        self.intents_dict = convert_list_to_dict(all_intents)
        self.num_intents = len(all_intents)
        self.slots_dict = convert_list_to_dict(all_slots)
        self.num_slots = len(all_slots)
        self.max_round_num = constants['run']['max_round_num']
        self.none_state = np.zeros(self.get_state_size())
        self.reset()

    def get_state_size(self):
        return 2 * self.num_intents + 7 * self.num_slots + 3 + self.max_round_num
```

```
    def reset(self):
        self.current_informs = {}
        self.history = []
        self.round_num = 0

    def print_history(self):
        for action in self.history:
            print(action)

    def get_state(self, done=False):
        if done:
            return self.none_state
        user_action = self.history[-1]
        db_results_dict =
self.db_helper.get_db_results_for_slots(self.current_informs)
        last_agent_action = self.history[-2] if len(self.history) > 1 else None
        user_act_rep = np.zeros((self.num_intents,))
        user_act_rep[self.intents_dict[user_action['intent']]] = 1.0
        user_inform_slots_rep = np.zeros((self.num_slots,))
        for key in user_action['inform_slots'].keys():
            user_inform_slots_rep[self.slots_dict[key]] = 1.0
        user_request_slots_rep = np.zeros((self.num_slots,))
        for key in user_action['request_slots'].keys():
            user_request_slots_rep[self.slots_dict[key]] = 1.0
        current_slots_rep = np.zeros((self.num_slots,))
        for key in self.current_informs:
            current_slots_rep[self.slots_dict[key]] = 1.0
        agent_act_rep = np.zeros((self.num_intents,))
        if last_agent_action:
            agent_act_rep[self.intents_dict[last_agent_action['intent']]] = 1.0
        agent_inform_slots_rep = np.zeros((self.num_slots,))
        if last_agent_action:
            for key in last_agent_action['inform_slots'].keys():
                agent_inform_slots_rep[self.slots_dict[key]] = 1.0
        agent_request_slots_rep = np.zeros((self.num_slots,))
        if last_agent_action:
            for key in last_agent_action['request_slots'].keys():
                agent_request_slots_rep[self.slots_dict[key]] = 1.0
        turn_rep = np.zeros((1,)) + self.round_num / 5.
        turn_onehot_rep = np.zeros((self.max_round_num,))
        turn_onehot_rep[self.round_num - 1] = 1.0
        kb_count_rep = np.zeros((self.num_slots + 1,)) +
db_results_dict['matching_all_constraints'] / 100.
        for key in db_results_dict.keys():
```

```
            if key in self.slots_dict:
                kb_count_rep[self.slots_dict[key]] = db_results_dict[key] / 100.
        kb_binary_rep = np.zeros((self.num_slots + 1,)) +
np.sum(db_results_dict['matching_all_constraints'] > 0.)
        for key in db_results_dict.keys():
            if key in self.slots_dict:
                kb_binary_rep[self.slots_dict[key]] = np.sum(db_results_dict[key] > 0.)
        state_representation = np.hstack(
            [user_act_rep, user_inform_slots_rep, user_request_slots_rep,
agent_act_rep, agent_inform_slots_rep,
             agent_request_slots_rep, current_slots_rep, turn_rep, turn_onehot_rep,
kb_binary_rep,
             kb_count_rep]).flatten()
        return state_representation

    def update_state_agent(self, agent_action):
        if agent_action['intent'] == 'inform':
            assert agent_action['inform_slots']
            inform_slots = self.db_helper.fill_inform_slot(agent_action['inform_slots'],
self.current_informs)
            agent_action['inform_slots'] = inform_slots
            assert agent_action['inform_slots']
            key, value = list(agent_action['inform_slots'].items())[0]
            assert key != 'match_found'
            assert value != 'PLACEHOLDER', f'KEY: {key}'
            self.current_informs[key] = value
        elif agent_action['intent'] == 'match_found':
            assert not agent_action['inform_slots'], 'Невозможно передать
ограничение при действии match_found!'
            db_results = self.db_helper.get_db_results(self.current_informs)
            if db_results:
                key, value = list(db_results.items())[0]
                agent_action['inform_slots'] = copy.deepcopy(value)
                agent_action['inform_slots'][self.match_key] = str(key)
            else:
                agent_action['inform_slots'][self.match_key] = 'no match available'
            self.current_informs[self.match_key] =
agent_action['inform_slots'][self.match_key]
        agent_action.update({'round': self.round_num, 'speaker': 'Agent'})
        self.history.append(agent_action)

    def update_state_user(self, user_action):
        for key, value in user_action['inform_slots'].items():
            self.current_informs[key] = value
```

```
        user_action.update({'round': self.round_num, 'speaker': 'User'})
        self.history.append(user_action)
        self.round_num += 1

class DBQuery:
    def __init__(self, database):
        self.database = database
        self.cached_db_slot = defaultdict(dict)
        self.cached_db = defaultdict(dict)
        self.no_query = no_query_keys
        self.match_key = usersim_default_key

    def fill_inform_slot(self, inform_slot_to_fill, current_inform_slots):
        assert len(inform_slot_to_fill) == 1
        key = list(inform_slot_to_fill.keys())[0]
        current_informs = copy.deepcopy(current_inform_slots)
        current_informs.pop(key, None)
        db_results = self.get_db_results(current_informs)
        filled_inform = {}
        values_dict = self._count_slot_values(key, db_results)
        if values_dict:
            filled_inform[key] = max(values_dict, key=values_dict.get)
        else:
            filled_inform[key] = 'no match available'
        return filled_inform

    def _count_slot_values(self, key, db_subdict):
        slot_values = defaultdict(int)
        for id in db_subdict.keys():
            current_option_dict = db_subdict[id]
            if key in current_option_dict.keys():
                slot_value = current_option_dict[key]
                slot_values[slot_value] += 1
        return slot_values

    def get_db_results(self, constraints):
        new_constraints = {k: v for k, v in constraints.items() if k != self.no_query
and v != 'anything'}
        inform_items = frozenset(new_constraints.items())
        cache_return = self.cached_db[inform_items]
        if cache_return == None:
            return {}
        if cache_return:
            return cache_return
```

```
        available_options = {}
        for id in self.database.keys():
            current_option_dict = self.database[id]
            if len(set(new_constraints.keys()) - set(self.database[id].keys())) == 0:
                match = True
                for k, v in new_constraints.items():
                    if str(v) != str(current_option_dict[k]):
                        match = False
                if match:
                    self.cached_db[inform_items].update({id: current_option_dict})
                    available_options.update({id: current_option_dict})
        if not available_options:
            self.cached_db[inform_items] = None
        return available_options

    def get_db_results_for_slots(self, current_informs):
        inform_items = frozenset(current_informs.items())
        cache_return = self.cached_db_slot[inform_items]
        if cache_return:
            return cache_return
        db_results = {key: 0 for key in current_informs.keys()}
        db_results['matching_all_constraints'] = 0
        for id in self.database.keys():
            all_slots_match = True
            for CI_key, CI_value in current_informs.items():
                if CI_key in self.no_query:
                    continue
                if CI_value == 'anything':
                    db_results[CI_key] += 1
                    continue
                if CI_key in self.database[id].keys():
                    if CI_value == self.database[id][CI_key]:
                        db_results[CI_key] += 1
                    else:
                        all_slots_match = False
                else:
                    all_slots_match = False
            if all_slots_match:
                db_results['matching_all_constraints'] += 1
        self.cached_db_slot[inform_items].update(db_results)
        assert self.cached_db_slot[inform_items] == db_results
        return db_results

class User:
```

```
def __init__(self, constants):
    self.max_round = constants['run']['max_round_num']

def reset(self):
    return self._return_response()

def _return_response(self):
    response = {'intent': '', 'inform_slots': {}, 'request_slots': {}}
    while True:
        input_string = input('Response: ')
        chunks = input_string.split('/')
        intent_correct = True
        if chunks[0] not in usersim_intents:
            intent_correct = False
        response['intent'] = chunks[0]
        informs_correct = True
        if len(chunks[1]) > 0:
            informs_items_list = chunks[1].split(', ')
            for inf in informs_items_list:
                inf = inf.split(': ')
                if inf[0] not in all_slots:
                    informs_correct = False
                    break
                response['inform_slots'][inf[0]] = inf[1]
        requests_correct = True
        if len(chunks[2]) > 0:
            requests_key_list = chunks[2].split(', ')
            for req in requests_key_list:
                if req not in all_slots:
                    requests_correct = False
                    break
                response['request_slots'][req] = 'UNK'
        if intent_correct and informs_correct and requests_correct:
            break
    return response

def _return_success(self):
    success = -2
    while success not in (-1, 0, 1):
        success = int(input('Success?: '))
    return success

def step(self, agent_action):
    for value in agent_action['inform_slots'].values():
```

```
            assert value != 'UNK'
            assert value != 'PLACEHOLDER'
        for value in agent_action['request_slots'].values():
            assert value != 'PLACEHOLDER'
        print(f'Agent Action: {agent_action}')
        done = False
        user_response = {'intent': '', 'request_slots': {}, 'inform_slots': {}}
        if agent_action['round'] == self.max_round:
            success = FAIL
            user_response['intent'] = 'done'
        else:
            user_response = self._return_response()
            success = self._return_success()
        if success == FAIL or success == SUCCESS:
            done = True
        assert 'UNK' not in user_response['inform_slots'].values()
        assert 'PLACEHOLDER' not in user_response['request_slots'].values()
        reward = reward_function(success, self.max_round)
        return user_response, reward, done, True if success == 1 else False

class UserSimulator:
    def __init__(self, goal_list, constants, database):
        self.goal_list = goal_list
        self.max_round = constants['run']['max_round_num']
        self.default_key = usersim_default_key
        self.init_informs = usersim_required_init_inform_keys
        self.no_query = no_query_keys
        self.database = database

    def reset(self):
        self.goal = random.choice(self.goal_list)
        self.goal['request_slots'][self.default_key] = 'UNK'
        self.state = {}
        self.state['history_slots'] = {}
        self.state['inform_slots'] = {}
        self.state['request_slots'] = {}
        self.state['rest_slots'] = {}
        self.state['rest_slots'].update(self.goal['inform_slots'])
        self.state['rest_slots'].update(self.goal['request_slots'])
        self.state['intent'] = ''
        self.constraint_check = FAIL
        return self._return_init_action()

    def _return_init_action(self):
```

```
        self.state['intent'] = 'request'
        if self.goal['inform_slots']:
            for inform_key in self.init_informs:
                if inform_key in self.goal['inform_slots']:
                    self.state['inform_slots'][inform_key] =
self.goal['inform_slots'][inform_key]
                    self.state['rest_slots'].pop(inform_key)
                    self.state['history_slots'][inform_key] =
self.goal['inform_slots'][inform_key]
            if not self.state['inform_slots']:
                key, value = random.choice(list(self.goal['inform_slots'].items()))
                self.state['inform_slots'][key] = value
                self.state['rest_slots'].pop(key)
                self.state['history_slots'][key] = value
        self.goal['request_slots'].pop(self.default_key)
        if self.goal['request_slots']:
            req_key = random.choice(list(self.goal['request_slots'].keys()))
        else:
            req_key = self.default_key
        self.goal['request_slots'][self.default_key] = 'UNK'
        self.state['request_slots'][req_key] = 'UNK'
        user_response = {}
        user_response['intent'] = self.state['intent']
        user_response['request_slots'] = copy.deepcopy(self.state['request_slots'])
        user_response['inform_slots'] = copy.deepcopy(self.state['inform_slots'])
        return user_response

    def step(self, agent_action):
        for value in agent_action['inform_slots'].values():
            assert value != 'UNK'
            assert value != 'PLACEHOLDER'
        for value in agent_action['request_slots'].values():
            assert value != 'PLACEHOLDER'
        self.state['inform_slots'].clear()
        self.state['intent'] = ''
        done = False
        success = NO_OUTCOME
        if agent_action['round'] == self.max_round:
            done = True
            success = FAIL
            self.state['intent'] = 'done'
            self.state['request_slots'].clear()
        else:
            agent_intent = agent_action['intent']
```

```
            if agent_intent == 'request':
                self._response_to_request(agent_action)
            elif agent_intent == 'inform':
                self._response_to_inform(agent_action)
            elif agent_intent == 'match_found':
                self._response_to_match_found(agent_action)
            elif agent_intent == 'done':
                success = self._response_to_done()
                self.state['intent'] = 'done'
                self.state['request_slots'].clear()
                done = True
        if self.state['intent'] == 'request':
            assert self.state['request_slots']
        if self.state['intent'] == 'inform':
            assert self.state['inform_slots']
            assert not self.state['request_slots']
        assert 'UNK' not in self.state['inform_slots'].values()
        assert 'PLACEHOLDER' not in self.state['request_slots'].values()
        for key in self.state['rest_slots']:
            assert key not in self.state['history_slots']
        for key in self.state['history_slots']:
            assert key not in self.state['rest_slots']
        for inf_key in self.goal['inform_slots']:
            assert self.state['history_slots'].get(inf_key, False) or
self.state['rest_slots'].get(inf_key, False)
        for req_key in self.goal['request_slots']:
            assert self.state['history_slots'].get(req_key, False) or
self.state['rest_slots'].get(req_key,False), req_key
        for key in self.state['rest_slots']:
            assert self.goal['inform_slots'].get(key, False) or
self.goal['request_slots'].get(key, False)
        assert self.state['intent'] != ''
        user_response = {}
        user_response['intent'] = self.state['intent']
        user_response['request_slots'] = copy.deepcopy(self.state['request_slots'])
        user_response['inform_slots'] = copy.deepcopy(self.state['inform_slots'])
        reward = reward_function(success, self.max_round)
        return user_response, reward, done, True if success == 1 else False

    def _response_to_request(self, agent_action):
        agent_request_key = list(agent_action['request_slots'].keys())[0]
        if agent_request_key in self.goal['inform_slots']:
            self.state['intent'] = 'inform'
```

```
            self.state['inform_slots'][agent_request_key] =
self.goal['inform_slots'][agent_request_key]
            self.state['request_slots'].clear()
            self.state['rest_slots'].pop(agent_request_key, None)
            self.state['history_slots'][agent_request_key] =
self.goal['inform_slots'][agent_request_key]
        elif agent_request_key in self.goal['request_slots'] and agent_request_key in
self.state['history_slots']:
            self.state['intent'] = 'inform'
            self.state['inform_slots'][agent_request_key] =
self.state['history_slots'][agent_request_key]
            self.state['request_slots'].clear()
            assert agent_request_key not in self.state['rest_slots']
        elif agent_request_key in self.goal['request_slots'] and agent_request_key in
self.state['rest_slots']:
            self.state['request_slots'].clear()
            self.state['intent'] = 'request'
            self.state['request_slots'][agent_request_key] = 'UNK'
            rest_informs = {}
            for key, value in list(self.state['rest_slots'].items()):
                if value != 'UNK':
                    rest_informs[key] = value
            if rest_informs:
                key_choice, value_choice = random.choice(list(rest_informs.items()))
                self.state['inform_slots'][key_choice] = value_choice
                self.state['rest_slots'].pop(key_choice)
                self.state['history_slots'][key_choice] = value_choice
        else:
            assert agent_request_key not in self.state['rest_slots']
            self.state['intent'] = 'inform'
            self.state['inform_slots'][agent_request_key] = 'anything'
            self.state['request_slots'].clear()
            self.state['history_slots'][agent_request_key] = 'anything'

    def _response_to_inform(self, agent_action):
        agent_inform_key = list(agent_action['inform_slots'].keys())[0]
        agent_inform_value = agent_action['inform_slots'][agent_inform_key]
        assert agent_inform_key != self.default_key
        self.state['history_slots'][agent_inform_key] = agent_inform_value
        self.state['rest_slots'].pop(agent_inform_key, None)
        self.state['request_slots'].pop(agent_inform_key, None)
        if agent_inform_value != self.goal['inform_slots'].get(agent_inform_key,
agent_inform_value):
            self.state['intent'] = 'inform'
```

```
            self.state['inform_slots'][agent_inform_key] =
self.goal['inform_slots'][agent_inform_key]
            self.state['request_slots'].clear()
            self.state['history_slots'][agent_inform_key] =
self.goal['inform_slots'][agent_inform_key]
        else:
            if self.state['request_slots']:
                self.state['intent'] = 'request'
            elif self.state['rest_slots']:
                def_in = self.state['rest_slots'].pop(self.default_key, False)
                if self.state['rest_slots']:
                    key, value = random.choice(list(self.state['rest_slots'].items()))
                    if value != 'UNK':
                        self.state['intent'] = 'inform'
                        self.state['inform_slots'][key] = value
                        self.state['rest_slots'].pop(key)
                        self.state['history_slots'][key] = value
                    else:
                        self.state['intent'] = 'request'
                        self.state['request_slots'][key] = 'UNK'
                else:
                    self.state['intent'] = 'request'
                    self.state['request_slots'][self.default_key] = 'UNK'
                if def_in == 'UNK':
                    self.state['rest_slots'][self.default_key] = 'UNK'
            else:
                self.state['intent'] = 'thanks'

    def _response_to_match_found(self, agent_action):
        agent_informs = agent_action['inform_slots']
        self.state['intent'] = 'thanks'
        self.constraint_check = SUCCESS
        assert self.default_key in agent_informs
        self.state['rest_slots'].pop(self.default_key, None)
        self.state['history_slots'][self.default_key] =
str(agent_informs[self.default_key])
        self.state['request_slots'].pop(self.default_key, None)
        if agent_informs[self.default_key] == 'no match available':
            self.constraint_check = FAIL
        for key, value in self.goal['inform_slots'].items():
            assert value != None
            if key in self.no_query:
                continue
            if value != agent_informs.get(key, None):
```

```
                self.constraint_check = FAIL
                break
        if self.constraint_check == FAIL:
            self.state['intent'] = 'reject'
            self.state['request_slots'].clear()

    def _response_to_done(self):
        if self.constraint_check == FAIL:
            return FAIL
        if not self.state['rest_slots']:
            assert not self.state['request_slots']
        if self.state['rest_slots']:
            return FAIL
        assert self.state['history_slots'][self.default_key] != 'no match available'
        match =
copy.deepcopy(self.database[int(self.state['history_slots'][self.default_key])])
        for key, value in self.goal['inform_slots'].items():
            assert value != None
            if key in self.no_query:
                continue
            if value != match.get(key, None):
                assert True == False, f'match: {match}\ngoal: {self.goal}'
                break
        return SUCCESS

class ErrorModelController:
    def __init__(self, db_dict, constants):
        self.movie_dict = db_dict
        self.slot_error_prob = constants['emc']['slot_error_prob']
        self.slot_error_mode = constants['emc']['slot_error_mode']  # [0, 3]
        self.intent_error_prob = constants['emc']['intent_error_prob']
        self.intents = usersim_intents

    def infuse_error(self, frame):
        informs_dict = frame['inform_slots']
        for key in list(frame['inform_slots'].keys()):
            assert key in self.movie_dict
            if random.random() < self.slot_error_prob:
                if self.slot_error_mode == 0:
                    self._slot_value_noise(key, informs_dict)
                elif self.slot_error_mode == 1:
                    self._slot_noise(key, informs_dict)
                elif self.slot_error_mode == 2:
                    self._slot_remove(key, informs_dict)
```

```
                else:
                    rand_choice = random.random()
                    if rand_choice <= 0.33:
                        self._slot_value_noise(key, informs_dict)
                    elif rand_choice > 0.33 and rand_choice <= 0.66:
                        self._slot_noise(key, informs_dict)
                    else:
                        self._slot_remove(key, informs_dict)
        if random.random() < self.intent_error_prob:
            frame['intent'] = random.choice(self.intents)

    def _slot_value_noise(self, key, informs_dict):
        informs_dict[key] = random.choice(self.movie_dict[key])

    def _slot_noise(self, key, informs_dict):
        informs_dict.pop(key)
        random_slot = random.choice(list(self.movie_dict.keys()))
        informs_dict[random_slot] = random.choice(self.movie_dict[random_slot])

    def _slot_remove(self, key, informs_dict):
        informs_dict.pop(key)

class User:
    def __init__(self, constants):
        self.max_round = constants['run']['max_round_num']

    def reset(self):
        return self._return_response()

    def _return_response(self):
        response = {'intent': '', 'inform_slots': {}, 'request_slots': {}}
        while True:
            input_string = input('Response: ')
            chunks = input_string.split('/')

            intent_correct = True
            if chunks[0] not in usersim_intents:
                intent_correct = False
            response['intent'] = chunks[0]
            informs_correct = True
            if len(chunks[1]) > 0:
                informs_items_list = chunks[1].split(', ')
                for inf in informs_items_list:
                    inf = inf.split(': ')
```

```
                if inf[0] not in all_slots:
                    informs_correct = False
                    break
                response['inform_slots'][inf[0]] = inf[1]
            requests_correct = True
            if len(chunks[2]) > 0:
                requests_key_list = chunks[2].split(', ')
                for req in requests_key_list:
                    if req not in all_slots:
                        requests_correct = False
                        break
                    response['request_slots'][req] = 'UNK'
            if intent_correct and informs_correct and requests_correct:
                break
        return response

    def _return_success(self):
        success = -2
        while success not in (-1, 0, 1):
            success = int(input('Success?: '))
        return success

    def step(self, agent_action):
        for value in agent_action['inform_slots'].values():
            assert value != 'UNK'
            assert value != 'PLACEHOLDER'
        for value in agent_action['request_slots'].values():
            assert value != 'PLACEHOLDER'
        print(f'Действие агента: {agent_action}')
        done = False
        user_response = {'intent': '', 'request_slots': {}, 'inform_slots': {}}
        if agent_action['round'] == self.max_round:
            success = FAIL
            user_response['intent'] = 'done'
        else:
            user_response = self._return_response()
            success = self._return_success()
        if success == FAIL or success == SUCCESS:
            done = True
        assert 'UNK' not in user_response['inform_slots'].values()
        assert 'PLACEHOLDER' not in user_response['request_slots'].values()
        reward = reward_function(success, self.max_round)
        return user_response, reward, done, True if success == 1 else False
```

```
def convert_list_to_dict(lst):
    if len(lst) > len(set(lst)):
        raise ValueError('List must be unique!')
    return {k: v for v, k in enumerate(lst)}

def reward_function(success, max_round):
    reward = -1
    if success == FAIL:
        reward += -max_round
    elif success == SUCCESS:
        reward += 2 * max_round
    return reward

# Тренировка чат-бота
CONSTANTS_FILE_PATH = "C:/Users/nkmeo/Course
work/Notebooks/constants.json"
with open(CONSTANTS_FILE_PATH, "r") as read_file:
    constants = json.load(read_file)
file_path_dict = constants['db_file_paths']
DATABASE_FILE_PATH = file_path_dict['database']
DICT_FILE_PATH = file_path_dict['dict']
USER_GOALS_FILE_PATH = file_path_dict['user_goals']
run_dict = constants['run']
USE_USERSIM = run_dict['usersim']
WARMUP_MEM = run_dict['warmup_mem']
NUM_EP_TRAIN = run_dict['num_ep_run']
TRAIN_FREQ = run_dict['train_freq']
MAX_ROUND_NUM = run_dict['max_round_num']
SUCCESS_RATE_THRESHOLD = run_dict['success_rate_threshold']
database = pickle.load(open(DATABASE_FILE_PATH, 'rb'), encoding='latin1')
remove_empty_slots(database)
db_dict = pickle.load(open(DICT_FILE_PATH, 'rb'), encoding='latin1')
user_goals = pickle.load(open(USER_GOALS_FILE_PATH, 'rb'),
encoding='latin1')
if USE_USERSIM:
    user = UserSimulator(user_goals, constants, database)
else:
    user = User(constants)
emc = ErrorModelController(db_dict, constants)
state_tracker = StateTracker(database, constants)
dqn_agent = DQNAgent(state_tracker.get_state_size(), constants)

def run_round(state, warmup=False):
```

```
    agent_action_index, agent_action = dqn_agent.get_action(state,
use_rule=warmup)
    state_tracker.update_state_agent(agent_action)
    user_action, reward, done, success = user.step(agent_action)
    if not done:
        emc.infuse_error(user_action)
    state_tracker.update_state_user(user_action)
    next_state = state_tracker.get_state(done)
    dqn_agent.add_experience(state, agent_action_index, reward, next_state, done)
    return next_state, reward, done, success


def warmup_run():
    print('Тренировка началась...')
    total_step = 0
    start = time.time()
    while total_step != WARMUP_MEM and not dqn_agent.is_memory_full():
        # Reset episode
        episode_reset()
        done = False
        # Get initial state from state tracker
        state = state_tracker.get_state()
        while not done:
            next_state, _, done, _ = run_round(state, warmup=True)
            total_step += 1
            state = next_state

    print(f'...Тренировка закончилась {time.time()-start}')


def train_run():
    print('Тренировка началась...')
    episode = 0
    period_reward_total = 0
    period_success_total = 0
    success_rate_best = 0.0
    success_rate_by_period = {}
    success_rate_best_period = {}
    period_reward_total_period = {}
    while episode < NUM_EP_TRAIN:
        start = time.time()
        episode_reset()
        episode += 1
        done = False
```

```
        state = state_tracker.get_state()
        while not done:
            next_state, reward, done, success = run_round(state)
            period_reward_total += reward
            state = next_state
        period_success_total += success
        if episode % TRAIN_FREQ == 0:
            success_rate = period_success_total / TRAIN_FREQ
            avg_reward = period_reward_total / TRAIN_FREQ
            if success_rate >= success_rate_best and success_rate >=
SUCCESS_RATE_THRESHOLD:
                dqn_agent.empty_memory()
            if success_rate > success_rate_best:
                print(f':Эпизод: {episode} Новая лучшая вероятность успешного
завершения диалога: {success_rate} Средняя награда: {avg_reward}')
                success_rate_best = success_rate
                dqn_agent.save_weights()
            success_rate_by_period[episode] = success_rate
            success_rate_best_period[episode] = success_rate_best
            period_reward_total_period[episode] = avg_reward
            period_success_total = 0
            period_reward_total = 0
            dqn_agent.copy()
            dqn_agent.train()
    print('...Тренировка закончена')
    return(success_rate_by_period, success_rate_best_period,
period_reward_total_period)

def episode_reset():
    state_tracker.reset()
    user_action = user.reset()
    emc.infuse_error(user_action)
    state_tracker.update_state_user(user_action)
    dqn_agent.reset()

# Запуск для реального пользователя
def test_run():
    print('Тестирование началось..')
    episode = 0
    while episode < NUM_EP_TEST:
        episode_reset()
        episode += 1
        ep_reward = 0
        done = False
```

```
    state = state_tracker.get_state()
    while not done:
        agent_action_index, agent_action = dqn_agent.get_action(state)
        state_tracker.update_state_agent(agent_action)
        user_action, reward, done, success = user.step(agent_action)
        ep_reward += reward
        if not done:
            emc.infuse_error(user_action)
        state_tracker.update_state_user(user_action)
        state = state_tracker.get_state(done)
    print(f'Эпизод: {episode} Успех: {success} Награда: {ep_reward}')
print('...Тестирование закончено')
```