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 3](#_Toc134726419)

[ГЛАВА 1. ОБУЧЕНИЕ ДИАЛОГОВОЙ СИСТЕМЫ С ИСПОЛЬЗОВАНИЕМ ГЛУБОКОГО ОБУЧЕНИЯ С ПОДКРЕПЛЕНИЕМ. 4](#_Toc134726420)

[1.1 Машинное обучение с подкреплением. 4](#_Toc134726421)

[1.2 Метод Q-learning. 5](#_Toc134726422)

[1.3 Deep Q Networks (DQN). 7](#_Toc134726423)

[1.4 Double Q-learning и Double Deep Q-Network (DDQN). 9](#_Toc134726424)

[ГЛАВА 2. РЕАЛИЗАЦИЯ ДИАЛОГОВОЙ СИСТЕМЫ. 11](#_Toc134726425)

[2.1 Основные этапы разработки и обучения. 11](#_Toc134726426)

[2.2 Используемые данные. 13](#_Toc134726427)

[2.3 Агент на базе Deep Q-Network и обучение. 14](#_Toc134726428)

[2.4 Трекер состояния диалога. 16](#_Toc134726429)

[2.5 Симуляция пользователя. 17](#_Toc134726430)

[2.6 Контроль ошибок. 20](#_Toc134726431)

[2.7 Запуск диалоговой системы. 20](#_Toc134726432)

[ЗАКЛЮЧЕНИЕ 22](#_Toc134726433)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ. 23](#_Toc134726434)

[ПРИЛОЖЕНИЕ А. ОПИСАНИЕ ПРОЦЕССОРА. 24](#_Toc134726435)

# **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 Машинное обучение с подкреплением.**

Для правильного понимания терминологии необходимо дать четкие определения некоторых основных объектов машинного 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 машинного 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]](#footnote-1)).

Определим функцию такую, что для текущего состояния и действия 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 для таких оптимальных политик как .

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

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

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

В таком случае необходимо ввести коэффициент дисконтирования (), 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 определяет, насколько функция в состоянии 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):

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

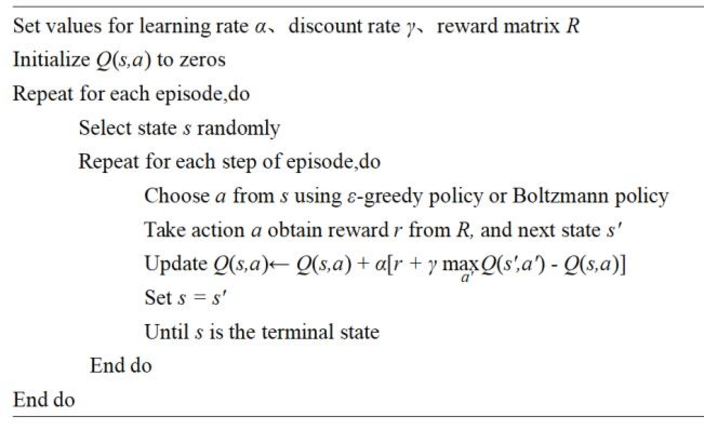


Рис. 1. Псевдокод, описывающий алгоритм Q-learning.

## **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]](#footnote-2) без каких-либо корректировок в архитектуре системы консоли[[3]](#footnote-3). Данная работа положила начало применению нейронных сетей для реализации глубокого машинного обучения с подкреплением.

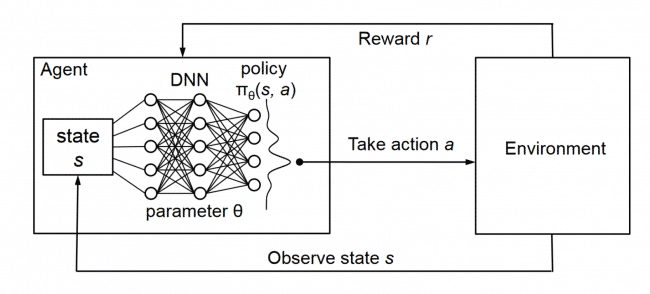


Рис. 2. Структура алгоритма глубокого обучения с подкреплением на основе Deep Q-network.

Deep Q-network (DQN) это многослойная нейронная сеть, которая для текущего состояния возвращает вектор действий , где – параметры нейронной сети. Для n-мерного пространства состояний и пространства действий, содержащего m действий, нейронная сеть является функцией от к .

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

Для воспроизведения опыта наблюдаемы 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 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 предотвратить это, необходимо отделить выбор от произведения оценки. В этом и заключается идея Double Q-learning[[5]](#footnote-5).

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

Изображение выглядит как текст, Шрифт, чек, линия

Автоматически созданное описание

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:

Следует обратить внимание, что выбор действия в операторе по-прежнему зависит от весов . Это означает, что мы производим выбор действия на основе «жадной политики» в соответствии с текущими значениями, определенными в . Однако также используется второй набор весов , 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 алгорим.
* Модуль генерации естественного языка (Natural Language Generator (NLG)). Обратная NLU задача генерации ответа на естественном языке для пользователя.

Изображение выглядит как диаграмма

Автоматически созданное описание

Рис. 1. 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.**

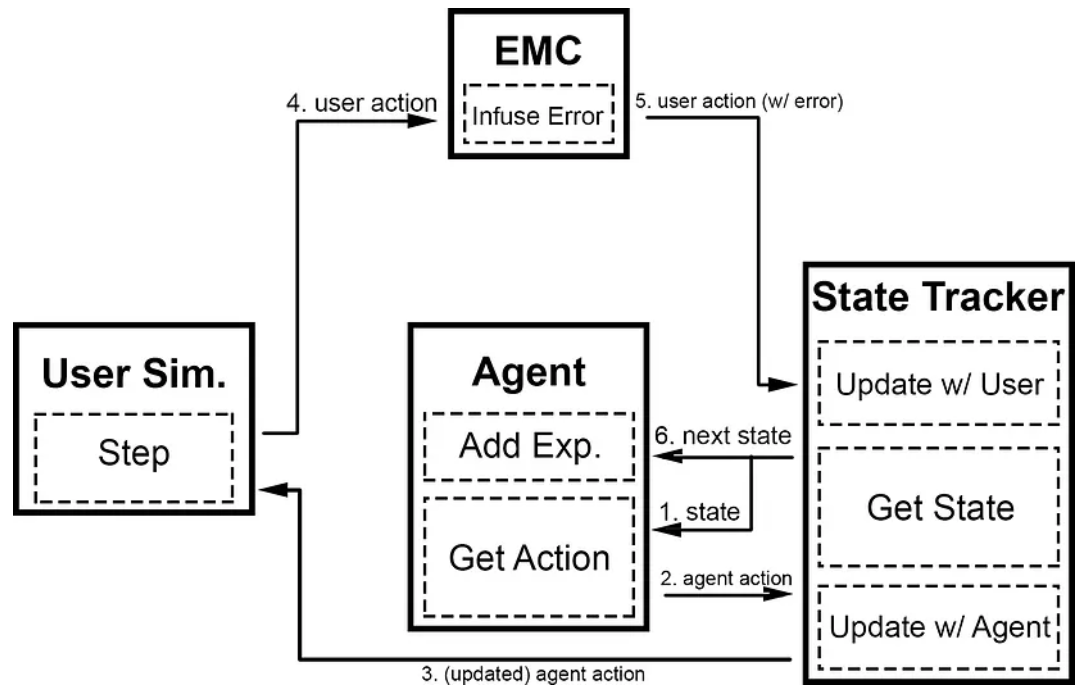


Рис. 2. Тренировочный цикл

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» - «Keros». 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. С этой целью выступает 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]](#footnote-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:

* «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 , 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 20000 проведенных диалогов, достигла значения 0,83 для модели DQN.

# **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. – 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. – 2019 г.
10. Christopher Watkins: «Learning from delayed rewards». – 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 2.**

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')

1. Watkins Ch. - Learning from delayed rewards, PhD. thesis, Cambridge University, 1989. [↑](#footnote-ref-1)
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». [↑](#footnote-ref-2)
3. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. – «Playing Atari with Deep Reinforcement Learning», 2013 г. [↑](#footnote-ref-3)
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. [↑](#footnote-ref-4)
5. Hado van Hasselt, Arthur Guez, David Silver. «Deep Reinforcement Learning with Double Q-learning», 2015 г. [↑](#footnote-ref-5)
6. Xiujun Li, Yun-Nung Chen, Lihong Li, Jianfeng Gao, Asli Celikyilmaz: «End-to-End Task-Completion Neural Dialogue Systems». – 2018 г. [↑](#footnote-ref-6)