Министерство образования и науки Российской Федерации
Московский физико-технический институт
(национальный исследовательский университет)

Факультет инноваций и высоких технологий
Кафедра корпоративных информационных систем

Выпускная квалификационная работа бакалавра по направлению 01.03.02
«Прикладные математика и информатика»

Разработка механизма автоматического перевода текста на фотографии для мобильных устройств

Студент 597 группы
Ткаченко Д. А.

Научный руководитель
Родюков А. В.

Долгопрудный
2019

Содержание

1. Введение **1**

2. Дизайн и разработка интерфейса **3**

2.1. Выбор инструмента . . . 3

2.2. Разработка интерфейсов компонент . . . 3

3. Архитектурная разработка **9**

3.1. Разработка структурной архитектуры приложения . . . 9

3.2. Разделение ответственности компонент и их взаимодействия . . . 11

4. Разработка приложения **14**

4.1. Swift и его особенности . . . 14

4.2. База данных с помощью стандартного механизма от Apple и протоколо-ориентированность . . . 14

4.3. ViewModel с помощью типизированных перечислений с присоединяемыми значениями . . . 16

4.4. Директива class для протокола . . . 17

5. Библиотеки сторонних разработчиков **19**

5.1. Система контроля внешних библиотек CocoaPods . . . 19

5.2. Библиотека Koloda . . . 20

5.3. Библиотека Toast-Swift . . . 20

5.4. Библиотеки компьютерного зрения Firebase, Tesseract, CoreML Vision . . . 21

5.5. Работа с Google Translate API . . . 22

6. Архитектура и обучение модели распознавания **23**

6.1. Описание архитектуры модели . . . 23

6.2. Обучение нейронной сети, подсчет CTC функции потерь . . . 25

6.3. Генерация обучающей выборки . . . 27

7. Портирование модели в формат CoreML **28**

7.1. Библиотека coremltools . . . 28

7.2. Ограничения на конвертируемую модель . . . 28

7.3. Процесс конвертации . . . 29

8. Предобработка изображений в Swift **31**
8.1. Механизм использования C++ библиотеки OpenCV в Swift 31
8.2. Предобработка изображения 33

9. Тестирование **35**
9.1. Сравнительное тестирование моделей распознавания 35

10. Заключение **38**
10.1. Результаты и выводы 38

Список литературы **39**

Введение

С развитием технологий человечество вошло в новую эпоху - эру мобильных устройств. Их количество исчисляется миллиардами, люди проводят часы жизни за ними и уже вряд ли могут представить себя без них. В данной ситуации крайне важно использовать мобильные устройства как помощников в быту или в решении технических задач. Сейчас вычислительные мощности Вашего смартфона или планшета уже превосходят мощности стационарных компьютеров 2-3 летней давности. Таким образом, с их помощью можно решать и ресурсоемкие задачи, которые раньше были под силу только мощным системам. Примером такой задачи является распознавание текста на фотографии. Но только распознать текст мало - его можно и нужно использовать для дальнейшей обработки и получения информации. Но, если текст на незнакомом для Вас языке, то он нуждается в переводе. Такая ситуация часто встречается в путешествиях или на конференциях, которых проводится все больше благодаря глобализации, а, значит, данная тема актуальна. Таким образом и родилась идея данной работы - разработка механизма автоматического перевода текста на фотографии для мобильных устройств.

Объектом исследования данной работы являются механизмы по распознаванию текста на фотографии, библиотеки для перевода и механизмы проектирования и реализации мобильного приложения.

Предметом исследования данной работы является процесс создания приложения по автоматическому переводу на платформе iOS.

Целью данной работы является изучение архитектурной и прикладной разработки мобильного приложения и применение знаний в области машинного обучения для использования модели распознавания непосредственно на мобильном устройстве.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Разработка интерфейса приложения с учетом интуитивности и получения пользователем положительного опыта использования
2. Разработка архитектуры приложения и взаимодействия его компонент
3. Исследование и применение библиотек сторонних разработчиков
4. Выбор модели для распознавания и ее обучение

5. Портирование полученной модели в формат, пригодный для использования на мобильном устройстве

6. Тестирование полученного прототипа

Методы исследования, применямые в данной работе:

1. Анализ предметной области

2. Разработка дизайна и архитектуры приложения

3. Анализ и использование реализованных библиотек перевода

4. Реализация модели распознавания текста, ее портирование в формат, пригодный для использования на мобильном устройстве

5. Программная реализация приложения на платформе iOS

Дизайн и разработка интерфейса

2.1 Выбор инструмента

Основная задача при разработке интерфейса приложения – создание макета и дальнейший перенос его элементов в само приложения. Сейчас на рынке достаточно хороших инструментов для дизайна и создания макета (Zeplin, Sketch, Figma). В данной работе использовался инструмент Figma по нескольким причинам:

- Поддержка всех платформ (Windows, MacOS, браузерный интерфейс)
- Возможность прототипирования и создания интерактивных переходов между экранами
- Хранение проекта в облаке
- Возможность создания публичной ссылки для просмотра проекта
- Интуитивность и наличие всех необходимых инструментов

2.2 Разработка интерфейсов компонент

Начальным этапом дизайна является выбор цветовой палитры приложения. В качестве цветовой палитры было решено использовать спокойное сочетание цветов, на фоне которых довольно контрастно бы смотрелось фотография, сделанная пользователем. Основные три цвета палитры – черный, серо-синий и бирюзовый.

В качестве основного шрифта приложения был выбран шрифт Rationale за свою футуристичность. Одна из ключевых компонент дизайна – логотип приложения. В нем крайне важно лаконично и понятно отразить суть приложения, чтобы заинтересовать пользователя уже на этапе просмотра чартов в AppStore или Google Play. Итоговая идея – «облако» по образу подложки сообщений в мессенджерах, внутри которого схематично изображены реплики с языками, на котором они написаны. Таким образом показаны главные цели приложения – передача и получение информации и возможность перевода, интернациональность.

Рис. 1. Финальный вариант логотипа приложения.

Приветственный экран приложения было решено сделать стандартным образом – логотип посередине, название приложения прибито к нижней части экрана. Для дальнейшей разработки дизайна нужно было понять и сформулировать функциональность приложения, в результате чего был составлен список базовой функциональности:

- Возможность создания фотографии, с которой будет осуществляться перевод текста
- Отображение результата перевода, языка текста на фотографии и языка, на который произошел перевод
- Возможность копирования результата перевода в буфер обмена
- Хранение переводов с возможностью удаления
- Настройки и их переключение
- Информация о приложении
- Обратная связь, возможность написать автору

На основе этого списка был составлен список основных экранов приложения:

- Приветственный экран
- Основной экран
- Экран настроек
- Экран конкретной настройки с выбором возможных значений

- Экран сохраненных переводов
- Экран «О приложении» с кнопкой «Написать разработчику»

Для основного экрана и для экрана сохраненных переводов было решено создать похожий интерфейс – основную часть занимает карточка перевода (колода карточек переводов в случае экрана с сохраненными переводами). В карточке перевода необходимо отобразить основную информацию – непосредственно фотографию, итоговый результат, языки и кнопки для реализации функционала (копирование в буфер, сохранение или удаление из списка переводов). Большую часть данной карточки, что логично, занимает фотография текста, которая в случае основного экрана служит отображением изображения с камеры. Под фотографией расположено поле, содержащее текст итогового перевода, а под ним кнопка сохранения/удаления из списка сохраненных переводов, целевой и конечный языки перевода, кнопка копирования итога в буфер. При оформлении карточки было решено использовать скругленные края и все три основные цвета палитры.

Рис. 2. Финальный макет карточки перевода.

На основном экране должна быть возможность перехода к экрану настроек и экрану сохраненных переводов. Так же должны быть возможность снять фотогра-

фию, чтобы инициировать процесс перевода. Для этого в панель управления основного экрана добавлены три кнопки, выполняющие соответствующие задачи.

Рис. 3. Панель управления основного экрана.

Каждый экран, кроме основного и экрана «О Приложении», подписан заголовком. Для навигации среди экранов предусмотрена кнопка «Назад» в левой верхней части в виде стрелки влево. Так как карточка перевода занимает почти весь экран, то на экране сохраненных переводов было решено сделать навигацию между переводами с помощью смахиваний перевода наверху стека в сторону. Получилось своеобразная колода карточек с переводами. В правом верхнем углу предусмотрена кнопка для отмены последнего действия, а по истечению карточек в колоде появляется надпись о том, что переводы закончились. Если же список изначально пуст, надпись посередине экрана оповещает об этом. Экраны настроек традиционно сделаны в единообразном стиле – это таблица с ячейками. Ячейки в приложении было решено сделать двух типов, а именно ячейка с дополнительной информацией и без нее. Для ячейки, в которой отображается только основная информация, текст центрируется, аналогично остальному интерфейсу приложения использованы скругленные края и общая палитра.

Рис. 4. Пример ячейки настроек с основной информацией.

В ячейке, содержащей дополнительную информацию (помимо основной) центрирование не происходит. Основная информация располагается по левому краю, дополнительная – по правому краю.

Рис. 5. Пример ячейки настроек с основной и дополнительной информацией.

Остальные экраны конкретных настроек представляют собой таблицу ячеек с основной информацией. Для перехода с экрана настроек к экрану «О приложении» отдельной ячейки не предусмотрено. За переход на данный экран отвечает надпись внизу экрана настроек.

Рис. 6. Переход к экрану о приложении.

Сам экран содержит основную информацию – логотип, название, версию, копирайт и возможность написать разработчику на почту. Для отличия кнопки обратной связи от остальных надписей было решено выделить ее подчеркиванием:

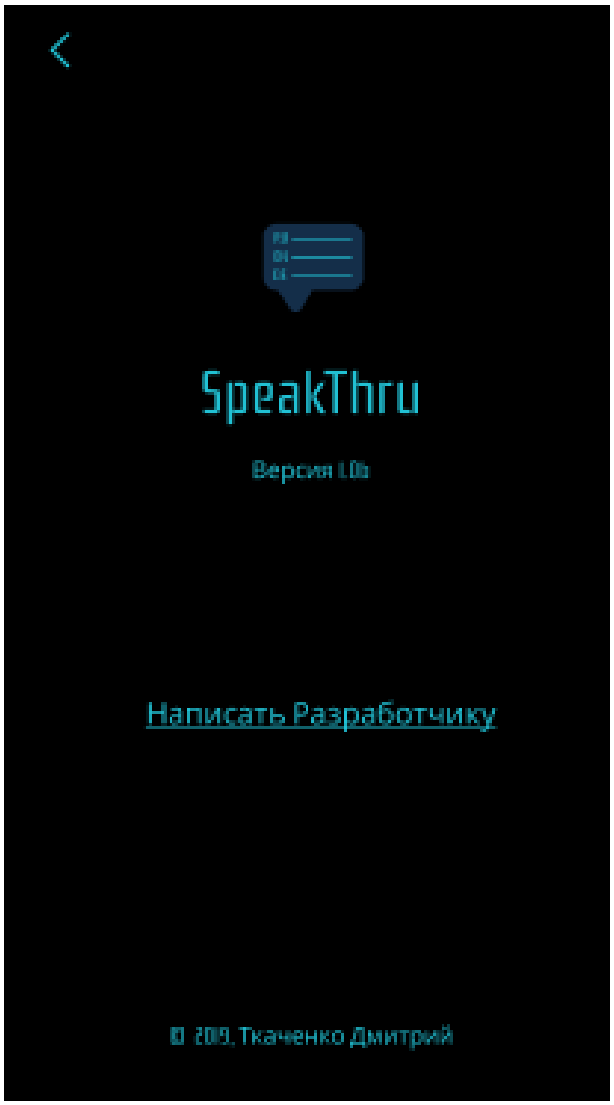

Рис. 7. Экран "О приложении".

В итоге, разработанный интерфейс приложения решает поставленные перед ним задачи:

- Отсутствие перегруженности элементами
- Реализация базовой функциональности
- Возможность изменения настроек
- Спокойная палитра и шрифт, улучшающие опыт взаимодействия пользователя

- Удобство просмотра перевода в силу расположения почти по всей поверхности экрана карточки с переводом
- Возможность обратной связи

Архитектурная разработка

3.1 Разработка структурной архитектуры приложения

Разработка архитектуры – одна из важнейших частей разработки приложения. Для быстрого и безопасного исполнения кода необходимо четко разделить ответственность между классами и объектами, продумать механизмы их взаимодействия. Так же очень важна модульность, которая позволяет быстро реагировать на изменения бизнес-логики или дизайна в коде.

При разработке данного приложения были использованы несколько подходов. Первый из них – разработка по принципу луковой или слоистой архитектуры [1]. Основная ее идея – разделение компонент программы на несколько слоев и непосредственное взаимодействие компонент только внутри одного слоя или соседних слоев. Таким образом довольно четко структурируются задачи компонент и механизмы их взаимодействия. Внутренний слой («ядро») – это те компоненты, которые решают самые базовые задачи и могут быть переиспользованы (полностью или частично) в других приложениях. Определимся, какие компоненты будут являться частью нашего «ядра»:

- Класс-синглтон, имплементирующий сущность приложения
- Класс-синглтон, содержащий текущий контекст, то есть состояние базы данных и сетевых взаимодействий
- Класс-синглтон, который контролирует переходы между экранами приложения
- Класс-синглтон, имплементирующий обертку над **API** переводчика
- Класс, который контролирует работу с камерой и видеопотоком с нее
- Класс, непосредственно отвечающий за процесс распознавания текста с фотографии

Рис. 8. Визуальное представление ядра архитектуры.

Теперь необходимо понять, какие задачи решает непосредственно каждая компонента, а так же, как они взаимодействуют между собой. Начнем с основной – приложения. Эта компонента является связующим звеном между остальными и производит их начальную настройку и инициализацию. Приложение непосредственно хранит контекст, класс, отвечающий за переходы, окно приложения и ключ для API перевода. Так же приложение хранит ссылку на базу данных, которая содержится в контексте.

Рис. 9. Визуальное представление приложения.

Сущность «Окно», о которой речи выше не было, появилась здесь из платформенных требований. На платформе iOS все представления, что размещены на экране, выстроены в иерархию. Корень этой иерархии – Окно. При старте приложения со-

здается его окно, а затем все представления размещаются в нем. Из-за того, что Окно в приложении – синглтон, а также его корневой роли, логично поместить его в Приложение.

Роль контекста, как уже было сказано выше – хранение базы данных и работа с сетью. Данные обязанности тоже разделены между двумя сущностями. При старте Контекст проверяет, первый ли запуск приложения произошел, и, в случае положительного ответа, устанавливает первоначальные настройки (язык, тип модели распознавания).

Рис. 10. Визуальное представление контекста.

Роутер (класс, отвечающий за переходы между экранами) не содержит каких-то подкомпонент. Его основная роль – при запросе конкретного экрана произвести переход на него с текущего состояния. Для этого, ему нужно Окно, которое при конструировании ему передает Приложение.

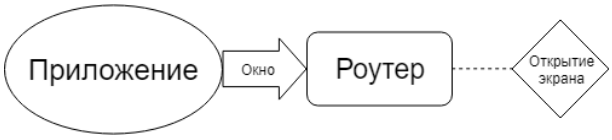

Рис. 11. Визуальное представление Роутера.

Закончив с основными компонентами ядра, можно перейти к тем, которые специфичны для нашего приложения – это элементы интерфейса, шрифты, цвета. Для хранения такой информации удобно использовать паттерн StyleKit. По сути это статичный класс, который является фабрикой для элементов интерфейса и еще каких-либо частей, связанных с визуальным представлением приложения. В нашем случае, StyleKit не является подмодулем Приложения, чтобы не нарушать возможность переиспользования.

3.2 Разделение ответственности компонент и их взаимодействия

Структурное описание архитектуры завершено и пришло время перейти к уровню более прикладному, а именно выбору архитектуры для разработки непосредственно экранов приложения. В этой части существует несколько основных подходов [2]:

1. MVC (Model-View-Controller)

2. MVVM (Model-View-View Model)
3. VIPER (View-Interactor-Presenter-Entity-Router)
4. Смешение каких-либо из перечисленных
5. Иные подходы

Первый подход, MVC, традиционно используется Apple и предлагается для использования разработчикам на платформе iOS. В данном подходе существует несколько недостатков, связанных с разделением ролей. Корень этих недостатков – очень большая зона ответственности класса Controller. Два других класса, Model и View, отвечают лишь за данные и их визуальное представление, в то время как Controller несет ответственность за их взаимодействие и всю работу с данными. В больших проектах список таких задач достаточно велик: сетевые запросы, парсинг их ответов, чтение/запись моделей данных, их преобразование для отображения, реакция на события от пользователя и пр. Таким образом, класс Controller разрастается до внушительных размеров, тем самым увеличивается кодовая база и ее становится крайне тяжело переиспользовать и поддерживать в длительном периоде. Частично устранить данные недостатки могут следующие две модели, поэтому при разработке данного приложения были использованы принципы из VIPER и MVVM. Раскроем эти подходы подробнее. Парадигма VIPER [3] предполагает разделение ответственности на 5 типов классов, которые выполняют следующие роли:

1. Визуальное отображение данных и взаимодействие с пользователем
2. Взаимодействие с данными в базе и их изменение или чтение, взаимодействие с сетью
3. Презентеры визуальных представлений, которые отвечают за взаимодействие между представлениями и данными
4. Сущности данных, доступные только для чтения
5. Перемещение между экранами приложения (Роутер)

Выше мы уже описали Роутер нашего приложения. Сущностями для чтения будут служить переводы, представлениями – карточки переводов и иные визуальные отображения в приложении. Четкое разделение ролей, предложенное в VIPER, обеспечивает безопасность и удобство в разработке. Такой код легко покрывается тестами и позволяет его поддерживать в долгой перспективе.

Для чего использовать MVVM, если почти все проблемы решает VIPER? Чтобы это понять, стоит немного раскрыть структуру и основные идеи MVVM. Роли классов в MVVM:

1. Model – создание моделей данных, имплементация бизнес-логики

2. View – интерфейсы представлений, логика отображения и обработка событий от пользователя

3. ViewModel – преобразование данных от Модели для отображения во View, использование событий от View для обновления данных в Модели

Таким образом, MVVM более легковесная структура, нежели чем VIPER, и подойдет для проектирования более мелких модулей, в которых выполняется простая бизнес-логика и отсутствует роутинг. Использование MVVM в таких случаях вместо VIPER позволит сократить время разработки, при этом не ухудшая качество архитектуры и кода. Примером такого модуля может служить переиспользуемый экран настроек приложения.

Разработка приложения

4.1 Swift и его особенности

Основная разработка приложения велась на языке Swift. Это современный (представлен в 2014 году) и гибкий язык программирования, разработанный компанией Apple, который пришел на замену языку Objective-C. Более того, Swift может исполнять рантайм Objective-C, что делает возможным использования Swift, Objective-C, C и даже C++ в рамках одной программы, что будет освещено в главе 7.

Язык Swift является объектно-ориентированным и протоколо-ориентированным. Вторая его особенность позволяет писать более безопасный код, скрывая реализацию за интерфейсом, который в Swift носит название протокола. Стоит заметить, что с помощью механизма расширений, любой класс в Swift можно сделать удовлетворяющим конкретному протоколу, а так же любой класс может удовлетворять сразу нескольким протоколам, в отличие от механизма наследования (в Swift каждый класс может быть наследником максимум одного класса). Чтобы лучше разобраться с этой парадигмой, в проекте был реализован класс базы данных, функционал которой закрыт протоколом и внешние классы ничего не знают об их реализации.

4.2 База данных с помощью стандартного механизма от Apple и протоколо-ориентированность

Для начала стоит определиться с механизмом работы базы данных. В качестве основы было выбрано персистентное хранилище UserDefaults от Apple. Оно позволяет хранить пары ключ-значение на диске и благодаря этому данные доступны от запуска к запуску приложения. Для того, чтобы объект мог стать значением в базе UserDefaults, необходимо, чтобы он мог быть архивирован. Для этого требуется реализовать инициализатор с помощью архиватора:

```
required convenience init(coder aDecoder: NSCoder)
```

А так же удовлетворить протокол NSCoding классом объекта, который мы хотим сохранять. Здесь можно использовать возможность, описанную выше – сначала объявить класс объекта и реализовать его функционал, а затем с помощью меха-

низма расширения удовлетворить протокол NSCoding. Пример для класса объекта перевода:

```
extension STTranslation: NSCoding {
    func encode(with aCoder: NSCoder) {
        aCoder.encode(id, forKey: "id")
        aCoder.encode(imageToTranslate, forKey: "image")
        aCoder.encode(translatedText, forKey: "text")
        aCoder.encode(isSaved, forKey: "saved")
        aCoder.encode(fromLanguage, forKey: "from")
        aCoder.encode(toLanguage, forKey: "to")
    }
}
```

Как можно заметить, поля класса более простых типов сохраняются по определенным ключам. Почему их не нужно так же отдельно архивировать? Дело в том, что стандартный архиватор для UserDefaults по умолчанию умеет архивировать простые объекты типов String, Int, Float, Double, URL, Bool, Data, Date и более сложные структуры типов Array и Dictionary, состоящие из простых типов, перечисленных выше или сложных объектов, удовлетворяющих протоколу NSCoding. В нашем случае сложным объектом будет словарь переводов, ключами в котором будут служить уникальные идентификаторы перевода, а значениями сами переводы. Благодаря тому, что наш класс перевода удовлетворяет протоколу NSCoding, мы можем свободно сохранить словарь, содержащий эти переводы, в UserDefaults.

Определимся с протоколами, необходимыми при реализации базы данных. Конечно, в первую очередь стоит создать протокол, описывающий функционал базы, который должен быть доступен снаружи. Нам понадобится возможность подписываться или отписываться от обновлений в базе, получать данные из нее и сохранять данные внутри, удалять некоторые данные, хранить и устанавливать целевой язык и модель распознавания. Так же в момент аварийного завершения приложения или его перехода в фоновый режим, стоит осуществлять дамп всех данных на диск, а при первом старте выставлять стандартные настройки. Таким образом, можно описать протокол базы данных:

```
protocol STDatabase {
    func isApplicationRunsFirstTime() -> Bool

    func forceDump()

    func add(listener: STDatabaseListener)
    func remove(listener: STDatabaseListener)
```

```
    func getAllTranslations() -> [String : STTranslation]
    func removeAllTranslations() -> Bool
    func getTranslation(with id: String) -> STTranslation?
    func store(translation: STTranslation) -> Bool
   ...
}
```

Так же нам понадобится протокол подписчика на обновления базы. Что нужно знать о таком подписчике? Его уникальный идентификатор, чтобы различать их между собой, и метод-делегат, который нужно вызвать у каждого подписчика после обновления данных. Готов протокол подписчика базы данных:

```
protocol STDatabaseListener {
    func onDataUpdated()
    func id() -> String
}
```

Таким образом, базе данных совершенно неважно знать природу ее подписчика, достаточно реализации всего двух методов. Так же как и подписчику совершенно неважно знать реализации базы данных, у него есть интерфейс, с которым он может работать напрямую. Таким образом улучшается безопасность и читабельность кодовой базы приложения, ведь зачастую для понимания архитектуры какого-либо модуля, достаточно понять, какие объекты исполняют конкретные протоколы и контракты, опуская детали реализации.

4.3 ViewModel с помощью типизированных перечислений с присоединяемыми значениями

Другая отличительная особенность языка Swift – типизированные перечисления с присоединяемыми значениями. Эту особенность удобно использовать в ситуациях, когда у какого-либо значение ограниченное число типов, все из которых известны заранее, а у каждого конкретного типа в связи должны находиться конкретные структуры. В нашем проекте есть место, где все это применимо – переиспользование экрана, построенного на архитектуре MVVM. Действительно, такой экран в нашем приложении – экран настроек. Заранее известно, что предусмотрено два типа ячеек в таблице – ячейка, содержащая только текст и ячейка, содержащая текст и дополнительную информацию. Более того, исходя из типа ячейки можно строить ее конкретный layout (чем может заниматься отдельная фабрика), а в присоединяемое значение можно положить и замыкание, вызываемое при нажатии на данную ячейку. Таким образом, можно описать типизированное перечисление, описывающее ViewModel нашей ячейки:

```
enum STSettingsCellViewModel {
    case detailedCell(
        mainText: String,
        detailedText: String,
        action: () -> Void
    )
    case textCell(
        mainText: String,
        action: () -> Void
    )
}
```

Самое важное и полезное в такой конструкции – четко определенный тип. Сразу видно, что ячейка может быть только одного из двух типов, а у каждого типа существуют определенные параметры. В коде физически нельзя проставить параметры сразу для нескольких типов. В случае использования общего класса для такого объекта, пришлось бы делать поле с текстом дополнительной информации опциональным и заводить лишнее поле типа объекта, что усложнило бы обработку каждого объекта. Также была возможность забыть обработать новый вариант при добавлении, в таком же варианте при добавлении нового case компилятор Swift укажет, что его нужно добавить во все конструкции Switch, где он обрабатывается. Более того, мы абстрагировались от типа настройки и ее содержания – тем самым экран, использующий данную ViewModel для построения, может быть переиспользован для настройки языка, модели распознавания и т.д.

Конструкция Switch, в которой рассматривается объект такого перечисления, тем самым кроме информации о типе самого объекта, получает всю остальную информацию:

```
switch model {
    case let .detailedCell(mainText, detailedText, _):
        // some code
    case let .textCell(mainText, _):
        // another code
}
```

4.4 Директива class для протокола

Еще одним примером в приложении, где используется протоколо-ориентированное программирование, является класс STRecognizer, цель которого – распознавание текста с фотографии. Логично, что объектам снаружи совсем не обязательно знать, как реализован сам процесс распознавания. Для этих объектов достаточно интерфейса,

с помощью которого они могут взаимодействовать с моделью распознавания и метод делегата, который получает результат. Таким образом, протоколы, описывающие подобное взаимодействие, выглядят так:

```
protocol STRecognizerDelegate: class {
    func onRecognized(from photo: UIImage, text: String, lang: String)
}

protocol Recognizer: class {
    func recognize(from image: UIImage)
}
```

Здесь, в отличие от протоколов выше, используется вспомогательный атрибут class, указывающий, что объект, который удовлетворяет данному протоколу, обязан быть классом. Это еще одна ступень защиты – действительно, операция распознавания и обработки полученного результата достаточно сложна, чтобы не доверять ее простым типам или структурам.

Библиотеки сторонних разработчиков

5.1 Система контроля внешних библиотек CocoaPods

При использовании внешних зависимостей одним из главных инструментов любого iOS разработчика становится CocoaPods. Это мощное средство управления зависимостями cocoa-библиотек. Обычно при использовании чужих наработок используется напрямую чужой код, но у такого подхода существует ряд недостатков:

- Сложно отследить версии библиотек и их взаимосвязи
- Нет единого реестра библиотек
- Необходимость ручного обновления кода

CocoaPods призван решить вышеперечисленные проблемы. Для начала работы с ним необходимо выполнить команду pod init в корневой директории проекта. Создадутся необходимые файлы, один из которых – Podfile. В нем хранится список зависимостей проекта. Добавление зависимостей происходит добавлением строки вида «pod <имя модуля>». Содержимое Podfile нашего проекта:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'SpeakThru' do
  # Comment the next line if you're not using Swift
  # and don't want to use dynamic frameworks
  use_frameworks!

  # Pods for SpeakThru
        pod 'Firebase/Core'
        pod 'Koloda'
        pod 'Toast-Swift'
    pod 'Firebase/MLVision'
        pod 'Firebase/MLVisionTextModel'
        pod 'TesseractOCRiOS'
```

```
    target 'SpeakThruTests' do
    inherit! :search_paths
    # Pods for testing
  end

end
```

После формирования списка зависимостей необходимо выполнить в корневой директории проекта команду pod install. Произойдет установка всех зависимостей и будет создан файл с расширением .xcworkspace, с которым теперь следует вести всю дальнейшую работу.

5.2 Библиотека Koloda

Разберем подробнее использованные библиотеки. Первая из них – Koloda. Это библиотека, имплементирующая специально представление KolodaView, которое повторяет поведение колоды карт – смахиваниями можно снимать представления друг за другом, открывая новые. Для корректной работы данного представления необходимо реализовать методы его делегата и источника данных. Объектом, реализующим оба этих протокола, будет контроллер экрана сохраненных переводов. Пример кода, отвечающий за имплементацию методов делегата:

```
extension STBookmarksVC: KolodaViewDelegate {
    func kolodaDidRunOutOfCards(_ koloda: KolodaView) {
        kolodaView.reloadData()
    }

    func koloda(_ koloda: KolodaView, didSelectCardAt index: Int) {
        return
    }
}
```

5.3 Библиотека Toast-Swift

Еще одной библиотекой, связанной с UI, является Toast-Swift. Это библиотека для показа представления с текстом в виде уведомления, написанная на языке Swift. Для данного представления можно задать общий стиль, имеется статический метод со множеством редактируемых параметров для конфигурации и показа самого представления.

5.4 Библиотеки компьютерного зрения Firebase, Tesseract, CoreML Vision

Для реализации объекта, отвечающего за распознавание и сравнение итоговой модели с реализованными, было решено использовать модели от Firebase [4] и Tesseract [5]. Обе библиотеки довольно мощные, Firebase является еще и Backend as Service продуктом с онлайн базой данных, авторизацией пользователей, уведомлениями, обычными базами данных и файловыми хранилищами. Из машинного обучения также представлен широкий арсенал – это механизмы для чат-ботов, выделение объектов и тематик на фотографии, распознавание языка и текста и т.д. Tesseract же предназначен исключительно для задач компьютерного зрения.

Рассмотрим реализацию решения задачи распознавания с помощью Firebase. После установки требуемых зависимостей через CocoaPods, необходимо создать экземпляр класса распознавателя:

```
let vision = Vision.vision()
let textRecognizer = vision.cloudTextRecognizer()
```

Этот объект принимает на вход изображения в специальном формате VisionImage (который умеет конструироваться из стандартного UIImage) и возвращает результат распознавания или ошибку. Причем распознанный текст возвращается блоками – производится так же сегментация текста.

Установка и использование Tesseract происходят немного сложнее. После установки соответствующего модуля в CocoaPods, необходимо добавить данные алфавита в формате tesseract. Эти данные хранятся на GitHub проекта. Для увеличения точности данной модели необходимы фотографии с текстом, занимающим почти всю площадь. Получение таких изображений из произвольного в данной ситуации помогает решить стандартная библиотека iOS называемая Vision. Эта библиотека была представлена на всемирной конференции разработчиков Apple в 2017 году. Она умеет выделять текст на изображениях, распознавать лица, следить за объектами и много другое. С помощью нее удобно подготовить исходное изображение. Для этого необходимо создать запрос и объект, обрабатывающий результаты запроса:

```
let handler = VNImageRequestHandler(
  cgImage: cgImage,
  orientation: inferOrientation(image: image),
  options: [VNImageOption: Any]()
)

let request = VNDetectTextRectanglesRequest(
completionHandler: { [weak self] request, error in
  DispatchQueue.main.async {
```

```
    self?.handle(image: image, request: request, error: error)
  }
})
```

В общем случае работы с моделями распознавания нам не важны цепочки преобразований и алгоритмы, которые используют модели, поэтому общая сущность модели распознавания скрыта протоколом, а класс, ему удовлетворяющий, содержит все три модели, включая ту, что написана самостоятельно.

5.5 Работа с Google Translate API

Последней внешней зависимостью, используемой в приложении, является API переводчика Google Translate. Для небольшого количества запросов к API предоставляется бесплатный доступ, для которого, тем не менее, нужно получить ключ. За работу с API будет отвечать класс STGoogleTranslator с одним публичным методом translate. Внутри него – создание сессии, заполнение параметров, отправка запроса и парсинг результата. В качестве примера – часть реализации метода translate:

```
let task = session.dataTask(with: urlRequest) { (data, response, error) in
  guard let data = data,
  let response = response as? HTTPURLResponse,
  (200 ..< 300) ~= response.statusCode,
  error == nil else {
    completion(nil, error)
    return
  }

  guard let object =
  (try? JSONSerialization.jsonObject(with: data)) as? [String: Any],
  let d = object["data"] as? [String: Any],
  let translations = d["translations"] as? [[String: String]],
  let translation = translations.first,
  let translatedText = translation["translatedText"] else {
     completion(nil, error)
     return
  }

   completion(translatedText, nil)
}
```

Архитектура и обучение модели распознавания

6.1 Описание архитектуры модели

В качестве основной модели для обучения нейронной сети была выбрана модель Image OCR [6] с функцией потерь CTC. Данная модель имеет несложную структуру, но при этом высокую точность и быстрое время работы. Один из примеров использования этой модели в большом продукте – распознавание документов в Dropbox. Использование CTC функции потерь позволяет абстрагироваться от длины текста во время обучения и применения модели, при этом существует ограничение сверху на число распознаваемых символов, о чем подробнее будет рассказываться далее.

Верхнеуровневое описание архитектуры сети представлено на следующей схеме:

Рис. 12. Визуальное описание архитектуры нейронной сети.

На вход сети подается изображение, затем оно проходит через конволюционную нейронную сеть, которая извлекает из него признаки. Затем эти признаки попадают в рекуррентную нейронную сеть, все выходы которой обрабатывает декодирующий алгоритм, результатом которого является строка. Более подробно можно разобрать на примере с фотографией слова “apple”:

Конволюционная нейронная сеть изменяет размер входящей фотографии и количество каналов, получая тензор размерности 4*8*4. Затем, каждый слой этого тензора (по горизонтали слева направо) растягивается в вектор и получается матрица размерности 16*8. Каждый столбец этой матрицы подается на вход рекуррентной нейронной сети lstm (long short-term memory), таким образом учитываются буквы, написанные непосредственно перед данной рассматриваемой. Фактически неявно происходит обучение свойствам языка. Выходы этих нейронов идут на полносвязный слой и слой softmax, откуда получаются вероятности каждой буквы на данном срезе изображения. В примере на картинке мощность алфавита равна 6, это символы

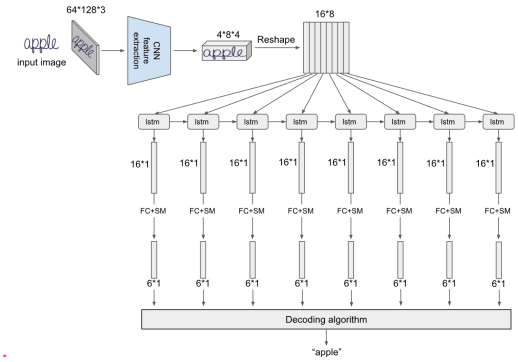

Рис. 13. Пример работы архитектуры.

‘a’, ‘e’, ‘l’, ‘p’, ‘z’, ‘-’. Соответственно, в векторе выхода каждый элемент показывает вероятность соответствующей буквы из алфавита находиться на этом месте фотографии. Как можно заметить, в алфавите присутствует специальный символ ‘-‘. При построении систем подобного рода он используется всегда. Его необходимость заключается в технической необходимости в процессе обучения, связанной с использованием CTC функции потерь. В иных случаях (более богатый алфавит, обучение распознаванию сразу с нескольких языков) размер алфавита может быть изменен, так же, как и число секторов, на которые делится исходная фотография. В больших системах, применяемых в продуктовых решениях, количество секторов может быть 32, 64 и больше. В данной работе фотография разбивалась на 128 секторов. Таким же образом варьируется размерность тензора, получаемого из нейронной сети, которая вычленяет признаки. Его глубину, то есть число каналов, можно свободно варьировать. Итоговые вектора вероятностей попадают на вход декодирующему алгоритму. Его задача – преобразовать их в строку, равную слову на фотографии. Есть несколько подходов и способов реализации декодирующего алгоритма. Самый распространенный – «жадный алгоритм». Он заключается в том, что из каждого вектора вероятности извлекается символ с наибольшей вероятностью.

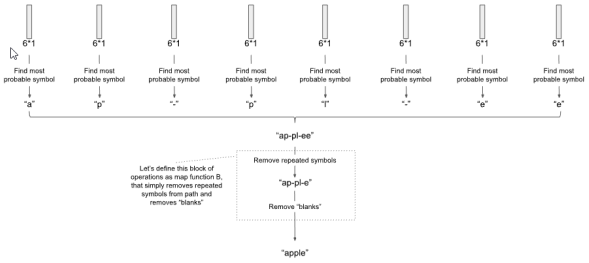

Рис. 14. Алгоритм декодирования.

Они конкатинируются наивным образом. В результате получается строка “ap-pl-ee”. Далее, внутри каждого среза (срезом будем называть последовательность символов между разделителями ‘-‘) нужно произвести удаление дубликатов и оставить только одно вхождение. Подобное происходит потому, что одна буква слова могла попасть в несколько секторов разбиения исходного изображения. После этого шага

мы получаем строку “ap-pl-e”. Затем происходит удаление символов разделителей. На выходе получаем строку “apple”.

6.2 Обучение нейронной сети, подсчет CTC функции потерь

Мы разобрались в архитектуре сети и способе получения строки из ее выходов. Теперь нужно понять, как такую нейронную сеть обучать. В этом вопросе помогает уже упомянутая функция потерь CTC [7]. Сложность обучения и специфика именно CTC функции заключается в том, что размерность входа сети и ее выхода не совпадают и не существует однозначного соответствия между ними.

Чтобы разобраться в работе CTC функции потерь, необходимо ввести некоторые понятия. После выбора декодирующим алгоритмом мы получаем последовательность символов алфавита. Такую последовательность будем называть путем. У каждого пути легко посчитать вероятность – это произведение вероятностей каждого символа. Функция B занимается отображением пути в строку (например, по «жадному алгоритму», описанному выше). Из каждого набора векторов вероятностей можно извлечь много путей:

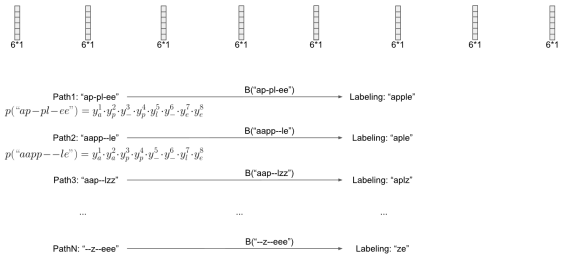

Рис. 15. Пути декодирования.

Из разных путей можно после применения функции B получить одну и ту же строку. Вероятность получения этой строки будем считать как сумму вероятностей путей, из которых она получается.

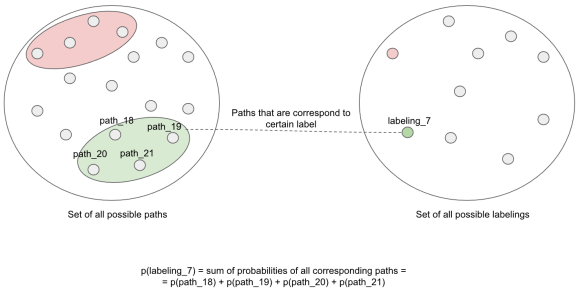

Рис. 16. Визуальное представление вероятностей путей.

Итоговая формула функции потерь очень проста и напоминает функцию кросс-энтропии. $CTCLoss = -ln(p(''apple''))$. В общем случае вместо “apple” соответственно подставляется слово, указанное на изображении. За простотой формулы при этом скрывается одна проблема – это общее количество путей. При мощности алфавита, равной шести, и текущем примере существует $6^8 = 1679616$ возможны путей. При увеличении размера алфавита и количества секторов разбиения, это число становится огромным и непригодным для расчетов. Вся сложность расчета заключается именно в эффективной схеме расчета. К счастью, в работе на тему CTC функции потерь, такая схема приведена. Такая система расчета основывается на динамическом программировании. Рассмотрим один из путей получения слова “apple”:

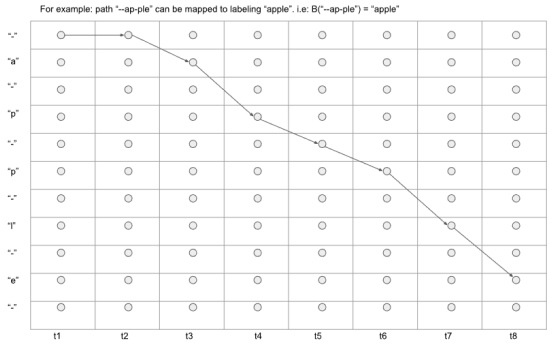

Рис. 17. Пример пути для слова "apple".

Какие переходы возможны в данной таблице? Не должно существовать стрелок вверх, так как нельзя предсказать предыдущий символ на следующем шаге последовательности. Все возможные пути начинаются в верхних двух ячейках. Валидным переходом назовем такой переход, из которого в итоге можно корректно дойти до пути, из которого получится правильный лейбл. Таким образом, для верхней левой ячейки валидны пути лишь в первые две клетки второго столбца. Для второй ячейки первого столбца валидны переходы лишь в 2-4 клетки второго столбца. Рассматривая все валидные переходы далее, можно построить граф всех возможных путей.

Далее рассмотрим величину $\alpha_t(s)$ – суммарная вероятность всех подпутей, префикс которых заканчивается в s-ой позиции на момент времени t. Такую величину как раз возможно посчитать с помощью динамического программирования. Она

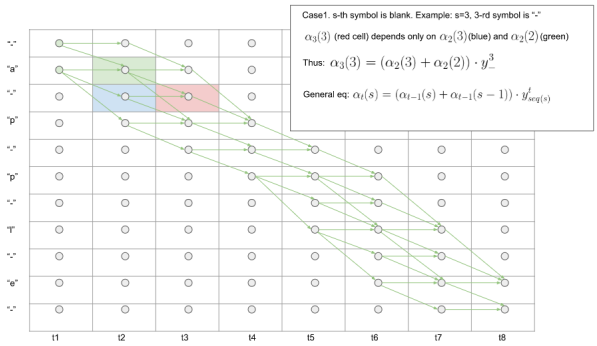

Рис. 18. Все допустимые пути для слова "apple".

равна произведению вероятности текущей буквы в данном секторе на сумму вероятностей в префиксах, из которых можно дойти до данной ячейки. В итоге, получаем $CTCLoss = -ln(p(''apple'')) = -ln(\alpha_8(10) + \alpha_8(11))$.

6.3 Генерация обучающей выборки

Для обучения модели требуется большое количество изображений с текстом и знанием того, какой текст на них изображен. Для такого типа данных целесообразнее не собирать готовые изображения и размечать их, а генерировать изображения самостоятельно. Для этого замечательно подходит библиотека cairocffi языка Python. Она позволяет создавать изображения с произвольным контентом. В нашем случае это будет текст. Шрифт текста мы будем выбирать случайно из пяти, размер фиксируем, чтобы сэкономить время обучения. С ростом номера эпохи усложняем структуру и содержание изображения: начиная с третьей эпохи двигаем текст на изображении вертикально на случайную величину, затем с шестой эпохи начинаем использовать разные шрифты, с девятой эпохи немного вращаем текст и добавляем шум на изображение.

Портирование модели в формат CoreML

7.1 Библиотека coremltools

После завершения обучения нейронной сети встает вопрос о том, как портировать ее в формат, пригодный для использования на мобильном устройстве. Для таких задач компания Apple создала библиотеку coremltools. Данная библиотека позволяет экспортировать готовые модели в формат .mlmodel, который воспринимается на мобильном устройстве на платформе iOS. Источником для конвертации могут служить модели, созданные с помощью библиотек Keras, Xgboost, scikit-learn или libSVM. В нашем случае для обучения нейронной сети использовалась библиотека Keras.

7.2 Ограничения на конвертируемую модель

Непосредственно перед конвертацией стоит учитывать некоторые ограничения библиотеки coremltools в связке с источником – моделью, созданной с помощью Keras. Во-первых, стандартно coremltools не умеет экспортировать Lambda-слои, что вполне логично, ведь внутри Lambda-слоя может быть какая-угодно функция, которая не обязана быть реализуема на языке Swift. В архитектуре нашей сети есть один такой слой – слой для подсчета CTC функции потерь. К счастью, для итогового использования этот слой не нужен, так как на вход декодирующему алгоритму подаются выходы Softmax-слоя. Таким образом, у обученной модели мы откидываем последний Lambda-слой, не нарушая при этом ее функциональность. Тем не менее, существуют способы конвертации Lambda слоя с помощью coremltools [8]. Такой слой придется реализовать на языке Swift, назвав класс, имплементирующий данный слой так же, как и слой в модели.

Еще одно ограничение, накладываемое библиотекой coremltools – отсутствие поддержки конкатенации тензоров с выходов слоев. В архитектуре сети предполагается конкатенация в одном месте – это конкатенация выходных тензоров двух GRU слоев. Один из слоев обучается с параметром обратного обучения, то есть входная последовательность проходит в обратном направлении, а затем возвращается перевернутая последовательность. Второй слой обучается стандартно, остальные настройки сло-

ев совпадают. Было решено проверить результат при отбрасывании одного из этих слоев. Гипотеза заключалась в том, что так как архитектурно эти слои не различаются, то это не должно фатально сказаться на итоговой точности модели. В связи с этим, в модели остался только GRU_B-слой (с переворотом последовательности). После обучения и тестирования результаты показали, что на точность модели данное изменение архитектуры не повлияло.

7.3 Процесс конвертации

В итоге после изменения архитектуры модели и ее обучения, она была сохранена на диск в формате ".h5", стандартном для сохранения моделей, реализованных с помощью Keras. Затем данная модель загружается с произвольной функцией потерь вместо Lambda-слоя, подойдет, например, функция потерь MSE:

```
model = load_model(
    "keras_ocr/model24.h5",
    custom_objects={'<lambda>': keras.losses.mse}
)
```

У этой модели, как было обговорено выше, удаляется слой, считающий функцию потерь:

```
cutted_model = Model(
        model.layers[0].input,
        model.get_layer("softmax").output
)
```

Затем происходит сама конвертация. Для конвертации нужно подать на вход саму модель, задать ее автора, тип лицензии и краткое описание. Затем функции save подать на вход имя файла, в который мы хотим сохранить модель.

```
import coremltools

coreml_model = coremltools.converters.keras.convert(cutted_model)

coreml_model.author = "klabertants"
coreml_model.license = "Public Domain."
coreml_model.short_description = "OCR with keras (only English)."

coreml_model.save("keras_ocr.mlmodel")
```

Данный файл нужно добавить в проект в XCode, с копированием файла в каталог проекта. После добавления XCode автоматически распознает данную модель, типы ее входов и выходов:

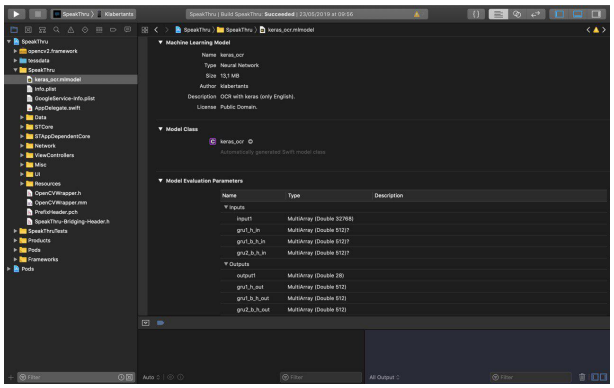

Рис. 19. Данные о модели в XCode.

Предобработка изображений в Swift

8.1 Механизм использования C++ библиотеки OpenCV в Swift

Для предобработки изображения аналогично той, что происходила при обучении нейронной сети в Python, необходима та же библиотека OpenCV.

OpenCV – (Open Computer Vision) — библиотека компьютерного зрения с открытым исходным кодом, предоставляющая набор типов данных и численных алгоритмов для обработки изображений алгоритмами компьютерного зрения.

Существующие реализации OpenCV написаны на языках программирования C++, Java, Python. К сожалению, для языка Swift реализации не предусмотрено, но именно на этом языке реализуется проект приложения. Таким образом, необходим способ использования какой-либо из реализаций в проекте. К счастью, язык Swift происходит от языка Obj-C, который основан на языке C (в проекте на Obj-C свободно компилируется код, написанный на C). Один из возможных способов использовать Obj-C и C++ в одном проекте — полностью разделить их, позволив взаимодействовать через чистый C. Таким образом, можно будет предотвратить их «смешение». Выглядеть это будет так: код, использующий библиотеку C++ переносится в .cpp файл, интерфейс объявлен в заголовочном файле C, C++ часть реализует этот интерфейс с помощью extern «C» функций, а код, в котором будет происходить обращение к интерфейсу C — чистый Objective-C (.m). Но сегодня фактически весь Objective-C компилируется с помощью GCC или clang. Оба компилятора поддерживают Objective-C++, а это означает, что существует более удобный способ смешать языки [9]. Далее в работе мы будем использовать возможность компиляции языка Objective-C++ (*.mm и *.hh файлы) современными компиляторами GCC и clang. Для начала необходимо загрузить фреймворк opencv2.framework с официального сайта и добавить в проект: в Build Phases основного таргета проекта необходимо добавить скачанный фреймворк. Для корректной работы, возможно, потребуется добавление фреймворков из списка (аналогичным методом):

- AssetsLibrary
- CoreGraphics
- CoreMedia

- CoreFoundation
- Accelerate

Затем компилятору необходимо указать в флагах путь к библиотеке. В Build Phases -> Framework Search Paths основного таргета проекта.

Следующим шагом нужно создать класс-обертку. Его основная цель – объявление методов, которые мы хотим использовать в Swift и их реализация, использующая методы библиотеки OpenCV из C++. На языке Objective-C создадим класс OpenCVWrapper. Пока опустив его реализацию, перейдем к этапу создания Bridging Header – заголовочного файла, содержашего иные заголовочные файлы, в которых содержатся методы, реализованные на языках Objective-C или Objective-C++, которые затем можно будет использовать в языке Swift. В сам Bridging Header добавим заголовочный файл нашего класса-обертки:

```
#import "OpenCVWrapper.h"
```

Файл, содержащий реализацию класса-обертки, переименуем из OpenCVWrapper.m в OpenCVWrapper.mm. Таким образом, компилятор теперь понимает, что в реализации данного файла может быть использован C++ код. Добавим заголовочный файл с реализацией библиотеки OpenCV:

```
#import <opencv2/opencv.hpp>
```

На данном этапе XCode выдаст ошибку о том, что ему необходим файл-префикс для заголовочных файлов. Создадим его с названием PrefixHeader.pch и поместим в него включение библиотеки:

```
#ifdef __cplusplus
#include <opencv2/opencv.hpp>
#endif
```

Теперь в Build Settings -> Prefix Header основного таргета проекта следует добавить путь к только что созданному файлу.

На данный момент все необходимые настройки закончены, и можно протестировать работу в Swift. Для теста распечатаем версию библиотеки. Объявим в заголовочном файле OpenCVWrapper.h метод, возвращающий строку, описывающую версию библиотеки:

```
(NSString *)openCVVersionString;
```

Затем реализуем данный метод в файле OpenCVWrapper.mm:

```
+ (NSString *)openCVVersionString {
    return [NSString stringWithFormat:@"OpenCV Version %s",  CV_VERSION];
}
```

Для теста вызовем его на этапе инициализации в STApp. Как результат – в окне вывода видим строку с текущей версией используемой библиотеки OpenCV.

8.2 Предобработка изображения

Настройка и проверка закончены, теперь нужно реализовать такой же алгоритм подготовки изображения, который был использован при обучении нейронной сети, для того, чтобы мы могли подать его результат на вход этой сети. В Python изображение перед отправкой в нейронную сеть проходило следующие этапы обработки:

- Изменение размера
- Изменение палитры на черно-белую

Из-за специфики проекта и разработки под мобильную платформу, к этой цепочке добавится еще два этапа – конвертация изображения из формата UIImage в формат cv::Mat, с которым можно работать в OpenCV, и превращение изображения в числовую матрицу. реализация данного метода есть в официальной документации на сайте OpenCV.

Метод, изменяющий размер изображения, реализован в языке Swift стандартными средствами. Остальные этапы конвертации реализуем в методе prepareForML:

```
+ (NSArray< NSArray<NSNumber*>* > *)prepareForML:(UIImage *)image {
    cv::Mat srcMat = [OpenCVWrapper cvMatFromUIImage:image];

    int rows = srcMat.rows;
    int cols = srcMat.cols;

    cv::Mat grayMat(rows, cols, CV_8UC1);

    cv::cvtColor(srcMat, grayMat, cv::COLOR_BGR2GRAY);

    NSMutableArray *result = [NSMutableArray arrayWithCapacity:rows];

    for (int i = 0; i < rows; i++) {
        NSMutableArray* row = [NSMutableArray arrayWithCapacity:cols];
        for (int j = 0; j < cols; j++) {
            double val = grayMat.at<uchar>(i,j) / 255.;
            [row addObject:[NSNumber numberWithDouble:val]];
        }
        [result addObject:row];
    }

    return result;
}
```

В итоге, для каждого изображения, текст с которого мы будем распознавать, мы будем применять статическую функцию нашего класса-обертки, которая вернет необходимую для входа нейронной сети вещественнозначную матрицу. Обработка изображения той же библиотекой, что и при обучении сети, гарантирует совпадение значений итоговых матриц в Python и в Swift.

Тестирование

9.1 Сравнительное тестирование моделей распознавания

Для сравнительного анализа всех трех моделей распознавания (Firebase, Tesseract и собственной) сначала были произведены общие тесты. Описание тестовых кейсов приведено в таблице:

Номер теста	Надпись	Шрифт	Размер шрифта
1	Lemon grass jasmine	Times new Roman	40
2	Test for the testing	Arial Bold	40
3	This is multiline test	Verdana	40
4	This is tricky test	Times new Roman	20, 32, 40
5	It is also tricky test	Times new Roman, Calibri, Tahoma, Impact	28

Таблица 1. Описание тестовых кейсов первого этапа тестирования

В данном наборе покрыты необходимые и краевые случаи (разные размеры, разные шрифты, надписи в несколько строчек). Результаты тестирования моделей представлены в таблице ниже:

	1	2	3	4	5
Firebase	Полное соответствие	Полное соответствие	Полное соответствие	Полное соответствие	Полное соответствие
Tesseract	Полное соответствие	Полное соответствие	Практически отсутствует соответствие	Частичное соответствие	Полное соответствие
Собственная	Частичное соответствие	Частичное соответствие	Практически отсутствует соответствие	Частичное соответствие	Частичное соответствие

Таблица 2. Результаты первого этапа тестирования

Как видно, Firebase модель распознала все тексты, Tesseract хуже справился с тестом из нескольких строк, а собственная модель уловила лишь текст с пропуском некоторых букв.

Такие результаты ответов модели могли получиться по нескольким причинам:

1. Искажение информации на фотографии во время изменения ее размеров (нейронная сеть принимает на вход изображения размером 512x64 пикселей)
2. Переобучение модели
3. Маленькая выборка с точки зрения шрифтов и размеров

Напомню, что модель обучалась на примерах надписей из 5 шрифтов и одного размера (в силу ограниченности ресурсов, так как обучение происходило на личном компьютере).

Проведем дальнейшее тестирование, чтобы выявить проблему подробнее. Следующие тесты будут состоять только из шрифтов, использованных во время обучения и начертаны размером 40.

Номер теста	Надпись	Шрифт
6	courier font	Courier
7	stix font	Stix
8	urw chancery l font	URW Chancery L
9	century schoolbok font	Century Schoolbook
10	freemono font	Freemono font

Таблица 3. Описание тестовых кейсов второго этапа тестирования

Результаты второго этапа тестирования приведены в таблице ниже:

	1	2	3	4	5
Firebase	Полное соответствие	Полное соответствие	Полное соответствие	Полное соответствие	Полное соответствие
Tesseract	Практически отсутствует соответствие	Полное соответствие	Частичное соответствие	Полное соответствие	Практически отсутствует соответствие
Собственная	Полное соответствие	Полное соответствие	Частичное соответствие	Полное соответствие	Полное соответствие

Таблица 4. Результаты второго этапа тестирования

Заключение

10.1 Результаты и выводы

В ходе работы были решены основные поставленные задачи. Проанализирован функционал приложения, на основании чего разработан и реализован интерфейс приложения с помощью инструмента Figma. Рассмотрены различные подходы архитектурного проектирования компонентов мобильного приложения и описана реализованная архитектура.

Были рассмотрены особенности языка Swift и стандартных библиотек Apple, позволяющие реализовать архитектурные паттерны и элементы приложения. Рассмотрен механизм использования C++ кода в проекте, реализованном на языке Swift. Описаны используемые сторонние библиотеки и механизм взаимодействия с ними - система контроля сторонних библиотек CocoaPods.

Была выбрана архитектура нейронной сети для распознавания текста с фотографии и реализована соответствующая модель. Описана архитектура и процесс обучения данной модели. Затем рассмотрен механизм перевода данной модели в формат, пригодный для использования на мобильном устройстве - библиотека coremltools.

Реализовано и протестировано на личном мобильном устройстве итоговое приложение. Результаты тестирования представлены в таблицах предыдущего раздела. По результатам тестирования выявлены недостатки обученной модели (в сравнении с аналогами сторонних разработчиков), в связи с чем поставлены дальнейшие задачи:

1. Дополнительно проанализировать архитектуру и процесс обучения выбранной модели.

2. Выявить слабые места.

3. Исправить недостатки и произвести последующее тестирование с улучшенной моделью.

Литература

[1] Jeffrey Palermo “The Onion Architecture” (2008), URL: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[2] Elina Bessarabova “MVP vs MVC vs MVVM vs VIPER. What is Better For iOS Development?” (2017), URL: https://themindstudios.com/blog/mvp-vs-mvc-vs-mvvm-vs-viper/

[3] Jeff Gilbert, Conrad Stoll “Architecting iOS Apps with VIPER” (2014), URL: https://www.objc.io/issues/13-architecture/viper/

[4] Firebase Docs “Recognize Text in Images with ML Kit on iOS”, URL: https://firebase.google.com/docs/ml-kit/ios/recognize-text

[5] Khoa Pham “Vision in iOS: Text detection and Tesseract recognition” (2018), URL: https://medium.com/flawless-app-stories/vision-in-ios-text-detection-and-tesseract-recognition-26bbcd735d8f

[6] Supervise.ly “How to build a recognition system: best practices” (2017), URL: https://towardsdatascience.com/lecture-on-how-to-build-a-recognition-system-part-1-best-practices-46208e1ae591

[7] Alex Graves, Santiago Fernandez, Faustino Gomez, Jurgen Schmidhuber “Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks” (2006)

[8] Matthijs Hollemans “Custom Layers in Core ML” (2017), URL: https://machinethink.net/blog/coreml-custom-layers/

[9] Yiwei Ni “OpenCV with Swift - step by step” (2017), URL: https://medium.com/@yiweini/opencv-with-swift-step-by-step-c3cc1d1ee5f1