Детальный разбор кода Telegram бота для игры в угадывание описаний изображений

Анализ кода

17 февраля 2026 г.

Содержание

1 Введение

Данный документ содержит подробный построчный анализ кода Telegram бота, который получает от пользователя фотографию, генерирует её описание с помощью нейросети BLIP, создаёт 4 неверных варианта описания из базы и предлагает пользователю угадать правильный вариант.

2 Импорт библиотек и установка зависимостей

```
!pip install -q python-telegram-bot transformers torch Pillow nest_asyncio > /
    dev/null 2>&1
```

Листинг 1: Установка зависимостей и импорт

Строка 1: Установка необходимых Python пакетов в среде Google Colab или Jupyter Notebook. Флаг `-q` (quiet) подавляет вывод установки, а перенаправление `> /dev/null 2>&1` скрывает все сообщения об ошибках и стандартный вывод.

```
import nest_asyncio
```

Листинг 2: Импорт библиотек

Строка 2: Импорт библиотеки `nest_asyncio`, которая позволяет запускать асинхронные циклы событий в средах, где они уже запущены (например, в Jupyter notebooks).

```
nest_asyncio.apply()
```

Листинг 3: Применение nest_asyncio

Строка 3: Применение патча `nest_asyncio` для разрешения вложенных асинхронных циклов.

```
import logging
import torch
import random
import re
import asyncio
```

Листинг 4: Импорт стандартных библиотек

Строки 4-8:

- `logging` - для ведения логов работы приложения
- `torch` - библиотека машинного обучения PyTorch для работы с нейросетями
- `random` - генерация случайных чисел для выбора вариантов
- `re` - регулярные выражения (в данном коде не используется, но импортирован)
- `asyncio` - асинхронное программирование

```
from PIL import Image
from io import BytesIO
from telegram import Update, InlineKeyboardButton, InlineKeyboardMarkup
from telegram.ext import Application, CommandHandler, MessageHandler, filters,
    ContextTypes, CallbackQueryHandler
```

Листинг 5: Импорт PIL и telegram

Строки 9-11:

- `PIL.Image` - работа с изображениями
- `io.BytesIO` - работа с байтовыми потоками
- `telegram.Update` - объект обновления от Telegram
- `InlineKeyboardButton` и `InlineKeyboardMarkup` - создание интерактивных кнопок
- `Application` - основной класс приложения бота

- `CommandHandler` - обработчик команд
- `MessageHandler` - обработчик сообщений
- `filters` - фильтры для сообщений
- `ContextTypes` - типы контекста
- `CallbackQueryHandler` - обработчик нажатий на кнопки

```
from transformers import BlipProcessor, BlipForConditionalGeneration,
    GPT2LMHeadModel, GPT2Tokenizer
```

Листинг 6: Импорт трансформеров

Строка 12:

- `BlipProcessor` - процессор для модели BLIP (обработка изображений)
- `BlipForConditionalGeneration` - модель BLIP для генерации описаний
- `GPT2LMHeadModel` - модель GPT-2 для генерации текста
- `GPT2Tokenizer` - токенизатор для GPT-2

3 Инициализация переменных и моделей

```
a = "8484962267:AAFfAoLvBWdzWd8UJtW8aW0BLNoH0LmDHf8"
```

Листинг 7: Токен бота

Строка 14: Присвоение переменной `a` токена доступа к Telegram боту. Токен получен от BotFather при создании бота.

```
logging.basicConfig(level=logging.WARNING)
```

Листинг 8: Настройка логирования

Строка 16: Настройка базовой конфигурации логирования на уровень WARNING, что означает запись только предупреждений и ошибок.

```
b = BlipProcessor.from_pretrained("Salesforce/blip-image-captioning-base")
c = BlipForConditionalGeneration.from_pretrained("Salesforce/blip-image-
    captioning-base")
d = GPT2Tokenizer.from_pretrained("distilgpt2")
e = GPT2LMHeadModel.from_pretrained("distilgpt2")
```

Листинг 9: Загрузка моделей

Строки 18-21:

- `b` - загрузка процессора BLIP из предобученной модели Salesforce
- `c` - загрузка модели BLIP для генерации описаний
- `d` - загрузка токенизатора DistilGPT2 (облегчённая версия GPT-2)
- `e` - загрузка модели DistilGPT2

```
f = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Листинг 10: Определение устройства

Строка 23: Определение устройства для вычислений. Если доступна CUDA (GPU), используется GPU, иначе CPU. Результат сохраняется в переменную `f`.

```
c = c.to(f)
e = e.to(f)
```

Листинг 11: Перенос моделей на устройство

Строки 24-25: Перенос моделей BLIP и GPT-2 на выбранное устройство (GPU или CPU).

```
d.pad_token = d.eos_token
```

Листинг 12: Настройка токенизатора

Строка 27: Установка токена заполнения (padding token) равным токену конца последовательности (end of sequence token) для токенизатора GPT-2.

```
g = {}
```

Листинг 13: Хранилище данных пользователей

Строка 29: Инициализация пустого словаря `g` для хранения данных пользователей (правильные ответы и варианты).

4 База описаний

```
h = [
    "A large group of people enjoying dinner together at a busy restaurant with red chairs.",
    "The sun setting behind tall mountains casting long shadows across the valley below.",
    ...
]
```

Листинг 14: База описаний (фрагмент)

Строки 31-181: Большая база `h` из 150+ детальных описаний на английском языке, каждое примерно по 10 слов. Эти описания используются для генерации неверных вариантов ответа.

5 Обработчики команд

```
async def i(j: Update, k: ContextTypes.DEFAULT_TYPE):
    l = "                          .                    5
                    .                                    ."
    await j.message.reply_text(l)
```

Листинг 15: Команда /start

Строки 183-186:

- Асинхронная функция `i` - обработчик команды /start
- `j` - объект Update от Telegram
- `k` - контекст выполнения
- `l` - приветственное сообщение
- `await j.message.reply_text(l)` - отправка ответного сообщения

```
async def m(n: Update, o: ContextTypes.DEFAULT_TYPE):
    p = "                                              5
                                        ."
    await n.message.reply_text(p)
```

Листинг 16: Команда /help

Строки 188-191: Обработчик команды /help, отправляющий краткую инструкцию.

```
async def q(r: Update, s: ContextTypes.DEFAULT_TYPE):
    t = "         : 1              , 4                .                                ."
    await r.message.reply_text(t)
```

Листинг 17: Команда /game

Строки 193-196: Обработчик команды /game, объясняющий правила игры.

6 Функции генерации описаний

```
def u(v):
    w = Image.open(BytesIO(v))
    if w.mode != "RGB":
        w = w.convert("RGB")
    w.thumbnail((400, 400))
    x = b(images=w, return_tensors="pt").to(f)
    with torch.no_grad():
        y = c.generate(**x, max_length=30, num_beams=3)
    z = b.decode(y[0], skip_special_tokens=True)
    aa = z.split()
    if aa:
        aa[0] = aa[0].capitalize()
        z = ' '.join(aa)
        if not z.endswith('.'):
            z = z + '.'
    return z
```

Листинг 18: Генерация описания изображения

Строки 198-214: Функция генерации правильного описания

- `def u(v):` - функция принимает байты изображения `v`
- `w = Image.open(BytesIO(v))` - открывает изображение из байтового потока
- `if w.mode != "RGB": w = w.convert("RGB")` - конвертирует в RGB если нужно
- `w.thumbnail((400, 400))` - уменьшает изображение до 400x400 для скорости
- `x = b(images=w, return_tensors="pt").to(f)` - подготавливает тензоры для модели BLIP
- `with torch.no_grad():` - отключает вычисление градиентов для экономии памяти
- `y = c.generate(**x, max_length=30, num_beams=3)` - генерирует описание (beam search)
- `z = b.decode(y[0], skip_special_tokens=True)` - декодирует токены в текст
- `aa = z.split()` - разбивает текст на слова
- `if aa: aa[0] = aa[0].capitalize()` - делает первую букву заглавной
- `if not z.endswith('.'): z = z + '.'` - добавляет точку в конце если её нет
- `return z` - возвращает сгенерированное описание

```
def ac(ad, ae=4):
    af = []
    ag = len(ad.split())
    ah = []
    for ai in h:
        if ai.lower() != ad.lower():
            aj = len(ai.split())
            if abs(aj - ag) <= 7:
                ah.append(ai)

    random.shuffle(ah)

    if len(ah) >= ae:
        af = ah[:ae]
    else:
        af = ah.copy()
        al = [am for am in h if am not in af and am.lower() != ad.lower()]
        random.shuffle(al)
        if al:
```

```
            af.extend(al[:ae - len(af)])

    return af[:ae]
```

Листинг 19: Генерация неверных описаний

Строки 216-239: Функция генерации неверных описаний

- `def ac(ad, ae=4):` - принимает правильное описание `ad` и количество нужных фейков `ae`
- `af = []` - инициализация списка для фейков
- `ag = len(ad.split())` - подсчёт количества слов в правильном описании
- `ah = []` - временный список подходящих описаний
- `for ai in h:` - перебор всех описаний в базе
- `if ai.lower() != ad.lower():` - исключает совпадение с правильным
- `aj = len(ai.split())` - длина текущего описания
- `if abs(aj - ag) <= 7:` - выбирает описания с похожей длиной (разница 7 слов)
- `ah.append(ai)` - добавляет подходящее описание
- `random.shuffle(ah)` - перемешивает список
- Если подходящих достаточно - берёт первые `ae`
- Если нет - добирает из оставшихся
- `return af[:ae]` - возвращает нужное количество фейков

7 Создание вариантов ответа

```
def an(ao, ap, aq):
    ar = ac(ao, 4)
    as_all = [ao] + ar

    correct_index = random.randint(0, 4)
    if as_all[0] == ao:
        as_all[0], as_all[correct_index] = as_all[correct_index], as_all[0]
    else:
        for i, val in enumerate(as_all):
            if val == ao:
                as_all[i], as_all[correct_index] = as_all[correct_index],
    as_all[i]
                break

    if ap not in g:
        g[ap] = {}
    g[ap][aq] = {
        'at': ao,
        'au': as_all
    }
    av = []
    for aw in range(len(as_all)):
        av.append([InlineKeyboardButton(
            f"                {aw+1}",
            callback_data=f"ans_{ap}_{aq}_{aw}"
        )])
    ax = "\n".join([f"{ay+1}. {az}" for ay, az in enumerate(as_all)])
    return InlineKeyboardMarkup(av), as_all, ax
```

Листинг 20: Создание вариантов и кнопок

Строки 241-265: Функция создания вариантов

- `def an(ao, ap, aq):` - принимает правильное описание, ID пользователя, ID сообщения
- `ar = ac(ao, 4)` - получает 4 неверных описания
- `as_all = [ao] + ar` - создаёт список из 5 описаний (1 правда + 4 фейка)
- `correct_index = random.randint(0, 4)` - выбирает случайную позицию для правды
- Блок `if/else` гарантированно перемещает правильный ответ на случайную позицию
- `if ap not in g: g[ap] = {}` - создаёт запись для пользователя если её нет
- `g[ap][aq] = {'at': ao, 'au': as_all}` - сохраняет правильный ответ и все варианты
- `av = []` - инициализация списка для кнопок
- Цикл `for aw in range(len(as_all)):` создаёт кнопки для каждого варианта
- Каждая кнопка содержит callback_data с ID пользователя, ID сообщения и индексом варианта
- `ax` - форматирует текст с вариантами для отображения
- Возвращает разметку кнопок, список вариантов и отформатированный текст

8 Обработчик фото

```
async def ba(bb: Update, bc: ContextTypes.DEFAULT_TYPE):
    bd = await bb.message.photo[-2].get_file() if len(bb.message.photo) > 1
    else await bb.message.photo[-1].get_file()
    be = await bd.download_as_bytearray()
    bf = u(be)
    bg = bb.effective_user.id
    bh = await bb.message.reply_text("                    ...")
    bi = bh.message_id
    bj, bk, bl = an(bf, bg, bi)
    bm = f"                              :\n\n{bl}"
    await bh.edit_text(bm, reply_markup=bj)
```

Листинг 21: Обработка полученных фотографий

Строки 267-276: Асинхронная функция обработки фото

- `async def ba(bb: Update, bc: ContextTypes.DEFAULT`$_{TYPE}$`) : –`
- `bd = await bb.message.photo[-2].get_file()... - получает файл фото:`
  - `Telegram отправляет несколько версий фото (разных размеров)`
  - `Если есть несколько, берёт предпоследнюю (средний размер)`
  - `Иначе берёт последнюю`
- `be = await bd.download_as_bytearray() - скачивает фото как массив байт`
- `bf = u(be) - генерирует правильное описание через функцию u`
- `bg = bb.effective_user.id - получает ID пользователя`
- `bh = await bb.message.reply_text("Обработка...") - отправляет временное сообщение`
- `bi = bh.message_id - сохраняет ID временного сообщения`
- `bj, bk, bl = an(bf, bg, bi) - создаёт варианты, кнопки и текст`
- `bm = f"Выбери вариант:{bl}" - формирует финальный текст`
- `await bh.edit_text(bm, reply_markup=bj) - редактирует временное сообщение, добавляя варианты и кнопки`

9 Обработчик нажатий на кнопки

```
async def bo(bp: Update, bq: ContextTypes.DEFAULT_TYPE):
    br = bp.callback_query
    await br.answer()
    bs = br.data.split('_')
    if len(bs) != 4:
        return
    bt = int(bs[1])
    bu = int(bs[2])
    bv = int(bs[3])
    if bt not in g or bu not in g[bt]:
        await br.edit_message_text("                                    .")
        return
    bw = g[bt][bu]
    bx = bw['at']
    by = bw['au']
    bz = by[bv]
    ca = (bz == bx)
    cb = []
    for cc, cd in enumerate(by):
        ce = "    " if cd == bx else ("    " if cc == bv and not ca else "  ")
        cb.append(f"{ce}{cc+1}. {cd}")
    cf = "\n".join(cb)
    if ca:
        cg = f"          .\n\n{cf}"
    else:
        cg = f"              .
                .\n\n{cf}"
    await br.edit_message_text(cg)
    del g[bt][bu]
```

Листинг 22: Обработка ответов пользователя

Строки 278-305: Обработка нажатий на кнопки

- async def bo(bp: Update, bq: ContextTypes.DEFAULT$_TYPE$) : $-callback$
- br = bp.callback_query - получает объект callback запроса
- await br.answer() - подтверждает получение callback (убирает "часики"на кнопке)
- bs = br.data.split('_') - разбирает callback_data (формат: ans_userid_msgid_index)
- if len(bs) != 4: return - проверяет корректность формата
- bt = int(bs[1]) - ID пользователя
- bu = int(bs[2]) - ID сообщения
- bv = int(bs[3]) - индекс выбранного варианта
- if bt not in g or bu not in g[bt]: - проверяет существование сессии
- bw = g[bt][bu] - получает данные сессии
- bx = bw['at'] - правильный ответ
- by = bw['au'] - все варианты
- bz = by[bv] - выбранный пользователем вариант
- ca = (bz == bx) - проверка правильности ответа
- cb = [] - список для формирования результата
- Цикл for cc, cd in enumerate(by): формирует строки результата:

- " для правильного ответа
- " для неверного выбора пользователя
- - для остальных вариантов

- cf = "\n".join(cb) - объединяет строки
- Формирует финальное сообщение с результатом
- await br.edit_message_text(cg) - обновляет сообщение с результатом
- del g[bt][bu] - удаляет данные сессии (очистка памяти)

10 Запуск бота

```
1 ci = Application.builder().token(a).build()
```

Листинг 23: Создание и запуск приложения

Строка 307: Создание приложения бота с использованием токена из переменной a.

```
1 ci.add_handler(CommandHandler("start", i))
2 ci.add_handler(CommandHandler("help", m))
3 ci.add_handler(CommandHandler("game", q))
4 ci.add_handler(MessageHandler(filters.PHOTO, ba))
5 ci.add_handler(CallbackQueryHandler(bo, pattern="^ans_"))
```

Листинг 24: Добавление обработчиков

Строки 308-312: Регистрация всех обработчиков:

- Команда /start → функция i
- Команда /help → функция m
- Команда /game → функция q
- Сообщения с фото → функция ba
- Callback запросы с pattern "ans_"→ функция bo

```
1 print("                        ")
2 print("                                                ")
3 ci.run_polling()
```

Листинг 25: Запуск бота

Строки 313-315:

- Вывод сообщения о запуске
- Запуск polling (бесконечный цикл проверки новых сообщений от Telegram)

11 Заключение

Бот представляет собой полноценное приложение, использующее современные нейросетевые модели для анализа изображений и создания интерактивной игры. Код оптимизирован для работы как на GPU, так и на CPU, и включает все необходимые компоненты: обработку команд, анализ изображений, генерацию вариантов и интерактивный интерфейс.