Разработка механизма автоматического перевода текста на фотографии для мобильных устройств

**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. Таким образом и родилась идея данной работы - разработка механизма автоматического перевода текста на фотографии для мобильных устройств. **Целью** данной работы является развитие прикладных навыков в разработке мобильного приложения для платформы iOS 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 UI приложения с учетом интуитивности и получения пользователем положительного UX
2. Разработка архитектуры приложения и взаимодействия его компонент
3. Исследование и применение фреймворков и 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

**Терминология**

UI (user interface) - интерфейс программы, непосредственно с которым взаимодействует пользователь.

UX (user experience) - опыт пользователя, полученный вследствие взаимодействия с программой.

**Главы**

1. Разработка UI, макета приложения
2. Разработка архитектуры (Core, StyleKit, Routing, Data…)
3. Разработка приложения
4. Сторонние библиотеки и фреймворки
5. Обучение и использование ML модели
6. Портирование нейронной сети в CoreML model
7. C++ -> Obj-C -> Swift bridging
8. Тестирование
9. Результаты и выводы

**1) Дизайн и разработка UI**

Основная задача при разработке UI 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 (<https://www.figma.com/>) 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

Начальным этапом дизайна является выбор цветовой палитры приложения. В качестве цветовой палитры было решено использовать спокойной 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.



Финальный вариант логотипа

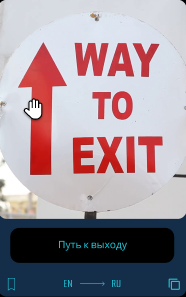
Приветственный экран приложения было решено сделать стандартным образом – логотип посередине, название приложения прибито к нижней части экрана. Для дальнейшей разработки дизайна нужно было понять и сформулировать функциональность приложения, в результате чего был составлен список базовой функциональности:

* 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.



Итоговый макеты карточки перевода

На основном экране должна быть возможность перехода к экрану настроек и экрану сохраненных переводов. Так же должны быть возможность снять фотографию, чтобы инициировать процесс перевода. Для этого в Tab Bar основного экрана добавлены три кнопки, выполняющие соответствующие задачи.



Tab bar 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, надписать 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.



Пример ячейки с основной информацией

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



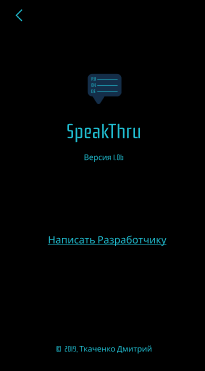
Пример ячейки с основной и дополнительной информацией

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.



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:



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

1. **Разработка архитектуры**
2. **Разработка приложения**
3. **Сторонние библиотеки и фреймворки**
4. **Обучение и использование ML модели**
5. **Портирование нейронной сети в CoreML model**
6. **C++ -> Obj-C -> Swift bridging**

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 проистекает 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 С. Таким образом, можно будет предотвратить их «смешение». Выглядеть это будет так: код, использующий библиотеку С++ переносится в .cpp файл, интерфейс объявлен в заголовочном файле С, С++ часть реализует этот интерфейс с помощью extern «C» функций, а код, в котором будет происходить обращение к интерфейсу С — 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. 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 (тут сайт) 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"

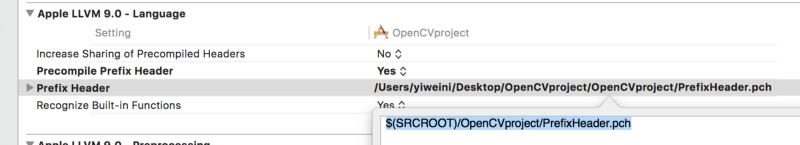
Файл, содержащий реализацию класса-обертки переименуем из OpenCVWrapper.m в OpenCVWrapper.mm. Таким образом компилятор теперь понимает, что в реализации данного файла может быть использован C++ код. Добавим заголовочный файл с реализацией библиотеки OpenCV:

#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**:

print("\(OpenCVWrapper.openCVVersionString())")

Как результат – в окне вывода видим строку с текущей версией используемой библиотеки OpenCV.

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 этап – 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 (<https://docs.opencv.org/2.4/doc/tutorials/ios/image_manipulation/image_manipulation.html>)

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.

1. **Тестирование**
2. **Результаты и выводы**