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

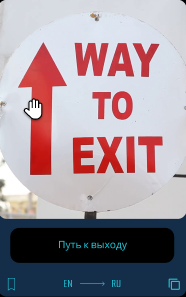
Приветственный экран приложения было решено сделать стандартным образом – логотип посередине, название приложения прибито к нижней части экрана. Для дальнейшей разработки дизайна нужно было понять и сформулировать функциональность приложения, в результате чего был составлен список базовой функциональности:

* 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.



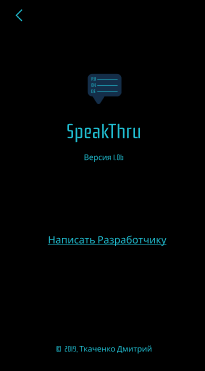
Пример ячейки с основной и дополнительной информацией

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**

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.

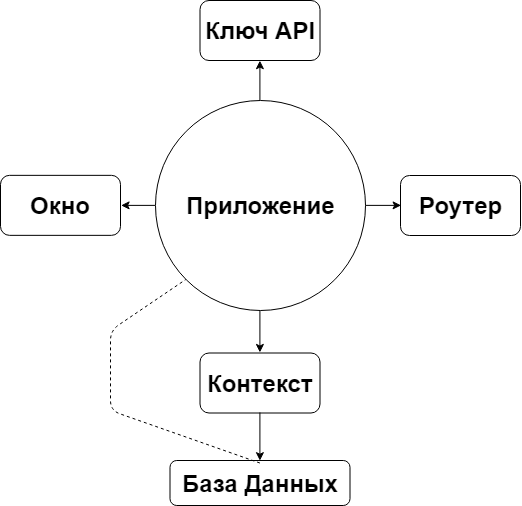
При разработке данного приложения были использованы несколько подходов. Первый из них – разработка по принципу луковой или слоистой архитектуры. Основная ее идея – разделение компонент программы на несколько слоев и непосредственное взаимодействие компонент только внутри одного слоя или соседних слоев. 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 БД 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



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.



Визуальное представление Приложения

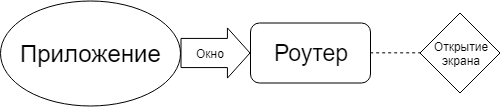
Сущность «Окно», о которой речи выше не было, появилась здесь из платформенных требований. На платформе iOS все представления, что размещены на экране, выстроены в иерархию. Корень этой иерархии – Окно. При старте приложения создается его окно, а затем все представления размещаются в нем. Из-за того, 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).



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.



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.

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:

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 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:

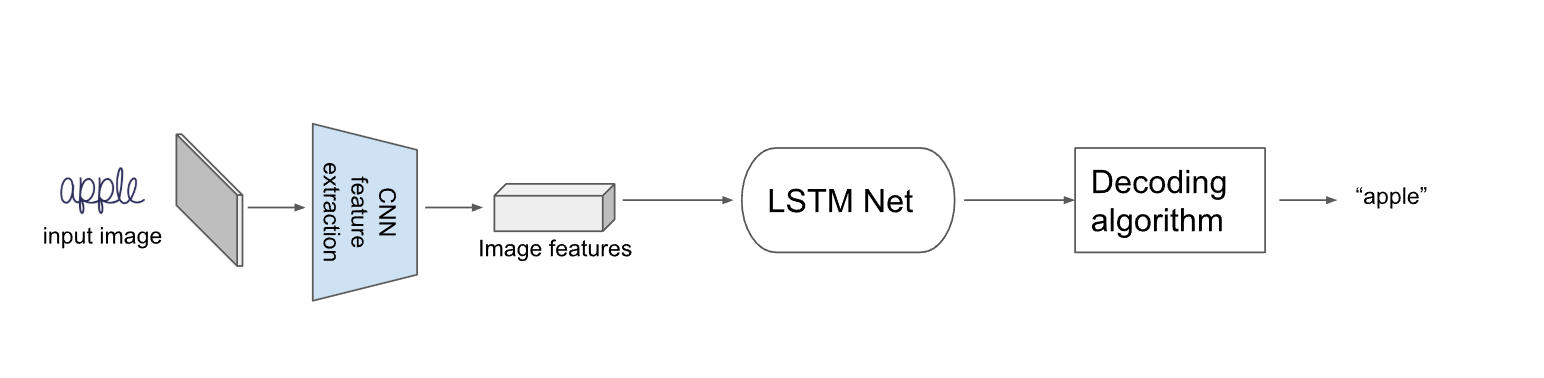
1. Model – создание моделей данных, имплементация бизнес-логики
2. View – интерфейсы представлений, логика отображения и обработка событий от пользователя
3. ViewModel – преобразование данных от Модели для отображения во View, использование событий от View для обновления данных в Модели

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.

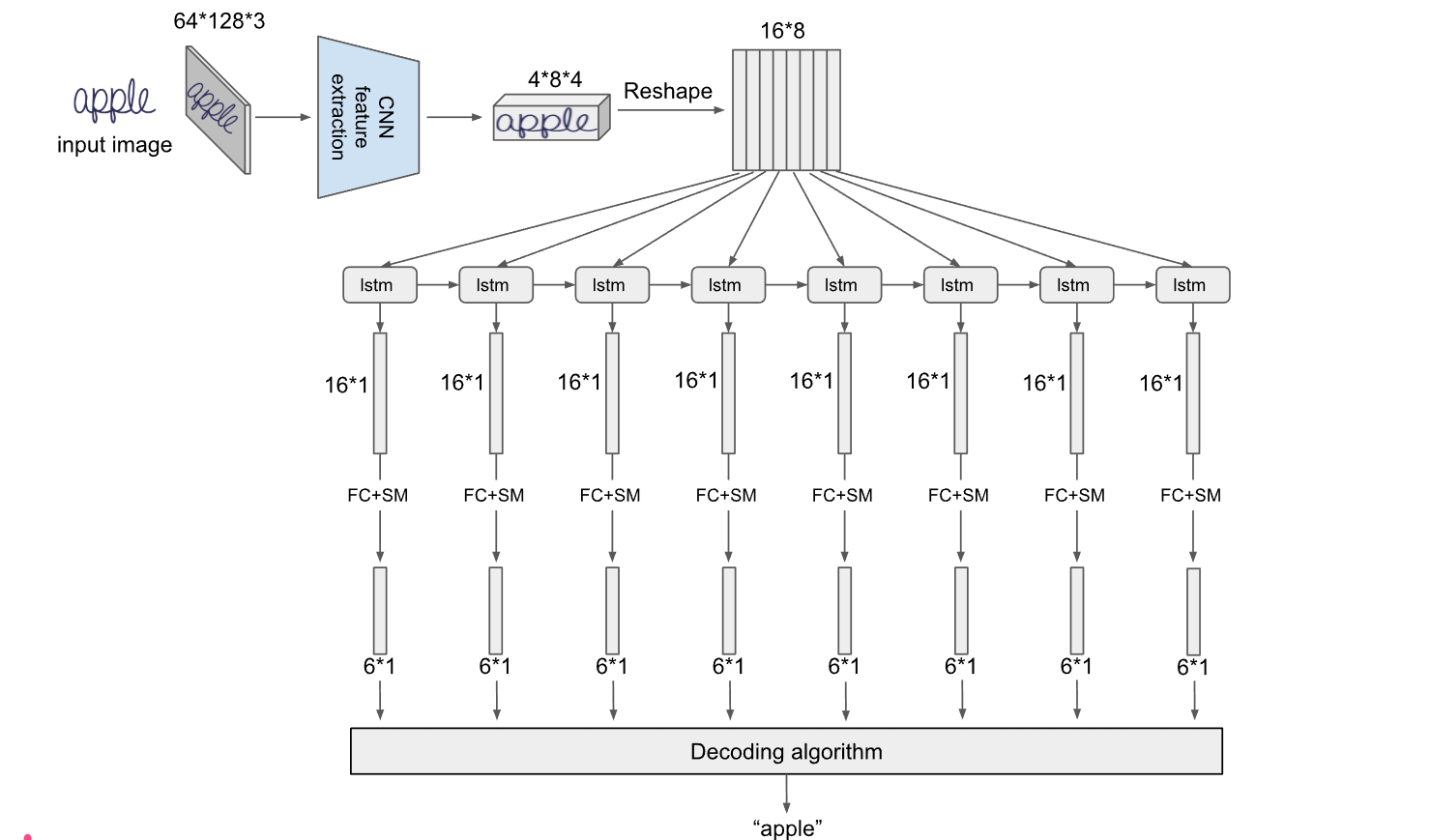
1. **Разработка приложения**
2. **Сторонние библиотеки и фреймворки**
3. **Обучение и использование ML модели**

В качестве основной модели для обучения нейронной сети была выбрана модель Image OCR 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 (<https://blogs.dropbox.com/tech/2017/04/creating-a-modern-ocr-pipeline-using-computer-vision-and-deep-learning/>). 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:



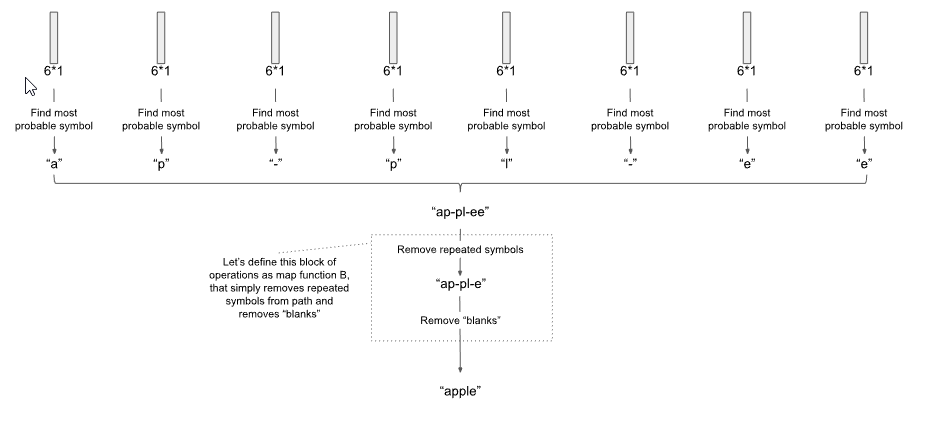
На вход сети подается изображение, затем оно проходит через конволюционную нейронную сеть, которая извлекает из него признаки. Затем эти признаки попадают в рекуррентную нейронную сеть, все выходы которой обрабатывает декодирующий алгоритм, результатом которого является строка. Более подробно можно разобрать 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 {‘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.

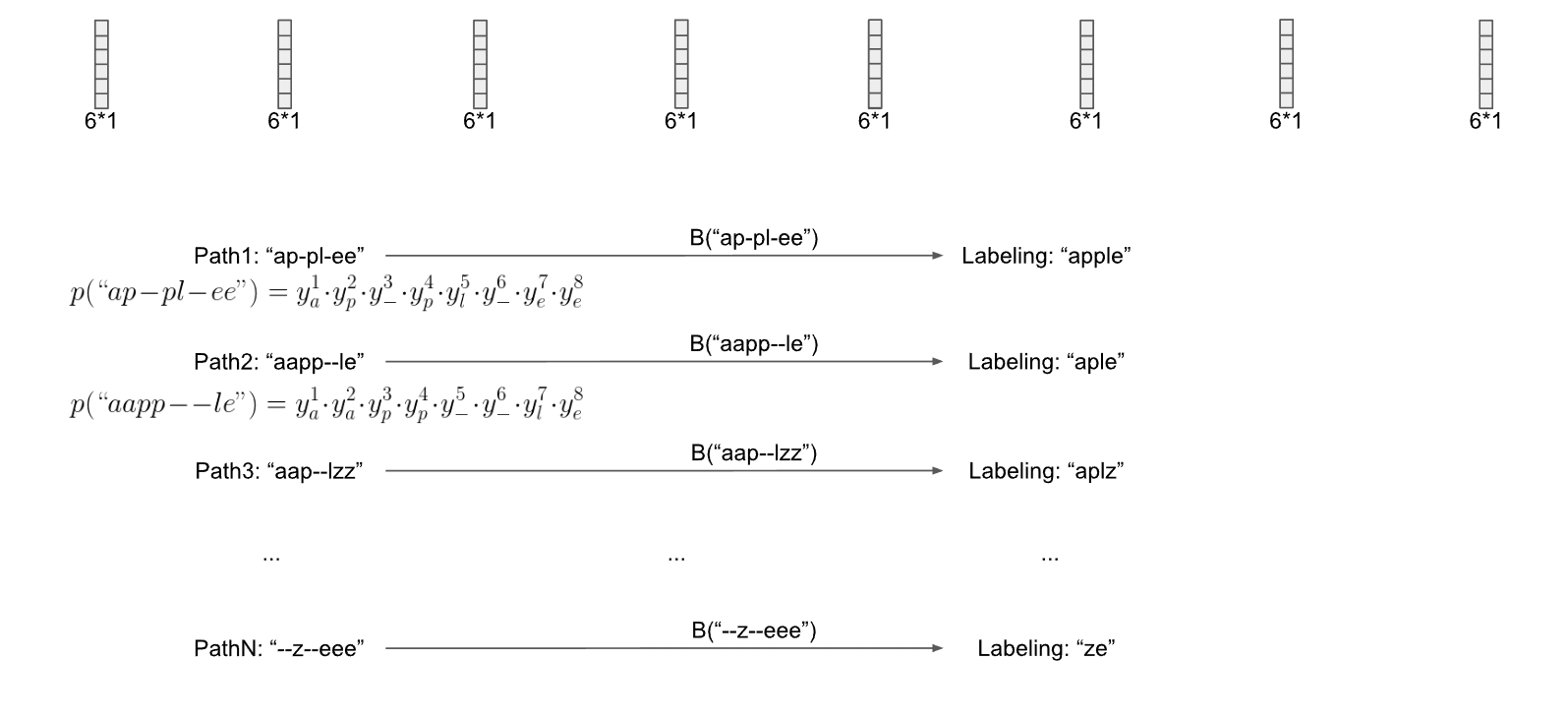
Итоговые вектора вероятностей попадают на вход декодирующему алгоритму. Его задача – преобразовать их в строку, равную слову на фотографии. Есть несколько подходов и способов реализации декодирующего алгоритма. Самый распространенный – «жадный алгоритм». Он заключается в том, что из каждого вектора вероятности извлекается символ, с наибольшей вероятностью.



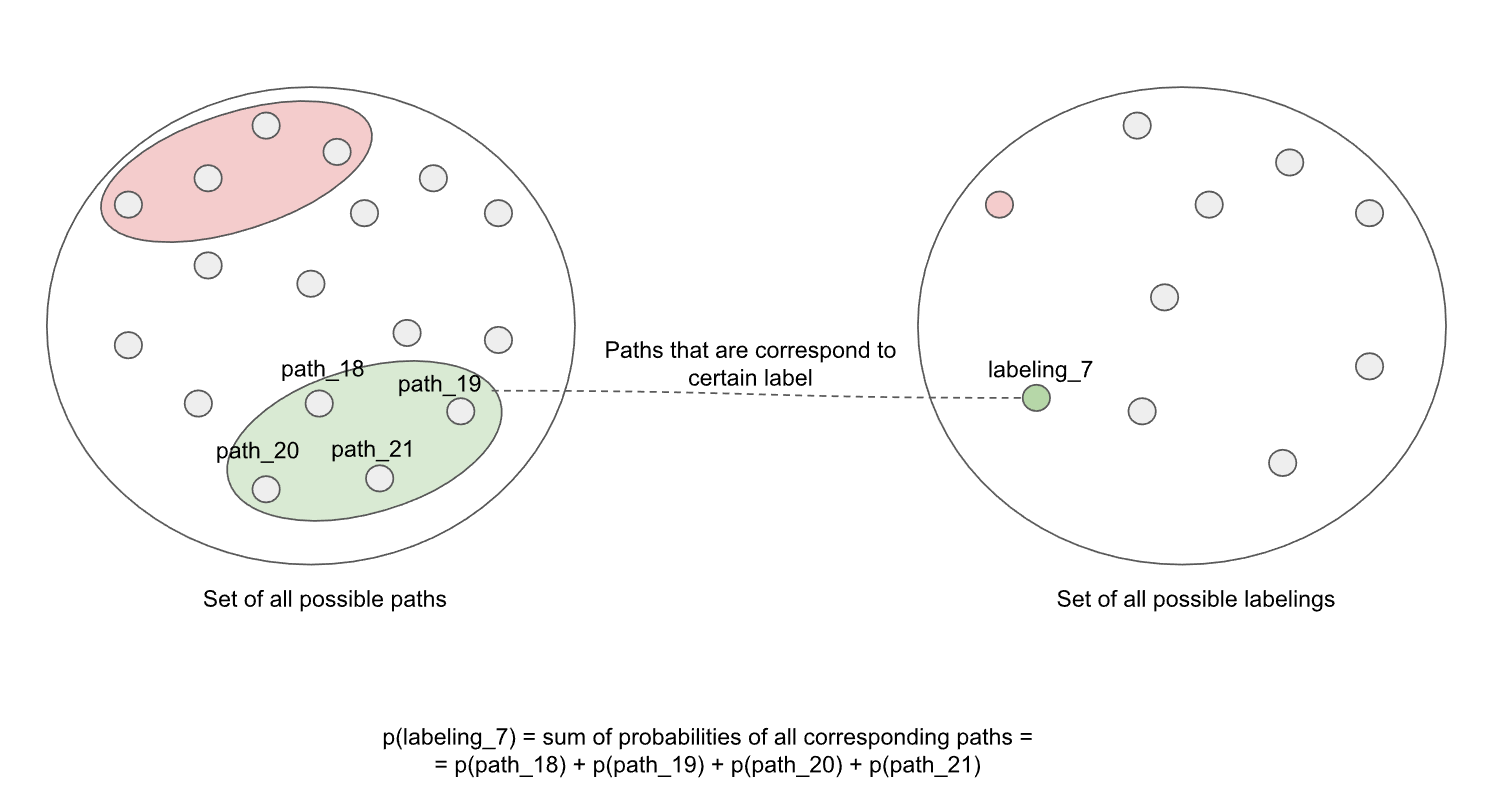
Они конкатинируются наивным образом. В результате, получается строка “ap-pl-ee”. Далее, внутри каждого среза (срезом будем называть последовательность символов между разделителями ‘-‘) нужно произвести удаление дубликатов и оставить только одно вхождение. Подобное происходит потому, что одна буква слова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

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 (<http://www.cs.toronto.edu/~graves/icml_2006.pdf>). 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.

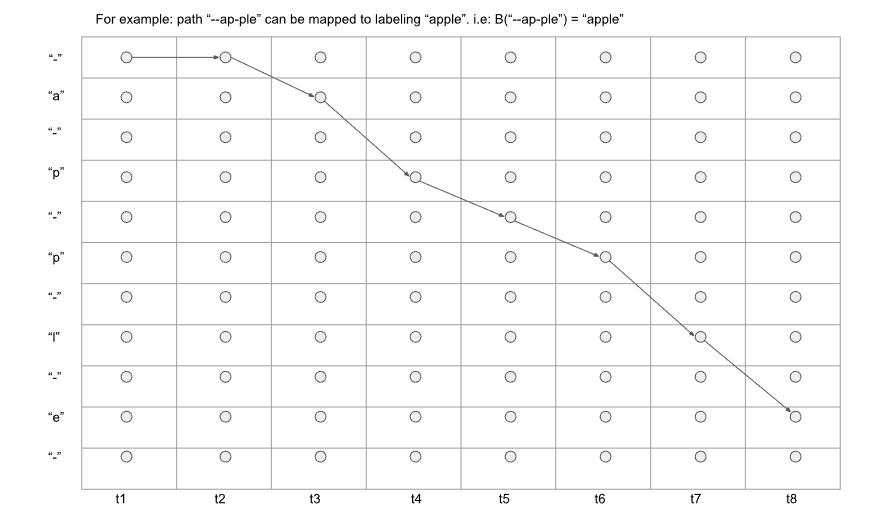
Чтобы разобраться в работе CTC функции потерь, необходимо ввести некоторые понятия. После выбора декодирующим алгоритмом, мы получаем последовательность символов алфавита. Такую последовательность будем называть путем. У каждого пути легко посчитать вероятность – это произведение вероятностей каждого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



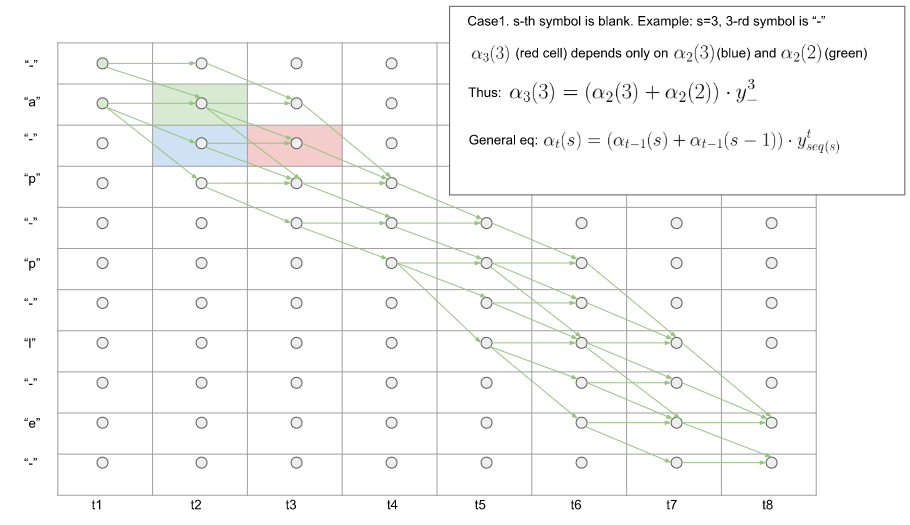
Из разных путей можно после применения функции В получить одну и ту же строку. Вероятность получения этой строки будем считать как сумму вероятностей путей, из которых она получается.



Итоговая формула функции потерь очень проста и напоминает функцию кросс-энтропии. CTC Loss = -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 = 1 679 616 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”:



Какие переходы возможны в данной таблице? Не должно существовать стрелок в верх, 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 αt(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 равна произведению вероятности текущей буквы в данном секторе на сумму вероятностей в префиксах, из которых можно дойти до данной ячейки. В итоге, получаем CTC Loss = -ln(p(“apple”)) = -ln(α8(10) + α8(11)).

1. **Портирование нейронной сети в CoreML model**
2. **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"

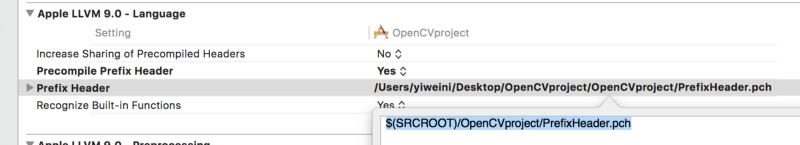
Файл, содержащий реализацию класса-обертки переименуем из OpenCVWrapper.m в OpenCVWrapper.mm. Таким образом компилятор теперь понимает, что в реализации данного файла может быть использован C++ код. Добавим заголовочный файл с реализацией библиотеки OpenCV:

#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**