**3** ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе детально рассматривается, как функционирует программное средство колоризации. Раздел состоит из подразделов, каждый из которых описывает функционирование отдельного модуля программы, а также интерфейс взаимодействия с другими модулями приложения. Сами функциональные блоки также будут различаться между собой в зависимости от их местонахождения.

Стоит заметить, что функциональное проектирование модулей и описание интерфейсов взаимодействия также будет зависеть от языка программирования и в некоторых случаях от используемого фреймворка, что объясняет некоторое различие подразделов данной записки.

**3.1** Структура модуля колоризации

Как было заявлено выше, для построения программного средства колоризации полутоновых изображений необходимо реализовать нейронную сеть, которая и будет обрабатывать изображения. Так как для реализации нейронной сети был выбран фреймворк Caffe, в пояснительной записке будет кратко изложены основные принципы построения нейросетей в данном фреймворке.

В библиотеке Caffe топология нейросетей, исходные данные и способ обучения задаются с помощью конфигурационных файлов в формате prototxt. Топология сетей описывается с помощью слоёв, каждый из которых имеет свой тип, параметры и связь с другими слоями нейронной сети. Ниже приведён пример описания свёрточного слоя в prototxt-файле:

# Объявление слоя

layer {

# Наименование слоя

name: "conv3\_1"

# Тип слоя

type: "Convolution"

# Ссылка на слой, за

# которым необходимо

# создать текущий

bottom: "conv2\_2norm"

top: "conv3\_1"

# Параметры слоя

convolution\_param {

# Размер выхода слоя

num\_output: 256

pad: 1

# Размер ядра свёртки

kernel\_size: 3

}

}

Файл с расширением prototxt для описания процесса обучения в сообществе пользователей Caffe принято называть «solver». Подобные файлы содержат набор обязательных и опциональных параметров для процесса обучения. Таковыми параметрами являются:

* путь к файлу с конфигурацией сети;
* периодичность тестирования во время обучения;
* параметры стохастического градиентного спуска;
* максимальное количество итераций;
* архитектура, на которой будут проводиться вычисления;
* путь для сохранения обученной сети.

Пример solver-файла будет рассмотрен в дальнейшем вместе с указанием параметров для обучения нейросети.

Для запуска обучения свёрточной нейронной сети и обработки входных данных будут использоваться соответственно интерфейс командной строки и Python-интерфейс, описание функционала, которого будет описано в одном из следующих подразделов.

**3.2** Архитектура нейронной сети

Свёрточная нейронная сеть – специальная архитектура искусственных нейронных сетей, предложенная Яном Лекуном и нацеленная на эффективное распознавание изображений. Основной идеей свёрточной нейронной сети является чередование слоёв, выполняющих операцию свёртки со слоями субдискретизации. По определению, свёртка – это математическая операция, применённая к двум функциям, порождающая третью функцию, которая иногда может рассматриваться как модифицированная версия одной из первоначальных. В контексте обработки изображений рассматривается понятие ядра свёртки – матрицы фиксированного размера, содержащей коэффициенты, значения которых перемножаются со значениями каналов пикселей изображения. Данная маска размещается в углу изображения, после чего высчитывается значение свёртки для пикселя в центре маски. Затем маска сдвигается к следующему пикселю, и операция повторяется.

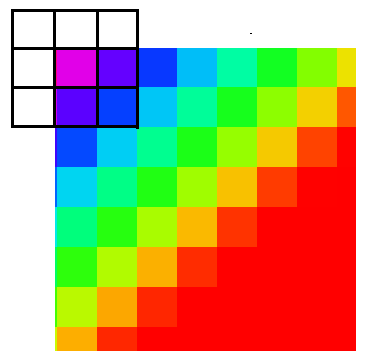


Рисунок 3.1 – Итерация обработки изображения свёрточным фильтром

Слои субдискретизации используются для уменьшения размера, последующих слоёв. Обычно из группы соседних результатов предыдущего слоя свёртки выбирается максимальный, который и становится входом для следуюшего слоя. Вставка данных слоёв увеличивает производительность нейронной сети, при этом незначительно влияя на результат.

Каждый свёрточный слой имеет несколько фильтров, после обработки которыми образуются карты признаков. Так при углублении в сеть размер слоя уменьшается, а количество карт признаков увеличивается. При этом теряется информация о расположении более мелких признаков на изображении (линий, окружностей), и появляется информация о наличии более крупных признаков на изображении (сложных фигур, объектов). Затем следуют несколько полносвязных слоёв, которые обрабатывают информацию о признаках. Подобные сети часто выполняют задачу классификации изображений. Примером такой архитектуры является VGG Convolutional Neural Network.

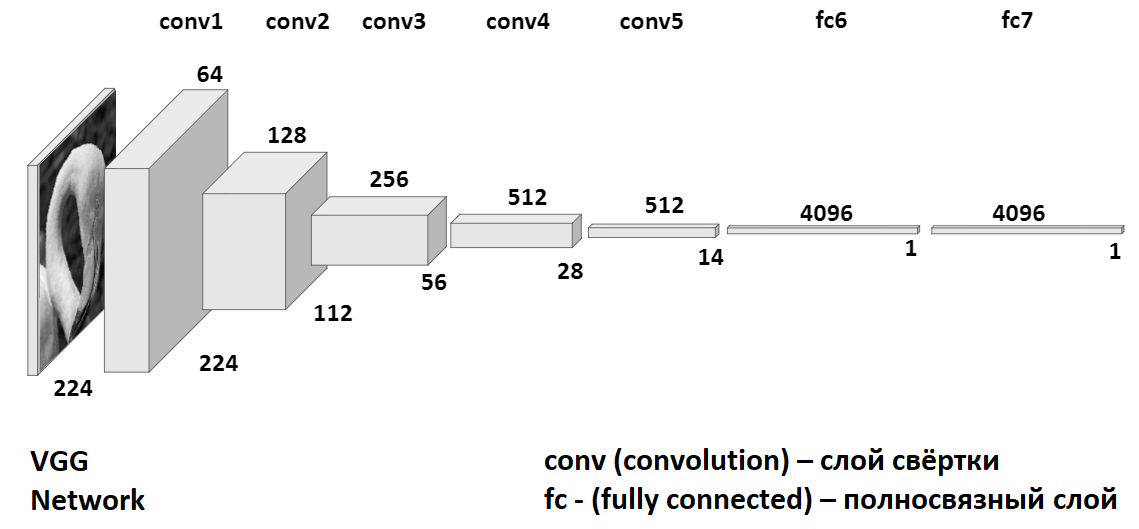


Рисунок 3.2 – Архитектура VGG сети

Для решения задачи колоризации данная архитектура не подходит из-за наличия полносвязных слоёв, при использовании которых информация о расположении признаков на изображении теряется полностью. Также количество нейронов в последнем слое нейронной сети должно превышать или хотя бы быть равным количеству нейронов входного слоя. Исходя из этого следует, что текущую архитектуру необходимо модифицировать под задачу колоризации.

Для начала необходимо определить формат выходных данных. Для этого произведён выбор цветовой модели, в которой будет представлено обработанное изображение. В первую очередь модель должна быть распространённой для упрощения процесса перевода изображений из RGB и обратно. Также выбранная модель должна охватывать достаточно большое цветовое пространство.

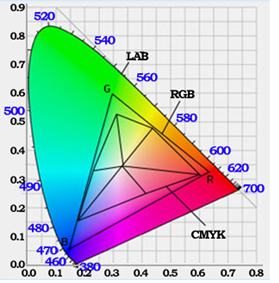


Рисунок 3.3 – Сравнение покрытия цветового пространства разными цветовыми моделями

Подходящей цветовой моделью является LAB. В данной цветовой модели цвет представлен двумя каналами: A и B. Яркость представлена третьим каналом L. В первую очередь данная модель охватывает более широкий диапазон цветового спектра, чем другие популярные цветовые модели (см. рисунок 3.3). Также важным достоинством этой модели является наличие отдельного канала для яркости. При подаче канала яркости L на вход нейросети, необходимо лишь получить значения каналов A и B.

Следующим этапом является квантование LAB пространства, которое покрывается RGB моделью (см рисунок 3.4). Для этого делим данное пространство на квадраты со стороной 10. Количество подобных квадратов равно 313. Каждый из квадратов соответствует одному цвету, который является средним для всех значений цветов в квадрате.

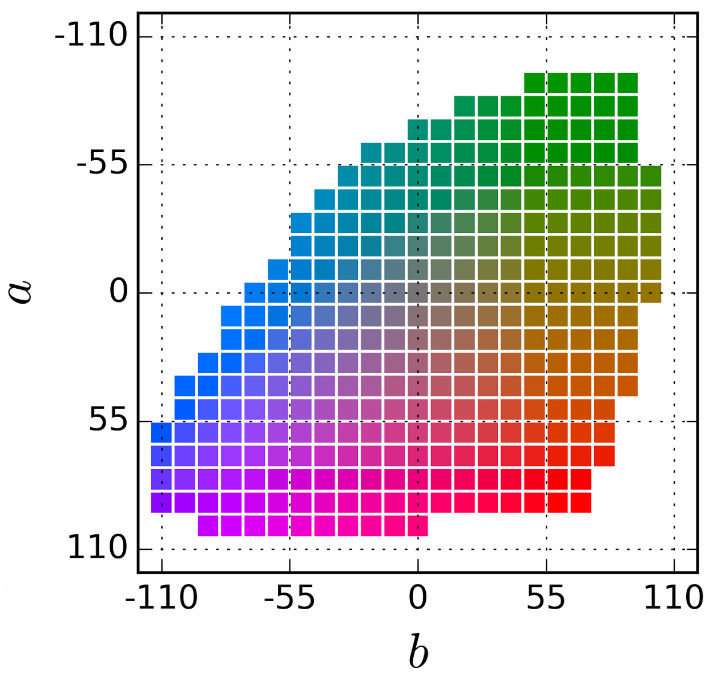


Рисунок 3.4 – Квантование цветового пространства LAB

На основании полученных цветов можно построить слой из 313 карт признаков, каждая из которых будет содержать вероятность того, что группы пикселей будут принимать цвет соответствующей этой карте. Входом для данного слоя будет являться результат последнего слоя свёртки, а на основании результатов группе пикселей будет присваиваться цвет с наибольшей вероятностью для данной группы (см. рисунок 3.5).

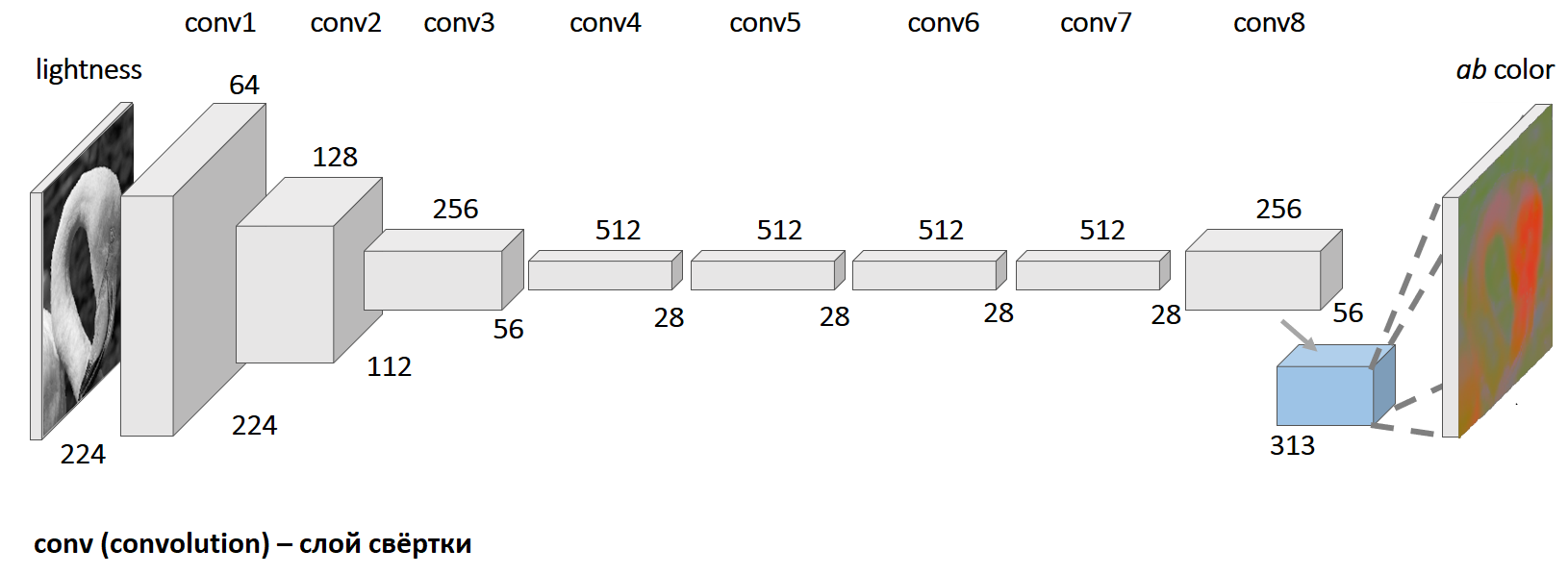


Рисунок 3.5 – Архитектура нейросети для колоризации изображения

Стоит также отметить, что полносвязные слои будут заменены свёрточными слоями с некоторыми модификациями. Реализация этих слоёв будет описана в следующих разделах, а также в приложенном листинге.

**3.3** Модуль запуска нейронной сети

Как было указано ранее, для запуска обработки нейронной сетью входных данных будет использоваться Python-интерфейс библиотеки Caffe. Код для запуска будет размещён в отдельном Python-пакете в виде единого файла colorization.py. Внутри данного модуля находится единственная функция с интерфейсом:

def colorize(id, token)

Аргументы функции:

* id – идентификатор изображения, необходимый для загрузки нужного файла из файловой системы; также данный идентификатор необходим для сохранения результата в файл и оповещения клиента о завершении колоризации конкретного изображения; является строкой;
* token – идентификатор клиента, загрузившего изображение, необходимый для оповещения конкретного клиента о завершении колоризации; также является строкой.

Оба аргумента генерируются в клиентском приложении и передаются серверной части. Формат этих параметров будет рассмотрен позднее при описании взаимодействия между сервером и клиентом.

Основная задачи модуля запуска колоризации состоит в инициализации нейронной сети и передачи ей параметров. Кроме этого данный модуль выполняет следующие функции:

* загрузка исходного файла из соответствующего каталога;
* изменение размера изображения для входного слоя нейронной сети;
* перевод изображения из цветовой модели RGB в LAB и последующее выделение L-канала в отдельное изображение, которое в дальнейшем загружается в сеть;
* изменение размера результата обработки до изначального;
* объединение результата работы нейронной сети с L-каналом, для получения колоризованного изображения;
* сохранение результата в файловую систему;
* вызов модуля, использующего python-gcm, для оповещения пользователя о окончании колоризации изображения.

Некоторые части реализации модуля для запуска нейросети будут представлены в следующем разделе.

**3.4** Модуль сетевого интерфейса серверной части

Для взаимодействия с клиентом используется протокол HTTP. В качестве веб-фреймворка для сервера был выбран Bottle. Чтобы обосновать такой выбор, в данном подразделе будет описан протокол передачи изображений между клиентской и серверной частью.

Исходя из того факта, что колоризация изображения может занимать длительное время, было решено спроектировать протокол таким образом, чтобы инициатором загрузки полутоновых и выгрузки колоризованных изображений являлся клиент. Согласно данному протоколу сервер должен иметь способ оповестить клиента, об окончании процесса колоризации, чтобы клиент смог своевременно получить обработанное изображение. При этом, механизмом для оповещения решено использовать технологию Google Cloud Messaging использующую протокол XMPP. Сам механизм оповещения вынесен в отдельный модуль. Следовательно, основными функциями данного блока являются:

* загрузка полутонового изображения на сервер, его сохранение в файловую систему по запросу пользователя и вызов модуля запуска колоризации изображений;
* загрузка колоризированного изображения из файловой системы и его отправка на клиентское приложение по запросу пользователя.

Интерфейс для поддержки данных функций является достаточно простым для его реализации при помощи Bottle. Bottle реализован в едином файле, потому процесс подключения этого фреймворка крайне прост. Для обработки HTTP запросов создаются функции с соответствующими аннотациями. Рассмотрим интерфейс функции, которая обрабатывает запрос на передачу изображения серверной части:

@post('/load')

def load():

# ...

id = request.get\_header("id")

# ...

Как видно в примере тип обрабатываемого HTTP-запроса обозначен аннотацией. Конкретно данная функция будет обрабатывать POST-запрос. Значение в кавычках объединяется со строкой адреса сервера в полный адрес, на который клиентской части необходимо будет слать запрос. Доступ к заголовкам и телу запроса можно получить при помощи объекта request, который доступен из тела функции.

Интерфейс функции для обработки запроса на получение обработанного изображения:

@get("/get")

def get():

# ...

return img

Как и функция для обработки POST-запроса метод не имеет аргументов, а все необходимые данные можно получить из самого запроса. Также стоит заметить, что данная функция имеет возвращаемое значение, которое в последствии передаётся клиенту через тело ответа от сервера.

Способ передачи изображений и содержание дополнительной информации в заголовке каждого из запросов будет описано в дальнейшем, при описании сетевого интерфейса клиентского приложения.

Для запуска фреймворка необходим вызов функции run:

run(host='172.17.0.2', port=8080, debug=True)

Аргументами этой функции являются:

* host – строка определяющее адрес сервера; в примере указан адрес виртуального Ethernet-интерфейса Docker-контейнера;
* port – порт, по которому будет доступен сервер;
* debug – булевая переменная, определяющая необходимость вывода сервисной информации в STDOUT поток.

Данный модуль является входной точкой для сервера – именно запуск этого модуля приводит серверную часть в рабочее состояние. Весь код для реализации блока размещён в одном файле server.py, который расположен в корневом каталоге проекта.

**3.5** Модуль оповещения

Основной и единственной функцией модуля оповещения является отправление сообщения о завершении процесса колоризации клиентской стороне. Данный модуль использует python-gcm – библиотеку для отправки сообщений через GCM. Отправка сообщения осуществляется через выделенные компанией Google сервера. Для взаимодействия с подобными серверами клиентское приложение и серверная часть должны иметь статичный API-ключи, необходимые для идентификации приложения или его компонентов. Также для отправки сервером GCM-сообщений конкретному клиенту необходим идентификатор клиента (token). Этот идентификатор назначается клиенту сторонним сервером в процессе работы приложения и может изменяться. Процесс получения идентификатора клиента будет описан в дальнейшем.

Для получения API-ключей необходимо зарегистрировать приложение в Google API Console, а затем импортировать его в Firebase console. Важно, чтобы при регистрации данные о корневом пакете Android-приложения совпадали с реальными. После данной процедуры можно получить API-ключи из интерфейса Google API Console (см. рисунок 3.6)

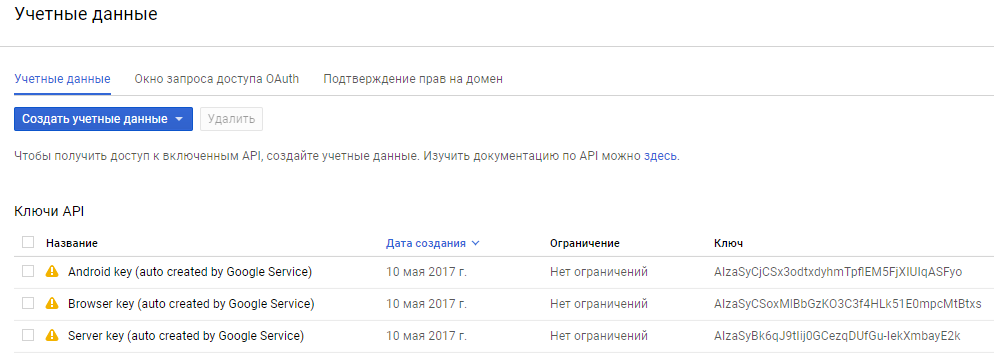


Рисунок 3.6 – Фрагмент веб-страницы Google API Console с учётными данными приложения

Для реализации модуля оповещения необходим API-ключ для сервера. Как было указано выше данный модуль вызывается модулем запуска колоризации. Модуль оповещения передаёт клиенту id обработанного изображения. Входными аргументами для единственного метода данного модуля является id изображения и token клиента:

def notify(id, token)

Код модуля находится в файле notifier.py, который не имеет дополнительных зависимостей и расположен в корневом каталоге проекта. Реализация модуля будет указана в следующем разделе.

**3.6** Иерархия каталогов для хранения изображений.

Для хранения изображений был создан каталог storage. Внутри данного каталога находятся два дочерних: input – для хранения полутоновых изображений; output – для хранения результатов обработки. Для наименования файлов используется id файла. Все изображения хранятся в формате PNG.

**3.7** Создание Docker-образа

За основу для создания собственного образа был взят образ от The Berkeley Vision and Learning Center. Данный образ представляет собой ОС Ubuntu 14 с предустановленным Caffe. Кроме добавления исходного кода проекта, для работы также необходимы некоторые дополнительные библиотеки Python. Кроме этого были установлены некоторые стандартные утилиты для проверки работоспособности сети.

Команда docker pull копирует образ из удалённого репозитория в каталог для хранения образов:

docker pull bvlc/caffe:cpu

Для создания контейнера используется команда docker run. Важно на данном этапе используя Docker NAT связать порт хоста с портом контейнера указанным в server.py:

docker run -p 192.168.0.104:8080:8080 -it bvlc/caffe:cpu

Открытие порта необходимо, чтобы возможно было получить доступ к Docker-контейнеру из сети.

После установки сети и всех модулей. необходимо сохранить изменения в новом образе:

docker commit -m «init commit» containername colorserver

Создание образа в дальнейшем позволит создавать контейнеры содержащие полностью сконфигурированный сервер. Также существует возможность загрузки образа на удалённый репозиторий. Использование Docker для установки готового сервера будет отображено в руководстве пользователя.

**3.8** Пользовательский интерфейс Android-приложения

Интерфейс Android-приложения должен быть удобным в использовании, простым и отражающим весь функционал приложения. Для построения интерфейса были использованы активности (activities).

Activity – основной компонент любого Android приложения. Activity является наследником класса Context, который в свою очередь имеет возможность взаимодействовать с другими компонентами приложения. Также данный класс имеет доступ к ресурсам приложения, которые расположены в каталоге res модуля. К таковым ресурсам относятся различные изображения, а также файлы в формате XML, необходимые для реализации различного функционала: меню, стилей, векторных изображений, разметки экранов.

Для клиентского приложения было решено создать две основные активности приложения:

* Activity для получения изображения с помощью камеры устройства или из локального хранилища; отображения преобразованного в полутон изображения на экран;
* Activity для отображения результата колоризации и взаимодействия с сетью.

Интерфейс обоих экранов будет схож между собой: большую часть экрана будет занимать элемент для отображения изображения. Внизу будет размещены кнопки, которые позволят пользователю управлять приложением. Первый экран, который будет отображён после запуска приложения, предоставит пользователю возможность получить изображение с помощью камеры устройства или из галереи. Третий контроллер будет направлять пользователя на второй экран, за контроль которого отвечает вторая активность. На втором экране будет отображён результат колоризации. Пользователь в свою очередь будет иметь возможность перейти обратно на первый экран, чтобы выбрать другое изображение. Отображение различных сообщений будет реализовано при помощи всплывающих на экран элементов представления.

Активность для получения изображения имеет название PickerActivity и реализована в одноимённом файле java-пакета (package) activities. Так как эта активность является входной точкой для приложения, большинство методов приложения являются приватными и их интерфейс не будет описан в данном разделе. Остальные методы необходимы для переопределения методов базового класса и реализации java-интерфейса. Одним из таких методов является onCreate:

@Override  
protected void onCreate(Bundle savedInstanceState)

Данный метод вызывается системой в момент создания сущности активности. Его функциями являются создание элементов представления из разметки, связь этих элементов с ссылками и назначение сущности активности обработчиком нажатий на контроллеры. Единственный аргумент необходим для восстановления состояния активности при смене конфигурации (изменение ориентации экрана, смена языка телефона и т. д.).

Для того, чтобы сущность активности могла обрабатывать нажатия на элементы представления, данный класс должен реализовывать интерфейс View.OnClickListener, являющийся частью стандартной библиотеки:

@Override  
public void onClick(View v)

Данный метод вызывается системой при нажатии на элемент отображения. В аргументы передаётся сам элемент, чтобы определить какое действие необходимо выполнить. При нажатии на кнопки для получения изображения будет создаваться объект намерения (Intent), благодаря которому система может найти стороннюю активность, которая выполнит описанное в намерении действие и вернёт результат. За создание намерения для получения фотографии с помощью камеры будет отвечать отдельный модуль. При нажатии на контроллер, отвечающий за переход на следующий экран, в систему будет отправляться намерение, открывающее следующую активность.

@Override  
protected void onActivityResult(int requestCode,

int resultCode, Intent data)

Данный метод вызывается в момент, когда сторонняя активность возвращает результат работы. Первый аргумент необходим для определения того, какая из сторонних активностей вернула результат. Код запроса передаётся вместе с намерением в систему в момент вызова сторонней активности. Второй аргумент является кодом результата. Перед дальнейшими действиями необходимо проверить чтобы код результата был равен -1. Третий аргумент – намерение с результатом, созданное сторонней активностью. Результатом является заключённая в намерении URI, с помощью которой можно получить несжатое изображение. При получении изображения оно обрабатывается, сохраняется в приватное поле активности и выводится на экран.

Активность для просмотра результата колоризации имеет имя ColorizationActivity и её код расположен в одноимённом файле в том же package, что и предыдущая активность. Данная активность также не имеет внешнего интерфейса, так как даже её создание происходит при помощи системного вызова.

Методы onCreate и onClick также переопределены в этой активности и имеют тот же интерфейс, что и PickerActivity. В методе onCreate настраиваются элементы представления схожим с предыдущей активностью способом. Нажатие на единственную кнопку вызывает отмену текущей операции загрузки изображения и дальнейшее возвращение в активность для выбора изображения. Кроме определения обработчика нажатий для кнопки в методе onCreate происходит регистрация объекта BroadcastReceiver, который принимает намерения от системы с определённым фильтром. В данном случае он принимает намерения от сервиса, который обрабатывает входящие GCM-сообщения. Такая процедура необходима, чтобы активность своевременно отображала обработанное изображение на экран. После регистрации ресивера происходит асинхронная загрузка изображения на сервер при помощи объектов AsyncTask, использование которых будет описано в дальнейшем.

Кроме того, приватными не являются следующие методы:

@Override  
protected void onDestroy()

// ...

@Override  
public void onBackPressed()

Оба метода вызываются самой системой. Первый метод служит для выполнения каких-либо действий при уничтожении сущности текущей активности. В конкретном случае в данном методе происходит отмена регистрации ресивера, который был зарегистрирован в методе onCreate. Второй метод вызывается в момент, когда пользователь нажимает кнопку «Назад» на своём устройстве. В коде метода происходит отмена текущей задачи загрузки изображения на сервер или с сервера, как и по нажатию на контроллер, расположенный на экране. Класс PushHandler, расширяющий класс BroadcastReceiver, который обрабатывает намерения от GCM-сервиса, расположен внутри активности и является приватным статическим классом. Такое его расположение обусловлено тем фактом, что класс нигде кроме этой активности не используется.

**3.9** Разметка пользовательского интерфейса

Каждая из активностей приложения создаёт элементы представления из файлов разметки. Разметка хранится в виде XML-файла в папке /res/layout. Это сделано для того, чтобы отделить код от дизайна, как это принято во многих технологиях (HTML и CSS). Кроме основной компоновки для всего экрана, существуют дочерние элементы компоновки для группы элементов. По сути, компоновка – это некий визуальный шаблон для пользовательского интерфейса приложения, который позволяет управлять элементами управления, их свойствами и расположением. Кроме того, элементы представления можно создавать программным способом, однако данный способ создания разметки в данной работе использован не будет. Каждый файл разметки должен содержать только один корневой элемент компоновки, который должен быть объектом View или ViewGroup. Внутри корневого элемента существует возможность добавлять дополнительные объекты разметки или виджеты как дочерние элементы, чтобы постепенно формировать иерархию элементов, которую определяет создаваемая разметка. Если обращаться к элементам управления через Java-код, то необходимо присваивать элементам уникальный идентификатор через атрибут android:id. Сам идентификатор назначается через выражение @+id/example. После этого появляется возможность обращаться к элементу через код при помощи метода findViewById(R.id.example). В качестве примера рассмотрим разметку элемента для отображения изображения на экране:

<ImageView  
 android:id="@+id/iv"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 />

Атрибуты layout\_width и layout\_height необходимы для определения ширины и высоты элемента представления. Значение match\_parent означает, что высота и ширина элемента должны быть ограничены только размерами корневого элемента. Связь ссылки на элемент представления в активности происходит следующим образом:

private ImageView imageView;

// ...

imageView = (ImageView) findViewById(R.id.iv);

**3.10** Модуль предоставления доступа к камере устройства

Код предоставляющий доступ к камере устройства будет выделен в отдельный модуль. Данное решение будет применено, так как предоставление доступа к камере устройства реализуется созданием намерения, требующего заключения в себя значений, создание которых является излишним при каждом создании подобного намерения. Иными словами, данный модуль хранит эти необходимые значения внутри себя, чтобы не возникала необходимость их инициализации при каждом вызове метода для получения намерения. Код модуля заключен в файле CameraProvider.java и содержит одноимённый класс. Сам файл размещён внутри пакета utils. В качестве внешнего интерфейса модуль предоставляет метод getCameraIntent:

public static Pair<Intent, Uri> getCameraIntent(

Context context)

Этот метод предоставляет пару значений: намерение на запуск сторонней активности, использующей камеру устройства и URI, по которой в последствии можно получить изображение.

**3.11** Модуль предобработки изображения на клиентском приложении

Код модуля предобработки заключён в файле ImageProcessor.java, который содержит одноимённый класс. В качестве внешнего интерфейса класс предоставляет два статических метода, принимающих оригинальное изображение в виде Bitmap-объекта и предоставляющих обработанный объект такого же типа. Первый метод необходим для получения полутонового изображения и имеет следующий интерфейс:

public static Bitmap getGrayScale(Bitmap original)

Второй метод необходим для получения изображения с меньшим разрешением, чем у оригинала. Интерфейс данного метода:

public static Bitmap resize(Bitmap original, float scale)

Дополнительный параметр этого метода scale определяет множитель для размеров обработанного изображения.

Файл с кодом данного модуля размещён в пакете utils проекта.

**3.12** Выполнение задач, требующих больших временных затрат

Библиотека Android предоставляет класс AsyncTask, для вынесения требующих сравнительно больших временных затрат задач в отдельный поток. К таким задачам в текущем приложении относятся работа с сетью и предобработка изображения.

В качестве решения задачи было создано три абстрактных класса, переопределяющих класс AsyncTask. GetBitmapTask необходим для выполнения предобработки изображения в отдельном потоке. Данный класс переопределяет метод doInBackground базового класса:

protected final Throwable doInBackground(Object[] params)

Этот метод заключает в себе операции, выполнение которых происходит в новом потоке. Директива final добавлена, чтобы запретить дальнейшее переопределение метода. Возвращаемым значением является объект Throwable – ошибка, которая возникла в процессе работы метода. При отсутствии ошибки возвращаемым значением является null. Аргумент params необходим для передачи нужных методу данных, однако в дипломной работе подобный способ передачи данных не используется. Вместо этого данные для асинхронного выполнения работы передаются в конструктор:

protected GetBitmapTask(Context context, Uri uri)

Аргументами конструктора являются:

* context – объект Context, который вызвал данную задачу;
* uri – URI, для получения объекта Bitmap.

Для использования задачи можно создать анонимный класс, переопределяющий лишь метод onPostExecute и затем запустить задачу вызовом метода execute. Для получения объекта Bitmap класс предоставляет метод getBitmap:

@Override  
protected void onPostExecute(Throwable tr) {

// ...  
 if (tr == null) {  
 PickerActivity.this.bitmap = getBitmap();  
 // ...  
 } // else show error message  
}

Весь код класса заключён в одноименном файле, который является частью пакета utils.

Использование классов UploadingTask и DownloadingTask не сильно отличается от предыдущего.

protected DownloadingTask(String id)

DownloadingTask предназначен для отправки запроса о получении обработанного изображения с конкретным идентификатором id, декодирования скачанного с сервера изображения из строки формата Base64 в Bitmap.

protected UploadingTask(Bitmap bitmap)

UploadingTask предназначен для получения строки формата Base64 из предобработанного Bitmap-объекта и дальнейшая его отправка на сервер. Данный класс не имеет метода getBitmap в отличие от предыдущих, однако имеет методы для кодировки изображения и получения случайного идентификатора. Эти методы не будут описаны в данном разделе, так как являются приватными.

Оба класса являются частью пакета api, так как предназначены для работы с сетью.

**3.13** Модуль предоставления интерфейса сервера.

Модуль предоставления интерфейса сервера является надстройкой над фреймворком Retrofit 2. Класс API, код которого заключён в одноимённом файле содержит единственный внешний метод getInterface предоставляющий доступ к интерфейсу сервера:

public static ServerInterface getInterface()

Внутри статического метода заключена «ленивая» инициализация объекта, реализующего интерфейс ServerInterface, который будет описан ниже. Сам объект строится фреймворком на основе определённого java-интерфейса с использованием механизма чтения аннотаций.

ServerInterface имеет два метода, которые описывают необходимый для взаимодействия с сервером интерфейс:

@POST("load")  
Call<String> loadImage(@Header("token") String token, @Header("id") String id, @Body String base64);  
  
@GET("get")  
Call<String> getImage(@Header("id") String id);

Как и при использовании Bottle используются аннотации @POST и @GET c указанием части адреса, которая будет объединена с адресом сервера, на который будет послан запрос. Аннотация @Header используется для указания фреймворку, что аргумент при создании запроса необходимо поместить в заголовок c указанным в аннотации ключом. Аннотация @Body аналогична предыдущей. Отличие состоит в том, что фреймворк при создании HTTP-запроса разместит значение в его теле. Возвращаемый объект типа Call<String> предоставляет доступ к блокирующему и неблокирующему способу отправки запроса, а также указывает на тип возвращаемого в теле HTTP-ответа значения. В данном проекте используется только блокирующая поток отправка запроса при помощи метода execute. Данный метод возвращает объект Response, который в свою очередь имеет метод body для получения тела ответа.

Метод loadImage принимает следующие аргументы:

* token – идентификатор пользователя, необходимый серверу для отправки GCM-уведомления;
* id – идентификатор изображения;
* base64 – изображение закодированное в строку формата Base64.

Метод getImage принимает единственный аргумент id – идентификатор изображения.

Класс API как и интерфейс ServerInterface являются частью пакета api.

**3.14** GCM-сервис и оповещение активностей об окончании колоризации

Service является компонентом приложения, который может выполнять длительные операции в фоновом режиме и не содержит пользовательского интерфейса. Другой компонент приложения может запустить службу, которая продолжит работу в фоновом режиме даже в том случае, когда пользователь перейдет в другое приложение. Firebase расширяет данный компонент в абстрактном классе FirebaseMessagingService.