Министерство науки и высшего образования Российской Федерации

ФГБОУ ВО АЛТАЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Институт цифровых технологий, электроники и физики

Кафедра вычислительной техники и электроники (ВТиЭ)

АВТОМАТИЗАЦИЯ РЕШЕНИЯ CAPTCHA В ТЕКСТОВОМ ФОРМАТЕ

Выполнил студент 5.306М группы:
_____ А. В. Лаптев

Проверил: доц. каф. ВТиЭ
_____ А. В. Калачев
«___» _____ 2025 г.

Барнаул 2025 г.

РЕФЕРАТ

Полный объём работы составляет 29 страниц, включая 3 рисунка и 1 таблицу.

Данная работа посвящена разработке модели нейронной сети для распознавания символов на текстовых CAPTCHA для автоматизации решения текстовых CAPTCHA на web-ресурсах для облегчения автоматизации тестирования web-приложений с использованием средств автоматизации.

Ключевые слова: CAPTCHA, нейронная сеть, TensorFlow, Python, OCR, Tesseract, CRNN, Seq-to-Seq.

Отчёт оформлена с помощью системы компьютерной вёрстки TEX и его расширения XETEX из дистрибутива *TeX Live*.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

CAPTCHA давно является стандартным инструментом для защиты веб-ресурсов от спама, автоматизированных ботов и нежелательного извлечения данных. Большинство современных сайтов и веб-приложений используют данную технологию в различных сценариях, включая регистрацию пользователей, подтверждение действий на сайте и защиту от автоматизированных атак.

Несмотря на развитие технологий CAPTCHA, включая невидимые для пользователя решения, текстовая CAPTCHA и её различные модификации остаются широко распространёнными. В связи с этим автоматизация процесса решения таких CAPTCHA сохраняет актуальность.

Автоматизированное распознавание текстовых CAPTCHA позволяет значительно снизить необходимость ручного тестирования веб-приложений, что, в свою очередь, повышает скорость и эффективность тестирования. Кроме того, подобные методы могут использоваться для анализа надёжности внедрённых CAPTCHA, выявления их слабых мест и повышения безопасности веб-приложений, например, за счёт комбинирования нескольких методов защиты.

Цель работы – разработать, реализовать и протестировать программу для автоматизированного распознавания текстовых CAPTCHA.

Для достижения поставленной цели необходимо решить следующие задачи:

1. изучить принципы реализации текстовых CAPTCHA на основе открытых источников (допустимые символы, применяемые искажения);
2. выбрать архитектуру нейронной сети, наиболее подходящую для распознавания CAPTCHA;
3. подготовить датасет изображений CAPTCHA с учётом возможных искажений;
4. обучить нейронную сеть с достаточной точностью;
5. протестировать модель на тестовом наборе данных и оценить её эффективность.

1. СОВРЕМЕННАЯ РЕАЛИЗАЦИЯ ТЕКСТОВЫХ CAPTCHA

Современные текстовые CAPTCHA обычно состоят из букв и цифр. Зачастую используются символы латинского алфавита (как прописные, так и строчные) и цифры от 0 до 9. Но обычно реализации исключают символы, которые могут быть легко перепутаны, например, буквы «O» и цифру «0», буквы «I» и «l» и тому подобное. Рекомендуемый набор символов в генераторах на некоторых CRM платформах выглядит следующим образом: ABCDEFGHJKLMNPQRSTWXYZ23456789 [1].

Длина последовательности символов в CAPTCHA обычно составляет от 4 до 8 символов, что обеспечивает баланс между удобством для пользователя и безопасностью, однако конкретная длина может варьироваться в зависимости от требований системы безопасности.

Для усложнения автоматического распознавания текстовые CAPTCHA подвергаются различным искажениям:

1. геометрические искажения: символы могут быть искажены, повернуты или наклонены, что затрудняет их распознавание автоматическими системами [2];
2. перекрытие символов: символы могут быть расположены близко друг к другу или даже перекрываться, что усложняет их сегментацию и последующее распознавание [3];
3. добавление шума: на изображение могут быть добавлены различные шумы, такие как линии, точки или круги, чтобы затруднить распознавание символов;
4. сложные фоны: использование фонов с различными цветами или узорами, что делает выделение символов более сложным [4];
5. нелинейные искажения: применение нелинейных трансформаций к символам, что делает их форму менее предсказуемой для автоматических систем распознавания [5].

Эти методы направлены на повышение сложности автоматического распознавания CAPTCHA, сохраняя при этом относительную легкость распознавания для человека.

2. ПОДГОТОВКА ДАТАСЕТА С ИЗОБРАЖЕНИЯМИ И ВЫБОР МОДЕЛИ НЕЙРОННОЙ СЕТИ

Качество используемого датасета оказывает существенное влияние на итоговую точность работы модели. Для эффективного обучения необходимо, чтобы набор данных соответствовал следующим требованиям:

1. достаточное количество изображений для каждого символа, что обеспечивает статистическую устойчивость модели;
2. разнообразие данных, включающее:
   2.1. различные углы наклона символов;
   2.2. вариативность написания символов и их искажения;
   2.3. наличие побочных визуальных элементов, создающих препятствия для распознавания;
   2.4. использование различных шрифтов.
3. переменная длина последовательностей символов, что позволяет модели адаптироваться к разным формам CAPTCHA.

Включение указанных факторов способствует обучению модели на более широком спектре признаков, что, в свою очередь, повышает её способность к обобщению на ранее невидимых данных.

Поскольку в открытом доступе отсутствует достаточное количество данных для формирования сбалансированного датасета, было принято решение о генерации синтетических изображений с использованием специализированных библиотек. В качестве основного инструмента выбрана библиотека captcha на языке Python, обладающая необходимым функционалом для создания изображений CAPTCHA с заданными параметрами. Данная библиотека поддерживает генерацию изображений с пользовательскими шрифтами и различными эффектами искажений, что исключает необходимость привлечения дополнительных инструментов.

Исходный код генератора синтетических CAPTCHA представлен в приложении 6.1.

После создания изображений все они прошли этапы предобработки, направленные на улучшение качества данных и повышение эффективности обучения модели. Предобработка включала следующие этапы:

1. преобразование изображений в градации серого для уменьшения количества каналов и снижения вычислительной нагрузки;
2. бинаризация изображений с целью получения контрастного представления символов (белый текст на черном фоне);

3. удаление шумов и фона с использованием морфологических операций, в частности, дилатации.

Исходный код обработчика изображений представлен в приложении 6.2.

Примеры сгенерированных и предобработанных CAPTCHA приведены на рисунке ниже:

Рис. 2.1 Изображение сгенерированной CAPTCHA и результат обработки.

Для распознавания текста с переменной длиной последовательности в задачах CAPTCHA наиболее часто применяются следующие архитектуры нейронных сетей:

1. оптическое распознавание символов (OCR);
2. рекуррентные сверточные нейронные сети (CRNN);
3. архитектуры последовательного обучения (Seq-to-Seq).

С целью выбора наиболее эффективной модели были реализованы и протестированы все указанные подходы, после чего была выбрана архитектура, обеспечивающая наилучшую точность предсказаний.

Для обучения моделей был сформирован датасет из 100 000 изображений CAPTCHA, содержащих случайные последовательности символов длиной от 4 до 7. Такой объем данных позволяет добиться высокой обобщающей способности модели и снизить вероятность переобучения.

3. ОПТИЧЕСКОЕ РАСПОЗНАВАНИЕ СИМВОЛОВ (OCR TESSERACT)

Изначально предполагалась реализация модели с использованием OCR, поскольку такие системы изначально разрабатывались для задач оптического распознавания текста. В качестве конкретной модели был выбран Tesseract.

Tesseract является одной из наиболее популярных систем OCR с открытым исходным кодом. Tesseract поддерживает более 100 языков, включая сложные письменности [6]. В версии 4.0 в модель была интегрирована нейронная сеть на основе долговременной краткосрочной памяти (LSTM), что позволило существенно повысить точность распознавания, особенно при обработке сложных шрифтов и рукописного текста [7].

Для решения поставленной задачи предполагалось использовать предобученную модель Tesseract и дообучить её на специализированном датасете, содержащем изображения CAPTCHA с характерными искажениями. Однако в ходе экспериментов было установлено, что точность распознавания последовательностей символов целиком составляла 0%, а точность для отдельных символов оказалась крайне низкой. Это связано с тем, что архитектура Tesseract недостаточно устойчива к искажениям, характерным для CAPTCHA, таким как деформация символов, наложение шумов и изменение углов наклона [8].

Таким образом, было принято решение отказаться от использования Tesseract в пользу более адаптированных к данной задаче моделей, таких как сверточные рекуррентные нейронные сети (CRNN) или модели последовательного обучения (Seq-to-Seq), обладающие высокой устойчивостью к вариативности и искажениям, характерным для CAPTCHA.

4. РЕКУРРЕНТНЫЕ СВЕРТОЧНЫЕ НЕЙРОННЫЕ СЕТИ (CRNN)

Сверточно-рекуррентные нейронные сети (CRNN) представляют собой гибридную архитектуру, сочетающую в себе возможности сверточных нейронных сетей (CNN) и рекуррентных нейронных сетей (RNN). Данный подход используется в задачах, связанных с обработкой последовательных данных, таких как распознавание текста, речь и видео [9].

Основное преимущество CRNN заключается в способности CNN-части извлекать пространственные признаки из изображений, тогда как RNN-часть позволяет учитывать временные зависимости между последовательными фрагментами данных [10].

Разработанная модель CRNN для распознавания CAPTCHA включает в себя три ключевых блока:

1. сверточный блок (CNN): предназначен для выделения признаков из изображений CAPTCHA. Включает в себя три последовательных сверточных слоя, а также методы нормализации и уменьшения размерности признакового пространства;
2. рекуррентный блок (RNN): использует двунаправленные слои GRU, позволяющие модели учитывать зависимость между последовательными символами в CAPTCHA;
3. выходной слой: полносвязный, который выполняет классификацию каждого символа в последовательности.

В приложении 6.4 представлена реализация CRNN-модели на языке Python с использованием библиотеки TensorFlow/Keras:

В данной архитектуре применяются слои Dropout для регуляризации, также используется l2-регуляризация, BatchNormalization для ускорения обучения и повышения устойчивости модели, а также функция softmax для предсказания классов символов.

После обучения данной модели результаты оказались превосходящими показатели OCR, однако все же не достигли удовлетворительного уровня. В частности, точность распознавания всей последовательности символов не превышала 10%, тогда как точность классификации отдельных символов составляла около 70%.

5. АРХИТЕКТУРА ПОСЛЕДОВАТЕЛЬНОГО ОБУЧЕНИЯ (SEQ-TO-SEQ)

Модели последовательного преобразования (Seq-to-Seq) широко применяются для задач, связанных с обработкой последовательностей переменной длины. Они используются в таких областях, как машинный перевод, распознавание речи и анализ текстов [11]. Данные модели основаны на архитектуре энкодера-декодера, где первый модуль кодирует входную последовательность в скрытое представление, а второй декодирует его в выходную последовательность.

Одним из ключевых элементов Seq2Seq является механизм внимания, который позволяет декодеру динамически фокусироваться на различных частях входной последовательности при генерации выходных символов [12]. Этот подход особенно полезен для распознавания CAPTCHA, так как символы в изображениях могут иметь разную ориентацию и степень искажения.

Кодировщик, в данной модели принимает входное изображение CAPTCHA и преобразует его в компактное представление. Архитектура кодировщика включает:

1. четыре сверточных блока, слои пакетной нормализации и слои подвыборки для понижения размерности входных данных;
2. глобальный усредненный слой для получения векторного представления изображения;
3. полносвязный слой для финального представления скрытого состояния;
4. рекуррентный слой для кодирования последовательности, возвращающий последнее скрытое состояние кодировщика.

Декодировщик выполняет пошаговую генерацию выходной последовательности, используя скрытое состояние кодировщика. В архитектуру декодировщика входят:

1. входной слой для последовательности токенов;
2. слой вложения, который преобразует входные токены в векторные представления;
3. рекуррентный слой, обрабатывающий последовательность с учетом скрытого состояния кодировщика;
4. механизм внимания, который позволяет декодеру учитывать релевантные части входного изображения;

5. полносвязный слой с функцией активации softmax для предсказания вероятностей символов.

Полная архитектура модели реализована в TensorFlow/Keras и реализация модели приведена в приложении 6.5.

На начальных этапах экспериментов предложенная Seq-to-Seq-модель показала наилучшие результаты среди рассмотренных вариантов. В отличие от OCR- и CRNN-моделей, данная архитектура смогла достичь более высокой точности распознавания последовательностей символов, что обусловлено применением механизма внимания. Дальнейшая работа с моделью была сосредоточена на её оптимизации и улучшении параметров обучения.

6. ТЕСТИРОВАНИЕ ВЫБРАННОЙ МОДЕЛИ НЕЙРОННОЙ СЕТИ

Как было установлено в предыдущих разделах, модель последовательного преобразования (Seq-to-Seq) продемонстрировала наилучшие результаты среди рассмотренных архитектур. Следующим этапом работы являлась оптимизация параметров модели, включая веса и коэффициенты регуляризации, с целью ускорения сходимости, минимизации риска переобучения и повышения точности распознавания целевых последовательностей.

Для проведения экспериментов исходный набор данных, содержащий 100 000 изображений, был случайным образом перемешан и разделён на три подмножества: обучающее, тестовое и валидационное в соотношении 80:10:10. Обучающая выборка использовалась непосредственно для обучения модели, валидационная — для контроля качества процесса обучения на каждой эпохе, а тестовая — для окончательной оценки модели на данных, с которыми она ранее не сталкивалась. В качестве основных метрик качества модели использовались функция потерь (loss) и точность (accuracy), рассчитываемая для каждого символа последовательности.

В процессе многократного обучения были экспериментально определены оптимальное количество эпох и значения гиперпараметров, обеспечивающие эффективное снижение функции потерь до приемлемых значений. График сходимости функции потерь представлен на рис. 6.1.

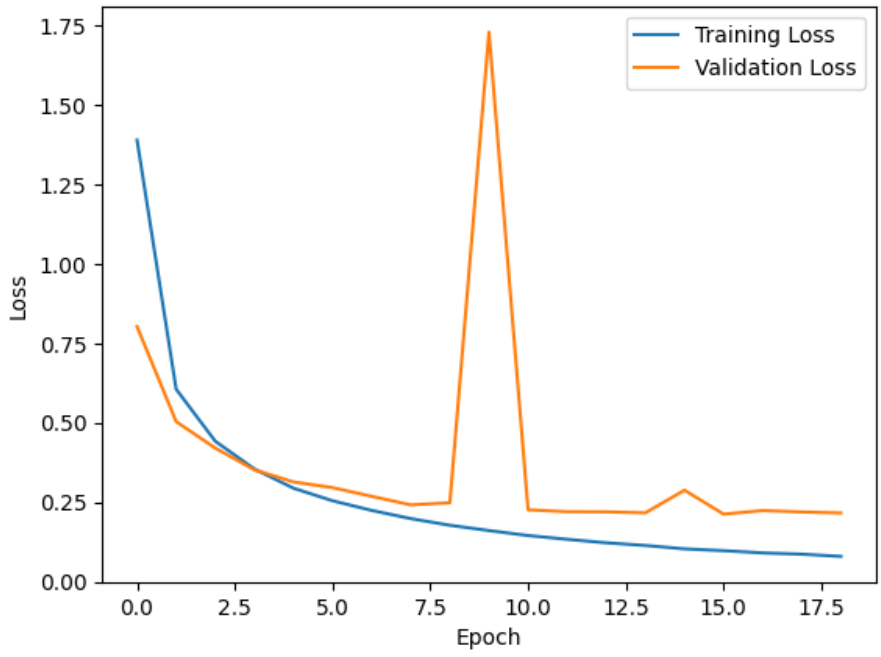

Рис. 6.1 График изменения значений функции потерь в процессе обучения.

Для предотвращения переобучения использовался механизм ранней остановки, согласно которому обучение прекращалось при отсутствии уменьшения значения функции потерь на валидационной выборке в течение трёх последовательных эпох. В данном эксперименте обучение завершилось на 18-й эпохе. На графике видно, что функция потерь стабилизировалась после 10 эпохе, поэтому 10 эпоха является балансом между точностью распознавания последовательностей и скоростью обучения модели.

Анализ графика сходимости функции потерь показывает наличие резкого увеличения её значения на 9-й эпохе, что может быть обусловлено следующими факторами:

1. перемешивание данных перед каждой эпохой могло привести к образованию несбалансированной выборки, содержащей значительное число сложных примеров.
2. динамическое изменение скорости обучения, осуществляемое с помощью механизма регулирования скорости обучения (learning rate scheduler), могло повлиять на изменение функции потерь.

Окончательная точность распознавания отдельных символов составила 0.9263.

После подбора оптимальных значений гиперпараметров модель была сохранена и протестирована на валидационной выборке. Точность распознавания последовательностей различной длины представлена в таблице 6.1.

Таблица 6.1 Точность предсказаний для последовательностей различной длины.

Длина последовательности	Точность распознавания
4 символа	0.9305
5 символов	0.7450
6 символов	0.4575
7 символов	0.1915

Также была построена матрица ошибок, позволяющая проанализировать частоту и характер ошибок модели при классификации различных классов. Данная матрица приведена на рис. 6.2.

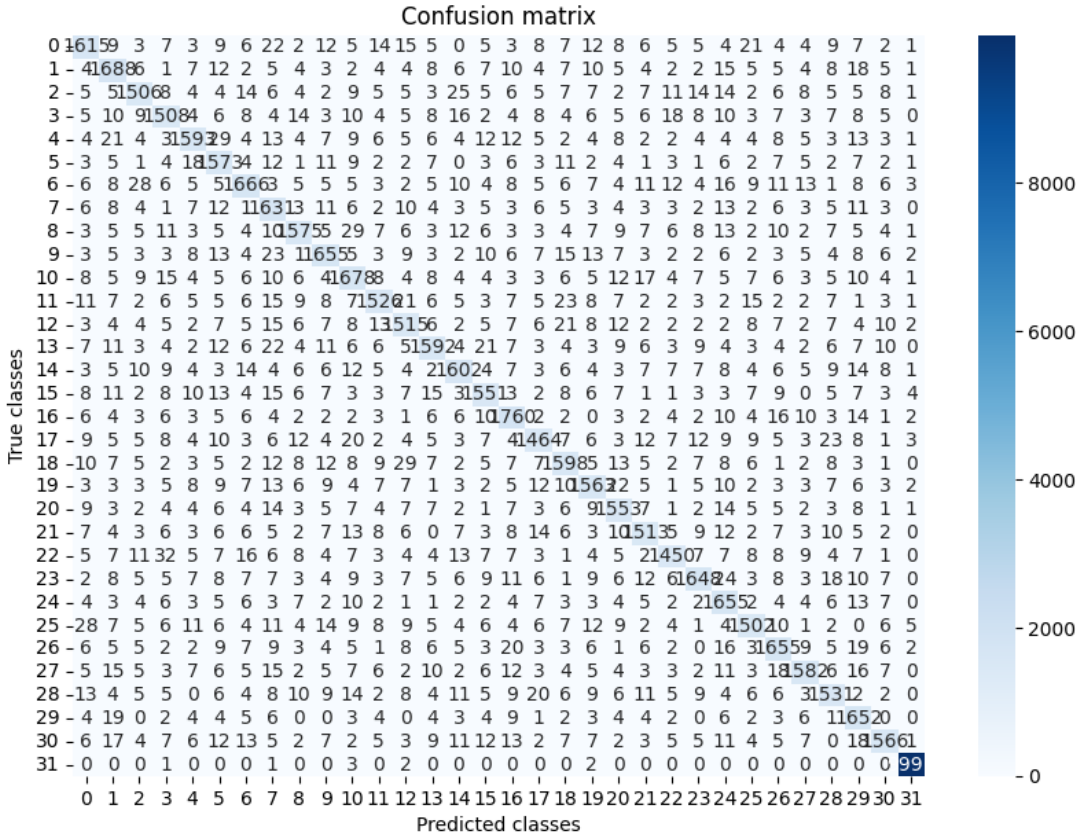

Рис. 6.2 Матрица ошибок для обученной модели.

Анализ полученных результатов показывает, что точность распознавания последовательностей значительной длины остаётся относительно низкой. Это можно объяснить высокой зависимостью модели Seq-to-Seq от объёма обучающих данных: для эффективного обобщения признаков, извлекаемых из изображений, требуется значительное количество примеров. Следовательно, увеличение размера обучающего набора данных потенциально может способствовать повышению точности модели, однако это также накладывает дополнительные требования к вычислительным ресурсам, необходимым для её обучения.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были решены задачи:

1. определен набор допустимых символов и искажений для создания датасета с текстовыми CAPTCHA;
2. выбрана модель нейронной сети Seq-to-Seq для распознавания текста с CAPTCHA;
3. подготовлен датасет из 100 000 изображений для обучения и тестирования модели нейронной сети;
4. обучена модель на изображениях с различной длиной последовательностей символов;
5. протестирована работа модели на тестовых изображениях.

В результате выполнения работы была создана модель, для распознавания текстовых CAPTCHA различной длины последовательности символов.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. CAPTCHA в 1С Битрикс [Электронный ресурс]. - URL: https://dev.1c-bitrix.ru/user_help/settings/settings/captcha.php.
2. Что такое капчи и как они работают [Электронный ресурс], - URL: https://ru-brightdata.com/blog/web-data-ru/what-is-a-captcha.
3. «Ломай меня полностью!» Как одни алгоритмы генерируют капчу, а другие её взламывают [Электронный ресурс]. - URL: https://proglib.io/p/lomay-menya-polnostyu-kak-odni-algoritmy-generiruyut-kapchu-a-drugie-ee-vzlamyvayut-2020-03-05.
4. *Рогов А. В.* CAPTCHA в эпоху цифровых угроз: эволюция, уязвимости и решения // Научные высказывания. — №21(68). — 2024. — С. 18—22.
5. Как настроить капчу [Электронный ресурс]. - URL: https://support.simai.ru/learn/courses/course/3/lesson/309/.
6. Tesseract OCR: How it works and when to use it [Электронный ресурс]. - URL: https://www.klippa.com/en/blog/information/tesseract-ocr/.
7. GitHub репозиторий проекта Tesseract [Электронный ресурс]. - URL: https://github.com/tesseract-ocr/tesseract.
8. Training Tesseract OCR with custom data [Электронный ресурс]. - URL: https://saiashish90.medium.com/training-tesseract-ocr-with-custom-data-d3f4881575c0.
9. Снова о распознавании рукописного текста, на этот раз с помощью CRNN [Электронный ресурс]. - URL: https://habr.com/ru/articles/720614/.
10. *Безвиконный Н. В. Гуськов А. А.* Программная эмуляция оптомагнитной нейронной сети для анализа рукописного текста // Мир науки. — 2024. — С. 1—63.
11. Модели «последовательность-последовательность» [Электронный ресурс]. - URL: https://www.ultralytics.com/ru/glossary/sequence-to-sequence-models.
12. *Морковников Н. М. Кипяткова И. С.* Исследование методов построения моделей кодер-декодер для распознавания русской речи // Информационно-управляющие системы. — №4. — 2019. — С. 1—9.

ПРИЛОЖЕНИЕ

Листинг 6.1 Исходный код генератора синтетических CAPTCHA

```python
from captcha.image import ImageCaptcha

from random import randint, shuffle
import numpy as np
import os

from textcaptcha.preprocessing_image import preprocessing_image


def generate_image(path_to_file: str, alphabet: list, number_of_start:int, number_of_captcha: int, size_of_image: tuple) -> list:
    # Генерация текстовых captcha
    text = ImageCaptcha(size_of_image[0], size_of_image[1], ['./fonts/arial.ttf', './fonts/comic.ttf', './fonts/cour.ttf', './fonts/georgia.ttf'])
    # Структура возвращаемого списка: [filename, label, (width, height)]
    filenames = []
    for _ in range(number_of_start, number_of_captcha):
        captcha_text = [alphabet[randint(0, len(alphabet) - 1)] for _ in range(randint(4, 7))]
        shuffle(captcha_text)
        text.write(''.join(captcha_text), f'{path_to_file}/{””.join(captcha_text)}.png')
        filenames.append(
            [f'{path_to_file}/{””.join(captcha_text)}.png',
            ''.join(captcha_text)]
        )

    return filenames


if __name__ == '__main__':
    # Алфавит допустимых символов
    alphabet = 'ABCDEFGHJKLMNPQRSTWXYZ023456789'
    # Создание директории для хранения полноценных синтетических текстовых captcha
```

```
    path_to_dataset = '../datasets/captcha'
    if not os.path.isdir(path_to_dataset):
        os.mkdir(path_to_dataset)
    # Создаем датасет из нужного полноценных синтетических
     ↪  captcha длиной от 4 до 7 символов размером 250x60
    filenames = generate_image(path_to_dataset, list(alphabet),
     ↪  0, 100000, (250, 60))

    # Предобработка изображений
    preprocessing_image(filenames)

    # Для отладки без создания датасета с нуля
    numpy_data = np.array(filenames, dtype=object)
    np.save('data.npy', numpy_data)
```

Листинг 6.2 Исходный код для предобработки изображений датасета

```
import cv2
import numpy as np


def preprocessing_image(list_filenames: list):
    '''Функция для предобработки изображений или изображения для
     ↪  предсказания'''
    # Предобработка изображений с CAPTCHA
    for file in list_filenames:
        # Открытие изображения в градациях серого
        gray_image = cv2.imread(file[0], 0)
        # Приведение всех изображений к одному размеру ширина x
         ↪  высота)
        resized_image = cv2.resize(gray_image, (250, 60))

        # Морфологический фильтр (дилатация) для сужения символов
         ↪  и более четкого отделения их друг от друга
        morph_kernel = np.ones((3, 3))
        dilatation_image = cv2.dilate(resized_image,
         ↪  kernel=morph_kernel, iterations=1)

        # Применяем пороговую обработку, чтобы получить только
         ↪  черные и белые пиксели
        _, thresholder = cv2.threshold(
            dilatation_image,
            0,
            255,
```

```
            cv2.THRESH_BINARY + cv2.THRESH_OTSU
        )

        cv2.imwrite(file[0], thresholder)
```

Листинг 6.3 Исходный код для создания датасета в формате тензоров

```
import pandas as pd
import tensorflow as tf
from keras._tf_keras.keras.preprocessing.sequence import
 ↪  pad_sequences
from sklearn.model_selection import train_test_split


def parse_data(image_path: list, encoder_labels: list,
 ↪  decoder_labels: list) -> tuple[tf.Tensor, list]:
    '''Функция для склеивания изображений и лейблов для
     ↪  датасета'''
    image = tf.io.read_file(image_path)
    image = tf.image.decode_png(image, channels=1)
    image = tf.cast(image, tf.float32) / 255.0

    return (image, encoder_labels), decoder_labels


def create_dataset(images: list, encoder_labels: list,
 ↪  decoder_labels: list, shuffle = True, batch_size = 32) ->
 ↪  tf.data.Dataset:
    '''Функция для создания датасета, понятного для TensorFlow'''
    dataset = tf.data.Dataset.from_tensor_slices((images,
     ↪  encoder_labels, decoder_labels))
    dataset = dataset.map(lambda x, y, w: parse_data(x, y, w))
    if shuffle == True:
        dataset = dataset.shuffle(len(images)).batch(batch_size)
    else:
        dataset = dataset.batch(batch_size)

    return dataset


def create_dataframe(images: list) -> pd.DataFrame:
    '''Функция для создания датафреймов на основе списков'''

    # Создание файла с лейблами о содержимом изображений с CAPTCHA
```

```
32      filenames = [objects[0] for objects in images]
33      list_labels = [objects[1] for objects in images]
34
35      # Создание DataFrame для сохранения соотвествия между путями,
        ↪  лейблами и размерами для каждого элемента датасета
36      data = {
37          'filename': filenames,
38          'label': list_labels,
39      }
40
41      return pd.DataFrame(data)
42
43
44  def preparing_dataset(dataframe: pd.DataFrame, alphabet: str,
    ↪  shuffle = True) -> tuple[
45          tuple[tf.data.Dataset, tf.data.Dataset, tf.data.Dataset,
            ↪  list],
46          tuple[tf.data.Dataset, tf.data.Dataset, tf.data.Dataset,
            ↪  list]
47      ]:
48      '''Подготовка датасета'''
49
50      # Сохранение отдельных составляющих DataFrame
51      X_captcha, y_captcha = dataframe['filename'].tolist(),
        ↪  dataframe['label'].tolist()
52
53      dict_alphabet = {alphabet[i]:i for i in range(len(alphabet))}
54      start_token = len(alphabet)  # Индекс токена <start>
55      end_token = len(alphabet) + 1  # Индекс токена <end>
56
57      # Кодируем лейблы с добавлением токена <start> для кодера
58      encoder_labels = [[start_token] + [dict_alphabet[char] for
        ↪  char in label] for label in y_captcha]
59
60      # Кодируем лейблы с добавлением токена <end> для декодера
61      decoder_labels = [[dict_alphabet[char] for char in label] +
        ↪  [end_token] for label in y_captcha]
62
63      # Преобразование меток в тензоры
64      encoder_tensors = pad_sequences(encoder_labels, maxlen=8,
        ↪  padding='post')
65      decoder_tensors = pad_sequences(decoder_labels, maxlen=8,
        ↪  padding='post')
66
67      # Создание датасета
```

```
68      dataset = create_dataset(X_captcha, encoder_tensors,
        ↪ decoder_tensors, shuffle)
69
70      return dataset
71
72
73  def create_dataset_for_captcha(filenames: list, alphabet: str) ->
    ↪ tuple[
74          tuple[tf.data.Dataset, tf.data.Dataset, tf.data.Dataset,
            ↪ list],
75          tuple[tf.data.Dataset, tf.data.Dataset, tf.data.Dataset,
            ↪ list]
76      ]:
77      '''Функция для создания датасета на основе алфавита и имен
        ↪ файлов'''
78
79      # Создание датафрейма для удобства последующей обработки
80      captcha_dataframe = create_dataframe(filenames)
81
82      # Разделение датасета на обучающую и тестовую выборки
83      train_captcha_df, test_captcha_df =
        ↪ train_test_split(captcha_dataframe, test_size=0.2,
        ↪ random_state=42)
84      # Разделение тестовой части датасета на валидационную и
        ↪ тестовую выборки
85      val_captcha_df, test_captcha_df =
        ↪ train_test_split(test_captcha_df, test_size=0.5,
        ↪ random_state=42)
86
87      train_dataset = preparing_dataset(train_captcha_df, alphabet)
88      val_dataset = preparing_dataset(val_captcha_df, alphabet)
89      test_dataset = preparing_dataset(test_captcha_df, alphabet,
        ↪ False)
90
91      return train_dataset, val_dataset, test_dataset
```

Листинг 6.4 Исходный код CRNN модели

```
1  import tensorflow as tf
2  import numpy as np
3  from keras._tf_keras.keras.layers import Input, Conv2D,
   ↪ MaxPooling2D, Reshape, Dense, Dropout, Bidirectional, GRU,
   ↪ BatchNormalization
4  from keras._tf_keras.keras.regularizers import l2
```

```
from keras._tf_keras.keras.models import Model
from keras._tf_keras.keras import backend as K
from keras._tf_keras.keras.backend import ctc_decode
from keras._tf_keras.keras.optimizers import Adam
from keras._tf_keras.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from keras._tf_keras.keras.saving import register_keras_serializable


def decode_predictions(preds, max_length, alphabet):
    # Используем CTC-декодирование для предсказаний
    decoded_preds, _ = ctc_decode(preds, input_length=np.ones(preds.shape[0]) * preds.shape[1])
    texts = []
    for seq in decoded_preds[0]:
        text = ''.join([alphabet[i] for i in seq.numpy() if i != -1])  # Исключаем 'blank' символы
        texts.append(text)
    return texts


def decode_batch_predictions(pred):
    # CTC decode
    decoded, _ = ctc_decode(pred, input_length=np.ones(pred.shape[0]) * pred.shape[1],
                            greedy=True)
    decoded_texts = []

    # Преобразование в текст
    for seq in decoded[0]:
        text = ''.join([chr(x) for x in seq if x != -1])  # Пропускаем -1 (пустые символы CTC)
        decoded_texts.append(text)
    return decoded_texts


# Функция CTC Loss
# Функция для декодирования предсказаний модели
@register_keras_serializable(package='Custom', name='ctc_loss')
def ctc_loss(y_true, y_pred):
    # Формируем входные данные для CTC
    input_lenght = tf.ones(shape=(tf.shape(y_pred)[0], 1)) * tf.cast(tf.shape(y_pred)[1], tf.float32)
    label_length = tf.ones(shape=(tf.shape(y_true)[0], 1)) * 7
```

```
43      return tf.reduce_mean(K.ctc_batch_cost(y_true, y_pred,
        ↪  input_lenght, label_length))
44
45
46  # Модель
47  def build_model(num_of_classes):
48      '''Создание модели'''
49      # Входной слой
50      input_layer = Input((60, 250, 1))
51
52      # Первый сверточный блок
53      x = Conv2D(32, (3, 3), activation='relu', padding='same',
        ↪  kernel_regularizer=l2(0.003))(input_layer)
54      x = BatchNormalization()(x)
55      x = Conv2D(32, (3, 3), activation='relu',
        ↪  kernel_regularizer=l2(0.003))(x)
56      x = MaxPooling2D((1, 2))(x)
57      x = Dropout(0.25)(x)  # Dropout после каждого блока
58
59      # Второй сверточный блок
60      x = Conv2D(64, (3, 3), activation='relu', padding='same',
        ↪  kernel_regularizer=l2(0.003))(x)
61      x = BatchNormalization()(x)
62      x = Conv2D(64, (3, 3), activation='relu',
        ↪  kernel_regularizer=l2(0.003))(x)
63      x = MaxPooling2D((1, 2))(x)
64      x = Dropout(0.3)(x)
65
66      # Третий сверточный блок
67      x = Conv2D(128, (3, 3), activation='relu', padding='same',
        ↪  kernel_regularizer=l2(0.003))(x)
68      x = BatchNormalization()(x)
69      x = Conv2D(128, (3, 3), activation='relu',
        ↪  kernel_regularizer=l2(0.003))(x)
70      x = MaxPooling2D((1, 2))(x)
71      x = Dropout(0.4)(x)
72
73      # Изменяем размерность тензора
74      x = Reshape((-1, x.shape[-1] * x.shape[-2]))(x)
75
76      # Первый рекурентный блок
77      x = Bidirectional(GRU(128, return_sequences=True))(x)
78      x = BatchNormalization()(x)
79      x = Dropout(0.6)(x)
80
```

```
81      # Второй рекурентный блок
82      x = Bidirectional(GRU(128, return_sequences=True))(x)
83      x = BatchNormalization()(x)
84      x = Dropout(0.6)(x)
85
86      # Третий рекурентный блок
87      x = Bidirectional(GRU(128, return_sequences=True))(x)
88      x = BatchNormalization()(x)
89      x = Dropout(0.6)(x)
90
91      # Выходной слой
92      outputs = Dense(num_of_classes + 1, activation='softmax')(x)
93
94      # Создание модели
95      model = Model(inputs=input_layer, outputs=outputs)
96
97      return model
98
99
100 def fit_crnn(num_of_classes, train, val):
101     # Компиляция модели
102     model = build_model(num_of_classes)
103     optimizer = Adam(learning_rate=0.001, weight_decay=1e-6)
104     model.compile(
105         loss=ctc_loss,
106         optimizer=optimizer
107     )
108
109     # Вывод структуры модели
110     model.summary()
111
112     lr_sheduler = ReduceLROnPlateau(monitor='val_loss',
        ↪  factor=0.5, patience=3, min_lr=1e-6)
113     early_stop = EarlyStopping(monitor='val_loss', patience=3,
        ↪  restore_best_weights=True)
114
115     # Обучение модели
116     history = model.fit(
117         train,
118         validation_data=val,
119         epochs=15,
120         callbacks=[early_stop, lr_sheduler]
121     )
122
123     model.save('crnn_model.keras')
```

```
124
125     return model, history
```

Листинг 6.5 Исходный код Seq-to-Seq модели

```
1  import tensorflow as tf
2  from keras._tf_keras.keras import layers, Model
3  from keras._tf_keras.keras.callbacks import EarlyStopping,
   ↪  ReduceLROnPlateau
4
5  from create_dataset import create_dataset_for_captcha
6
7
8  # Обновлённый кодировщик
9  def build_encoder():
10     encoder_inputs = layers.Input(shape=(60, 250, 1),
       ↪  name="encoder_inputs")
11
12     # Первый сверточный блок
13     x = layers.Conv2D(32, (3, 3), activation='relu',
       ↪  padding='same')(encoder_inputs)
14     x = layers.BatchNormalization()(x)
15     x = layers.Conv2D(32, (3, 3), activation='relu')(x)
16     x = layers.MaxPooling2D((2, 2))(x)
17
18     # Второй сверточный блок
19     x = layers.Conv2D(64, (3, 3), activation='relu',
       ↪  padding='same')(x)
20     x = layers.BatchNormalization()(x)
21     x = layers.Conv2D(64, (3, 3), activation='relu')(x)
22     x = layers.MaxPooling2D((2, 2))(x)
23
24     # Третий сверточный блок
25     x = layers.Conv2D(128, (3, 3), activation='relu',
       ↪  padding='same')(x)
26     x = layers.BatchNormalization()(x)
27     x = layers.Conv2D(128, (3, 3), activation='relu')(x)
28     x = layers.MaxPooling2D((2, 2))(x)
29
30     # Четвертый сверточный блок
31     x = layers.Conv2D(256, (3, 3), activation='relu',
       ↪  padding='same')(x)
32     x = layers.BatchNormalization()(x)
33     x = layers.Conv2D(256, (3, 3), activation='relu')(x)
```

```
34      x = layers.MaxPooling2D((2, 2))(x)
35
36      x = layers.GlobalAveragePooling2D()(x)
37      x = layers.Dense(256, activation="relu")(x)
38      x = layers.BatchNormalization()(x)
39      x = layers.Reshape((1, 256))(x)  # Добавляем временное
        ↪ измерение
40
41      # RNN слой
42      encoder_output, encoder_state = layers.GRU(256,
        ↪ return_sequences=True, return_state=True)(x)
43
44      return Model(encoder_inputs, [encoder_output, encoder_state],
        ↪ name="encoder")
45
46
47  # Декодировщик с Attention
48  def build_decoder(alphabet_size):
49      decoder_inputs = layers.Input(shape=(None,),
        ↪ name="decoder_inputs")
50      encoder_state_input = layers.Input(shape=(256,),
        ↪ name="encoder_state_input")
51
52      x = layers.Embedding(alphabet_size, 128)(decoder_inputs)
53      rnn_output, decoder_state = layers.GRU(256,
        ↪ return_sequences=True, return_state=True)(x,
        ↪ initial_state=encoder_state_input)
54
55      # Attention
56      attention_output = layers.AdditiveAttention()([rnn_output,
        ↪ encoder_state_input])
57      x = layers.Concatenate()([rnn_output, attention_output])
58      decoder_outputs = layers.Dense(alphabet_size,
        ↪ activation="softmax")(x)
59
60      return Model([decoder_inputs, encoder_state_input],
        ↪ [decoder_outputs, decoder_state], name="decoder")
61
62
63  def fit_seq_to_seq(number_of_classes: int, train_dataset:
    ↪ tf.data.Dataset, val_dataset: tf.data.Dataset) ->
    ↪ tuple[Model, dict]:
64      # Построение полной модели
65      encoder = build_encoder()
66      decoder = build_decoder(number_of_classes + 2)
```

```
67
68      # Полная модель
69      encoder_inputs = encoder.input
70      decoder_inputs = layers.Input(shape=(None,),
        ↪  name="decoder_inputs")
71
72      _, encoder_state = encoder(encoder_inputs)
73      decoder_output, _ = decoder([decoder_inputs, encoder_state])
74
75      seq2seq_model = Model([encoder_inputs, decoder_inputs],
        ↪  decoder_output, name="seq2seq_model")
76
77      # Компиляция модели
78      seq2seq_model.compile(
79          loss="sparse_categorical_crossentropy",
80          optimizer="adam",
81          metrics=["accuracy"]
82      )
83
84      seq2seq_model.summary()
85
86      lr_sheduler = ReduceLROnPlateau(monitor='val_loss',
        ↪  factor=0.5, patience=3, min_lr=1e-6)
87      early_stop = EarlyStopping(monitor='val_loss', patience=3,
        ↪  restore_best_weights=True)
88
89      # Обучение модели
90      history = seq2seq_model.fit(
91          train_dataset,
92          validation_data=val_dataset,
93          epochs=20,
94          callbacks=[early_stop, lr_sheduler]
95      )
96
97      seq2seq_model.save('seq_to_seq_model.keras')
98
99      return seq2seq_model, history
```

Листинг 6.6 Исходный код тестирования модели

```
1  import numpy as np
2  import tensorflow as tf
3  from keras._tf_keras.keras.models import load_model
4
```

```
5
6  if __name__ == '__main__':
7      import matplotlib.pyplot as plt
8      import seaborn as sbn
9
10     # Алфавит допустимых символов
11     alphabet = ' ABCDEFGHJKLMNPQRSTWXYZ023456789'
12
13     list_filenames = np.load('data.npy', allow_pickle=True)
14     # Создание единого датасета
15     captcha_dataset = create_dataset_for_captcha(list_filenames,
       ↪  alphabet)
16     if False:
17         # Обучение модели
18         model_captcha, history_captcha =
           ↪  fit_seq_to_seq(len(alphabet), captcha_dataset[0],
           ↪  captcha_dataset[1])
19         # Построение графика изменения val_loss и loss
20         plt.plot(history_captcha.history['loss'], label='Training
           ↪  Loss')
21         plt.plot(history_captcha.history['val_loss'],
           ↪  label='Validation Loss')
22         plt.xlabel('Epoch')
23         plt.ylabel('Loss')
24         plt.legend()
25         # Сохраняем график для отчета
26         plt.savefig('Model_loss.png')
27
28     # Загружаем предобученную модель и получаем предсказания для
       ↪  тестовой выборки
29     model = load_model('seq_to_seq_model.keras')
30     predictions = model.predict(captcha_dataset[2])
31
32     # Переводим предсказания из представления вероятностей в
       ↪  классы
33     pred_classes = np.argmax(predictions, axis=-1)
34     captcha_labels = [label.numpy() for _, label in
       ↪  captcha_dataset[2].unbatch()]
35     captcha_labels = np.array(captcha_labels)
36
37     # Убираем padding
38     def remove_padding(sequences, padding_value=0):
39         return [seq[seq != padding_value] for seq in sequences]
40
41     # Убираем padding из предсказаний и меток
```

```
    pred_classes_no_padding = remove_padding(pred_classes,
    ↪  padding_value=0)
    true_labels_no_padding =
    ↪  remove_padding(np.array(captcha_labels), padding_value=0)

    # Проверяем размеры списков после удаления padding
    print(f"Количество предсказаний:
    ↪  {len(pred_classes_no_padding)}")
    print(f"Количество меток: {len(true_labels_no_padding)}")

    # Проверяем совпадение предсказаний и истинных меток
    ↪  посимвольно
    sequence_accuracy = np.mean(
        [np.array_equal(pred, true) for pred, true in
        ↪  zip(pred_classes, captcha_labels)]
    )
    print(f"Точность последовательностей (без padding):
    ↪  {sequence_accuracy:.4f}")

    # Расчет точности символов (character-level accuracy)
    total_characters = np.prod(captcha_labels.shape)
    correct_characters = np.sum(pred_classes == captcha_labels)
    character_accuracy = correct_characters / total_characters
    print(f"Точность символов: {character_accuracy:.4f}")

    from sklearn.metrics import confusion_matrix
    # Построение матрицы ошибок для анализа
    true_symb, pred_symb = [], []

    for true_seq, pred_seq in zip(true_labels_no_padding,
    ↪  pred_classes_no_padding):
        true_symb.extend(true_seq)
        pred_symb.extend(pred_seq)
    cm = confusion_matrix(true_symb, pred_symb)

    plt.figure(figsize=(10, 7))
    sbn.heatmap(cm, annot=True, fmt='g', cmap='Blues')
    plt.xlabel('Predicted classes')
    plt.ylabel('True classes')
    plt.title('Confusion matrix')
    # plt.show()
    plt.savefig('Confusion_matrix.png')

    from collections import defaultdict

```

```
    sequence_accuracy_by_length = defaultdict(list)
    for pred, true in zip(pred_classes_no_padding,
    ↪  true_labels_no_padding):
        seq_len = len(true)
        is_correct = np.array_equal(pred, true)
        sequence_accuracy_by_length[seq_len].append(is_correct)

    # Считаем точность для каждой длины
    for length, results in sequence_accuracy_by_length.items():
        acc = np.mean(results)
        print(f"Длина {length}: Точность {acc:.4f}")
```