Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

К ЗАЩИТЕ ДОПУСТИТЬ

Зав. каф. ЭВМ

\_\_\_\_\_\_\_\_\_\_\_ Д.И. Самаль

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к дипломному проекту

на тему

ПРОГРАММНОЕ СРЕДСТВО КОЛОРИЗАЦИИ ПОЛУТОНОВЫХ ИЗОБРАЖЕНИЙ

БГУИР ДП 1–40 02 01 01 196 ПЗ

Студент Е. А. Шилов

Руководитель Я. A. Бурый

Консультанты:

от кафедры ЭВМ Я. A. Бурый

по экономической части В.И. Шкода

Нормоконтролер А.С. Сидорович

Рецензент

МИНСК 2017

РЕФЕРАТ

СОДЕРЖАНИЕ

Оглавление

Элементы оглавления не найдены.

Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет: ФКСиС. Кафедра: ЭВМ.

Специальность: 40 02 01 «Вычислительные машины, системы и сети».

Специализация: 40 02 01 01 «Проектирование и применение локальных компьютерных сетей».

УТВЕРЖДАЮ

Заведующий кафедрой ЭВМ

\_\_\_\_\_\_\_\_\_ Д. И. Самаль

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_ 20\_\_ г.

ЗАДАНИЕ

по дипломному проекту студента

Шилова Евгения Александровича

**1** Тема проекта: «Программное средство колоризации полутоновых изображений» - утверждена приказом по университету от 13 апреля 2017 г. № 627-с.

**2** Срок сдачи студентом законченного проекта: 1 июня 2017 г.

**3** Исходные данные к проекту:

**3.1** Операционные системы: Linux, Android.

**3.2** Среды разработки: PyCharm, Android Studio

**3.3** Языки программирования: Python, Java.

**4** Содержание пояснительной записки (перечень подлежащих разработке

вопросов):

Введение 1. Обзор литературы. 2. Системное проектирование.   
3. Функциональное проектирование. 4. Разработка программных модулей.   
5. Программа и методика испытаний. 6. Руководство пользователя. 7. Экономическая часть. Заключение. Список используемых источников. Приложения.

**5** Перечень графического материала (с точным указанием обязательных чертежей):

**5.1** Программное средство колоризации полутоновых изображений. Плакат.

**5.2** Программное средство колоризации полутоновых изображений. Схема структурная.

**5.3** Программное средство колоризации полутоновых изображений. Диаграмма классов.

**5.4** Программное средство колоризации полутоновых изображений. Схема программы.

**5.5** Программное средство колоризации полутоновых изображений. Диаграмма последовательности.

**5.6** Заключительный плакат. Плакат.

**6** Содержание задания по экономической части: «Технико-экономическое

обоснование разработки программного средства колоризации полутоновых изображений».

ЗАДАНИЕ ВЫДАЛА В. И. Шкода

КАЛЕНДАРНЫЙ ПЛАН

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование этапов  дипломного проекта | Объем  этапа,  % | Срок выполнения этапа | Примечания |
| Подбор и изучение литературы | 20 | 27.03 – 02.04 |  |
| Структурное проектирование | 20 | 03.04 – 09.04 |  |
| Функциональное проектирование | 10 | 10.04 – 16.04 |  |
| Разработка программных модулей | 10 | 17.04 – 23.04 |  |
| Программа и методика испытаний | 10 | 24.04 – 30.04 |  |
| Расчет экономической эффективности | 20 | 01.05 – 14.05 |  |
| Оформление пояснительной записки | 10 | 18.05 – 28.05 |  |

Дата выдачи задания: 01.02.2017

Руководитель Я. А. Бурый

ЗАДАНИЕ ПРИНЯЛ К ИСПОЛНЕНИЮ \_\_\_\_\_\_\_\_\_

ВВЕДЕНИЕ

В современном мире вычислительные системы всё чаще внедряются в процесс производства и бытовую жизнь человека. Одной из причин этому является стремление к автоматизации огромного количества различных задач и процессов в разных сферах жизнедеятельности.

Некоторые из подобных задач автоматизации достаточно проблематично решить стандартными способами. Чтобы описать и разработать точный алгоритм необходимо уйма времени и средств. Примером такой задачи может быть классификация объектов. Для решения этой и многих других задач, которые требуют аналитических вычислений подобных тем, что делает человеческий мозг, используются нейронные сети.

Рассмотрим подробнее задачу колоризации полутоновых изображений. Речь идёт о генерации трёхканального цветного изображения исходя из информации об интенсивности каждого пикселя одноканального изображения.

Колоризация достаточно часто применяется в кинематографе. Самой первой технологией колоризации стала ручная раскраска киноплёнки анилиновыми красителями, применявшаяся уже в конце XIX века. Ручная раскраска фильмов применялась вплоть до 1930-х годов, когда существовали технологии цветного кино в натуральных цветах, для создания специальных эффектов или при реставрации старого чёрно-белого кино. По мере совершенствования компьютерных технологий работа упростилась: студии стали окрашивать фильмы, отмечая цвет каждого объекта в кадре, затем реализованный алгоритм окрашивал уже фрагментированное изображение нужным цветом. С прорывом в сфере изучения нейронных сетей стало возможным переложить задачу колоризации с человека на компьютер. При удачной попытке получения нейронной сети, способной обмануть стороннего наблюдателя, можно построить автоматизированный процесс, способный реставрировать достаточно большие архивы старых кинолент.

Целью данного дипломного проекта является попытка реализации веб-сервиса для колоризации полутоновых изображений, создание минимально необходимого программного интерфейса серверной части, а также пример реализации клиентского приложения. Упор делается на разработку готового сервиса, который в дальнейшем можно будет с минимальными усилиями разместить на серверной машине. Также особое внимание уделяется простоте разработки нейронной сети, выполняющей работу по колоризации изображений.

Предполагается разработка следующего функционала:

* загрузка полутонового изображения на сервер;
* колоризация полутонового изображения серверной частью;
* оповещение сервером клиента о конце обработки изображения;
* загрузка обработанного изображения на клиентскую часть;
* возможность просмотра обработанного сообщения.

Для достижения поставленных целей требуется выполнение следующих задач:

* разработка клиентской части приложения, что предполагает выбор платформы, для которой будет реализовано клиентское приложение, разработку пользовательского интерфейса и реализацию общения с сервером;
* разработка серверной части приложения для обслуживания запросов пользователей, обработки изображений, хранения обработанных изображений и оповещения пользователей о завершении операции обработки;
* разработка интерфейса и протокола сообщения между клиентской и серверной частью.

После проведенного анализа доступных средств и методов разработки, для выполнения поставленной задачи было решено использовать Python версии 2.7 в качестве языка для написания серверной части и Android фреймворк языка Java версии 1.7 для реализации клиентской части. Разработку упростит фреймворк Caffe, позволяющий упростить работу по построению архитектуры нейронной сети, а затем и обучить сеть. Целевыми ОС являются UNIX системы. В ходе работы над проектом будет использовано программное обеспечение компании JetBrains. Такой выбор обусловлен, наличием сред разработки под необходимые для написания проекта языки, простотой интерфейса, а также опытом работы с данными продуктами компании.

**1** ОБЗОР ЛИТЕРАТУРЫ

**1.1** Обзор аналогов

Для того, чтобы обосновать идею создания программного средства колоризации в данном подразделе будет рассмотрено готовое ПО, позволяющее колоризировать изображения.

Первым рассмотренным программным обеспечением является AKVIS Coloriage. AKVIS Coloriage – программа для раскрашивания черно-белых фотографий и замены цвета на цветных изображениях. Основная задача – разделение областей и выбор цветов. Согласно заявлению разработчиков, достаточно несколькими штрихами обозначить объекты для перекрашивания и запустить расчет. Coloriage распознает границы и раскрасит изображение, накладывая цвета с учетом текстур, освещенности и теней.



Рисунок 1.1 – Интерфейс ПО AKVIS Colorage

Программным обеспечением Colorage можно пользоваться как в виде отдельного продукта, так и в качестве плагина к Adobe Photoshop. Несмотря на заявление разработчиков об автоматической колоризации изображений, программе необходимо управление пользователя, который должен как выделить области, так и назначить цвета этим областям.

Несмотря на этот недостаток, достоинствами данного ПО являются удобный интерфейс для выделения областей и хороший инструмент для выбора цвета. Программа Coloriage поддерживает форматы JPEG, TIFF, BMP, PNG. AKVIS Colorage является ПО, распространяемым с помощью лицензии. однако предоставляется возможность пробного периода [1].

Adobe Photoshop является самым распространённым способом колоризации изображений. Данный факт объясняется широким распространением данного продукта. Adobe Photoshop является многофункциональным графическим редактором, разработанным и распространяемым фирмой Adobe Systems. В основном работает с растровыми изображениями, однако имеет некоторые векторные инструменты. Продукт является лидером рынка в области коммерческих средств редактирования растровых изображений и наиболее известным продуктом фирмы Adobe.



Рисунок 1.2 – Интерфейс ПО Adobe Photoshop

Несмотря на то, что изначально программа была разработана как редактор изображений для полиграфии, в данное время она широко используется и в веб-дизайне.

Несмотря на распространение данное программное обеспечение не является самым эффективным способом колоризации. Причиной тому является тот факт, что приложение не предоставляет никаких способов автоматической колоризации, а лишь предоставляет широкий выбор инструментов для изменения цвета объектов [2].

Ни одно ПО, рассмотренное ранее, не подразумевает полностью автоматической колоризации. В процессе поиска было найдено веб-приложение, демонстрирующее возможности искусственного интеллекта в решении задачи колоризации [3].

Веб-приложение является частью открытой платформы Algorithmia, где разработчики могут разместить какой-либо алгоритм, предоставив API для использования другими разработчиками. Конкретное приложение предоставляет минимальный интерфейс, где пользователь может загрузить изображение либо предоставить ссылку на него.

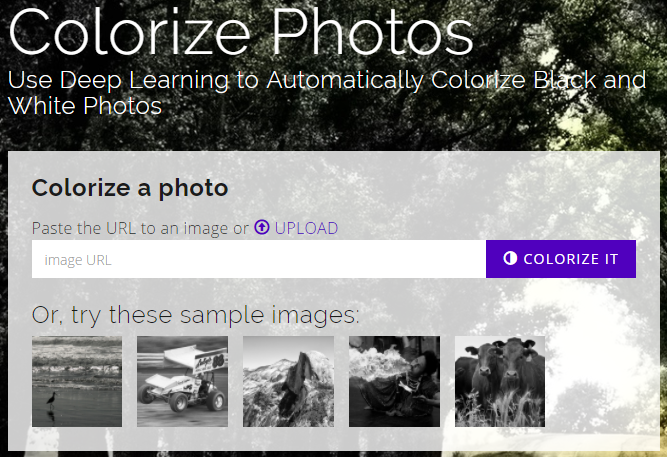


Рисунок 1.3 – Фрагмент веб-страницы с интерфейсом для колоризации

Также платформа предоставляет API для использования алгоритма разработчиками. Пример использования API на языке программирования Python:

import Algorithmia

input = {"image": "\_IMAGE\_URL\_"}

client = Algorithmia.client('\_API\_KEY\_')

algo = client.algo('deeplearning/ColorfulImageColorization/1.0.1')

# prints {"output": "data://.algo/temp/image.png"}

print algo.pipe(input)

Как видно из примера, для использования API необходим ключ, который можно приобрести, используя платформу Algorithmia.

Несмотря на тот факт, что данное ПО предоставляет способ полностью автоматической колоризации изображений, оно не является свободным, а минимальный интерфейс веб-приложение недостаточно удобен.

**1.2** Выбор архитектуры нейронной сети

На первом этапе разработки и проектирования системы была проведена работа по поиску подходящей для цели проекта архитектуры нейронной сети. Для этого был произведён поиск некоего подобия каталога архитектур нейронных сетей. Из найденного каталога, рассмотрим архитектуры сетей, которые спроектированы для обработки изображений. Одним из вариантов таких сетей является свёрточная сеть (рисунок 1.4).

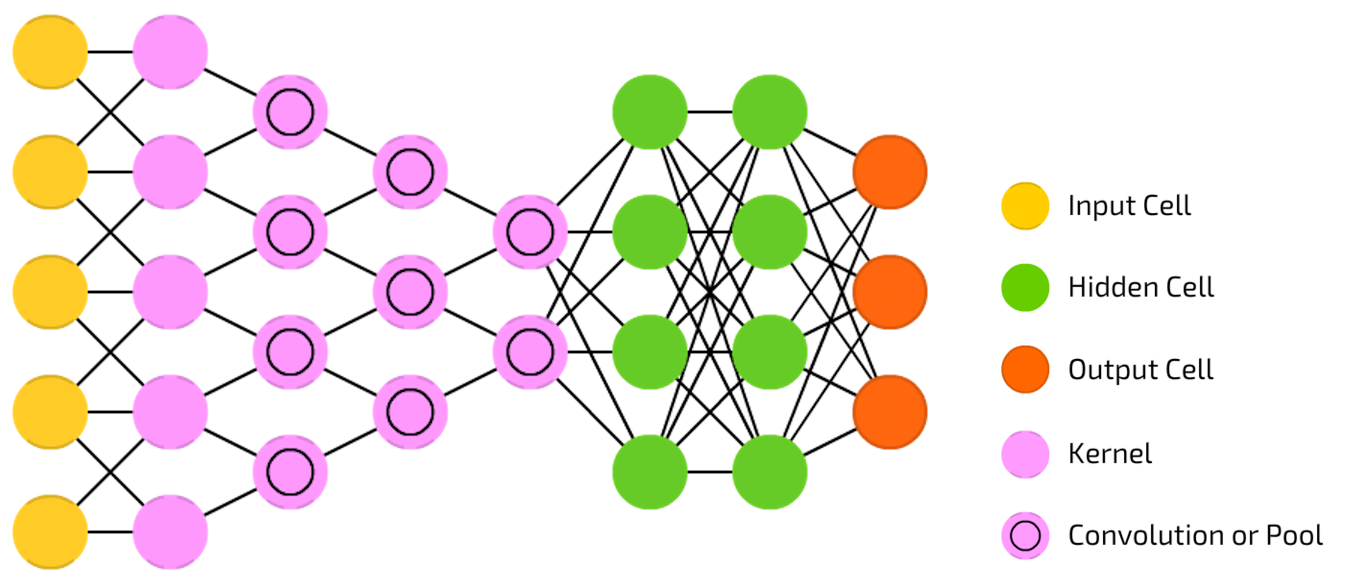


Рисунок 1.4 – Архитектура свёрточной нейронной сети

Свёрточные нейронные сети кардинально отличаются от других нейронных сетей. Они используются в основном для обработки изображений, иногда для аудио и других видов входных данных. Типичным способом применения сети является классификация изображений. Свёрточные нейронные сети используют операцию свёртки для фрагментов изображений, что даёт преимущество в производительности перед обычными сетями прямого распространения с полносвязными слоями [4]. Входные данные передаются через свёрточные слои, которые не являются полносвязными. Эти слои имеют свойство сжиматься с глубиной. Обычно они уменьшаются на делитель количества входных данных. Подобным делителем может быть степень двойки.

Кроме свёрточных слоев есть также так называемые слои объединения. Этот слой является хорошим способом уменьшить размерность получаемых данных, что в свою очередь опять же увеличивает производительность обучения и обработки сетью данных. После нескольких чередований слоёв свёртки и объединения, обработанные данные поступают на многослойный персептрон с относительно небольшим количеством скрытых слоёв сети. Такие сети называются глубокими свёрточными сетям, что обычно опускается на практике.

Другим вариантом глубокие свёрточные обратные глубинные сети (рисунок 1.5)



Рисунок 1.5 – Архитектура свёрточной обратной нейронной сети

Сеть напоминает вариационные автоэнкодеры со свёрточными и развертывающими сетями в качестве кодирующей и декодирующей частей соответственно. Такие сети представляют черты изображения в виде вероятностей и могут научиться строить, например, изображение кошки и собаки вместе, «взглянув» лишь на картинки только с кошками и только с собаками. Разработчики также продемонстрировали, что подобные сети могут моделировать различные сложные преобразования изображений, например, изменение источника света или поворот 3D объектов. Такие сети обычно обучают методом обратного распространения. Минусом подобной архитектуры, для решения данной задачи является тот факт, что на выходе нейронной сети само исходное изображение может достаточно сильно изменяться. Это связано с тем фактом, то в середине нейронной сети остаётся информация только о контексте изображения.

Третьим рассмотренным вариантом будет классический многослойный персептрон.

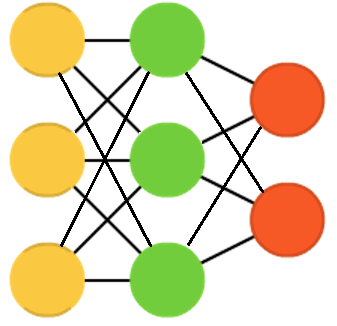


Рисунок 1.6 – Архитектура многослойного персептрона

Персептроны очень просты – они передают информацию от входа к выходу. Считается, что у нейронных сетей есть слои, каждый из которых состоит из входных, скрытых или выходных нейронов. Нейроны одного слоя между собой не связаны, при этом каждый нейрон этого слоя связан с каждым нейроном соседнего слоя. Простейшая мало-мальски рабочая сеть состоит из двух входных и одного выходного нейрона и может моделировать логический вентиль – базовый элемент цифровой схемы, выполняющий элементарную логическую операцию. FFNN обычно обучают методом обратного распространения ошибки, подавая модели на вход пары входных и ожидаемых выходных данных. Под ошибкой обычно понимаются различные степени отклонения выходных данных от исходных (например, среднеквадратичное отклонение или сумма модулей разностей). При условии, что сеть обладает достаточным количеством скрытых нейронов, теоретически она всегда сможет установить связь между входными и выходными данными. На практике использование сетей прямого распространения ограничено, и чаще они используются совместно с другими сетями. Данный факт связан с уменьшением производительности при увеличении количества нейронов и количества слоёв. Из-за того, что каждый нейрон предыдущего слоя связан с каждым нейроном следующего, количество связей настолько велико, что реализация достаточно большого многослойного персептрона требует огромных затрат производительности.

Из рассмотренных вариантов было принято решение использовать свёрточную нейронную сеть. Выбор обусловлен тем фактом, что ни один из аналогов не подходит для реализации сети для колоризации полутоновых изображений: многослойный персептрон имеет слишком много связей, каждая из которых имеет свой вес, который должен учитываться при прямом проходе через сеть; свёрточной обратной сети в свою очередь необходим длительный процесс обучения, для того, чтобы сеть выдавала подобные оригинальному изображения.

**1.3** Обзор вариантов обучения нейронной сети

Свёрточной нейронной сети, как и любой другой нейронной сети, необходимо обучение. Обучение нейронных сетей может происходить различными способами:

* обучение с учителем;
* обучение без учителя;
* смешанное обучение.

Наиболее типичным вариантом обучения для свёрточных нейронных сетей является обучение с учителем. Подобный вид обучения подразумевает наличие алгоритма, которые выполняет обучение нейронной сети. Для того, чтобы обучающий алгоритм мог выполнять свою работу необходима база подготовленных заранее изображений, которая являлась бы образцом того, как должна работать нейронная сеть. Поиск базы изображений был остановлен ImageNet.

ImageNet – это проект по созданию и сопровождению массивной базы данных аннотированных изображений, предназначенной для отработки и тестирования методов [распознавания образов](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D0%BE%D0%B7%D0%BD%D0%B0%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BE%D0%B1%D1%80%D0%B0%D0%B7%D0%BE%D0%B2) и [машинного зрения](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%BE%D0%B5_%D0%B7%D1%80%D0%B5%D0%BD%D0%B8%D0%B5). По состоянию на 2016 год в базу данных было записано около десяти миллионов URL с изображениями, которые прошли ручную аннотацию для ImageNet. В аннотациях перечислялись объекты, попавшие на изображение, и прямоугольники с их координатами. База данных с аннотацией и URL изображений от третьих лиц доступна непосредственно через ImageNet, но при этом сами изображения не принадлежат проекту [5]. Преимуществом данной базы изображений является наличие дерева изображений структурированного по контексту самого изображения (рисунок 1.7). Возможность получения изображений с определённым контекстом позволит в дальнейшем сделать нейронную сеть специализированной на обработке изображений с соответствующим контекстом. Такой подход к использованию программного средства позволит уменьшить время обучения сети и быстрее ввести продукт в эксплуатацию.

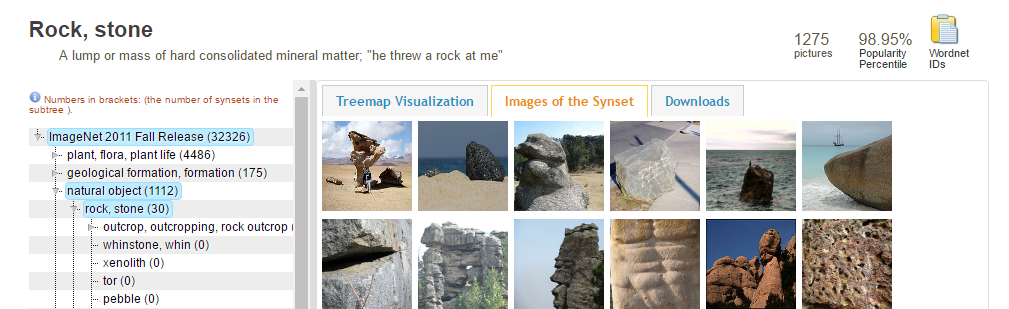


Рисунок 1.7 – Фрагмент веб-страницы базы изображений ImageNet, содержащий дерево изображений, структурированных по контексту

**1.4** Выбор средства построения нейронной сети

Построение нейронной сети является важным шагом в реализации проекта. Сам процесс построения нейронной сети должен быть простым и удобным, чтобы сохранить время, потраченное на создание сети и сконцентрироваться на других задачах. С этой целью был произведён поиск различных средств, которые позволили бы сократить процесс создания нейронной сети.

В статье [6] представлены возможности наиболее популярных программных средств глубокого обучения (см. таблицу 1.1). Кроме того, рассмотрены некоторые из фреймворков, представленных в таблице сравнения.

Таблица 1.1 – Возможности программных средств глубокого обучения

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Язык | Поддерживаемые операционные системы | Поддержка свёрточных сетей |
| [DeepLearnToolbox](https://github.com/rasmusbergpalm/DeepLearnToolbox) | Matlab | Windows, Linux | Есть |
| [Theano](http://deeplearning.net/software/theano/) | Python | Windows, Linux, Mac | Есть |
| [Pylearn2](http://deeplearning.net/software/pylearn2/) | Python | Linux, Vagrant | Есть |
| [Deepnet](https://github.com/nitishsrivastava/deepnet) | Python | Linux | Есть |
| [Deepmat](https://github.com/kyunghyuncho/deepmat) | Matlab |  | Есть |
| [Torch](http://torch.ch/) | Lua, C | Linux, OS X, iOS, Android | Есть |
| [Darch](http://cran.r-project.org/web/packages/darch/index.html) | R | Windows, Linux | Нет |
| [Caff](http://caffe.berkeleyvision.org/)e | C++, Python, Matlab | Linux, OS X | Есть |
| [nnForge](http://milakov.github.io/nnForge/) | С++ | Linux | Есть |
| [CXXNET](https://github.com/antinucleon/cxxnet) | С++ | Linux | Есть |
| [Cuda-convnet](https://code.google.com/p/cuda-convnet/) | С++ | Linux, Windows | Есть |
| [Cuda CNN](http://www.mathworks.com/matlabcentral/fileexchange/24291-cnn-convolutional-neural-network-class) | Matlab | Linux, Windows | Есть |

Автор статьи [6] для дальнейшего рассмотрения выбрал четыре библиотеки: [Theano](https://habrahabr.ru/company/intel/blog/254747/#TheanoLib) и [Pylearn2](https://habrahabr.ru/company/intel/blog/254747/#Pylearn2Lib) являются одними из самых зрелых и функционально полных библиотек, [Torch](https://habrahabr.ru/company/intel/blog/254747/#TorchLib) и [Caffe](https://habrahabr.ru/company/intel/blog/254747/#CaffeLib) – широко используются сообществом. Из данного списка только Theano, Pylearn2 и Caffe поддерживают язык программирования Python и операционные системы Linux. Выбор был остановлен на библиотеке Caffe.

Caffe реализована с использованием языка программирования C++, имеются обертки на Python и MATLAB. Официально поддерживаемые операционные системы – Linux и OS X. Для ускорения вычислений Caffe может быть запущена на GPU с использованием базовых возможностей технологии CUDA или библиотеки примитивов глубокого обучения [cuDNN](https://developer.nvidia.com/cuDNN).

Разработчики Caffe поддерживают возможности создания, обучения и тестирования полносвязнных и сверточных нейросетей. Входные данные и преобразования описываются понятием слоя. В зависимости от формата хранения могут использоваться следующие типы слоев исходных данных:

* DATA – определяет слой данных в формате leveldb и lmdb;
* HDF5\_DATA – слой данных в формате hdf5;
* IMAGE\_DATA – простой формат, который предполагает, что в файле приведен список изображений с указанием метки класса.

Преобразования могут быть заданы с помощью слоев:

* INNER\_PRODUCT – полностью связанный слой;
* CONVOLUTION – сверточный слой;
* POOLING – слой пространственного объединения;
* Local Response Normalization (LRN) — слой локальной нормализации.

Наряду с этим, при формировании преобразований могут использоваться различные функции активации [6].

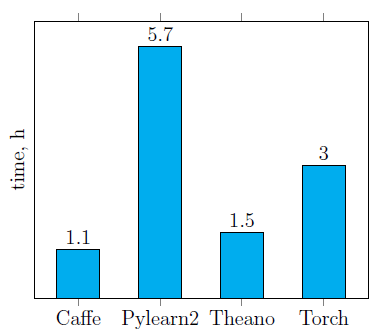


Рисунок 1.8 – Сравнение времени обучения выбранными библиотеками свёрточных сетей. GPU реализация.

**1.5** Выбор средства для реализации внешнего интерфейса сервера

Наиболее простыми и популярными Python-фреймворками для работы с сетью являются Bottle и Flask. Ниже подробнее рассматриваются эти 2 фреймворка.

[Bottle](http://bottlepy.org/) – это быстрый, простой и легкий микро веб-фреймворк для Python. Он распространяется в виде одного файла-модуля и не имеет никаких зависимостей, кроме стандартной библиотеки Python. Данный фреймворк требует установленный Python версии 2.5 или выше. Как было сказано выше Bottle не имеет дополнительных зависимостей, что упрощает процесс установки. Для установки Bottle можно воспользоваться easy\_install от Python [7]. Преимуществом данной библиотеки является скорость разработки небольших проектов.

Flask – [фреймворк](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA) для создания [веб-приложений](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) на языке программирования [Python](https://ru.wikipedia.org/wiki/Python), использующий набор инструментов Werkzeug, а также шаблонизатор [Jinja2](https://ru.wikipedia.org/wiki/Jinja). Как и Bottle Flask относится к категории так называемых [микрофреймворков](https://ru.wikipedia.org/w/index.php?title=%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D1%84%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA&action=edit&redlink=1) – минималистичных каркасов веб-приложений, сознательно предоставляющих лишь самые базовые возможности. Поддерживается установка посредством пакетного менеджера [PyPI](https://ru.wikipedia.org/wiki/PyPI). Flask требует версию Python 2.6 либо более высокую [8]. Пример простейшего веб-приложения с использованием данного фреймворка:

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

return "Hello World!"

if \_\_name\_\_ == "\_\_main\_\_":

app.run()

В процессе обзора данных фреймворков не был сделан однозначный выбор. Несмотря на простоту Bottle относительно Flask, сложно предугадать, окажется ли достаточно того функционала, что предоставляет эта библиотека, для реализации поставленной задачи.

Было решено произвести окончательный выбор фреймворка в процессе функционального проектирования, когда будет определён тот минимальный интерфейс, необходимый для передачи исходных и обработанных изображений по сети.

**1.6** Обзор ПО Docker

Docker – программное обеспечение для автоматизации развёртывания и управления приложениями в среде [виртуализации на уровне операционной системы](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BD%D0%B0_%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B). Позволяет «упаковать» приложение со всем его [окружением](https://ru.wikipedia.org/w/index.php?title=%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BE%D0%BA%D1%80%D1%83%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5&action=edit&redlink=1) и зависимостями в контейнер, который может быть перенесён на любую [Linux](https://ru.wikipedia.org/wiki/Linux)-систему с поддержкой [cgroups](https://ru.wikipedia.org/wiki/Cgroups) в [ядре](https://ru.wikipedia.org/wiki/%D0%AF%D0%B4%D1%80%D0%BE_Linux), а также предоставляет среду по управлению контейнерами.

Для экономии дискового пространства проект использует файловую систему [Aufs](https://ru.wikipedia.org/wiki/Aufs) с поддержкой технологии [каскадно-объединённого монтирования](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%81%D0%BA%D0%B0%D0%B4%D0%BD%D0%BE-%D0%BE%D0%B1%D1%8A%D0%B5%D0%B4%D0%B8%D0%BD%D1%91%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BC%D0%BE%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5): контейнеры используют образ базовой операционной системы, а изменения записываются в отдельную область. Также поддерживается размещение контейнеров в файловой системе [Btrfs](https://ru.wikipedia.org/wiki/Btrfs) с включённым режимом [копирования при записи](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BF%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%B8_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D0%B8).

В состав программных средств входит [демон](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BC%D0%BE%D0%BD_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0)) – [сервер](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5)) контейнеров, [клиентские](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B8%D0%B5%D0%BD%D1%82_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) средства, позволяющие из [интерфейса командной строки](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%BE%D0%B9_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B8) управлять образами и контейнерами, а также [API](https://ru.wikipedia.org/wiki/API), позволяющий в стиле [REST](https://ru.wikipedia.org/wiki/REST) управлять контейнерами программно.

Демон обеспечивает полную изоляцию запускаемых на узле контейнеров на уровне файловой системы (у каждого контейнера [собственная корневая файловая система](https://ru.wikipedia.org/wiki/Chroot)), на уровне процессов (процессы имеют доступ только к собственной файловой системе контейнера, а ресурсы разделены средствами libcontainer), на уровне сети (каждый контейнер имеет доступ только к привязанному к нему сетевому пространству имён и соответствующим виртуальным сетевым интерфейсам).

Набор клиентских средств позволяет запускать процессы в новых контейнерах, останавливать и запускать контейнеры, приостанавливать и возобновлять процессы в контейнерах. Серия команд позволяет осуществлять мониторинг запущенных процессов. Новые образы возможно создавать из специального сценарного файла, возможно записать все изменения, сделанные в контейнере в новый образ. Кроме того, в интерфейсе командной строки встроены возможности по взаимодействию с публичным репозиторием Docker Hub, в котором размещены предварительно собранные образы контейнеров. Например, команда docker search позволяет осуществить поиск среди размещённых в нём образов. Образы можно скачивать в локальную систему, возможно также отправить локально собранные образы в Docker Hub [9].

Было решено использовать Docker, т. к. это удобное средство, с помощью которого, в дальнейшем можно будет с минимальными усилиями разместить готовый сервис на серверной машине с Linux системой и предустановленным ПО Docker.

**1.7** Обзор Android SDK

В качестве клиентской части было выбрано приложение под ОС Android. Клиент должен иметь возможность поставлять полутоновые изображения на сервер. Если применять программное средство для реставрации устаревших фотографий или кинолент, необходимо соответствующее оборудование. Обычному прикладному приложению, без доступа к такому оборудованию, относительно сложно получить доступ к подобной фотографии. Приложение на мобильной платформе в свою очередь имеет доступ к камере мобильного телефона. Фотографии, полученные таким образом можно преобразовывать в полутоновые на клиентской стороне, а затем отправлять на сервер. Построив клиентскую часть подобным образом, можно использовать её, демонстрируя возможности нейронной сети и одновременно популяризируя эту сферу. Также причиной выбора платформы Android можно считать широкое распространение ОС и приложений по обработке изображений на этой системе, а также наличие опыта разработки программ под эту платформу.

Приложения для Android пишутся на языке программирования Java. Инструменты Android SDK компилируют написанный код – и все требуемые файлы данных и ресурсов – в файл APK – программный пакет Android, который представляет собой файл архива с расширением apk. В файле APK находится все, что требуется для работы Android-приложения, и он позволяет установить приложение на любом устройстве под управлением системы Android.

В системе существует понятие компонентов, из которых состоит приложение для Android. Компонент представляет собой отдельную точку, через которую система может войти в приложение. Не все компоненты являются точками входа для пользователя, а некоторые из них зависят друг от друга.

Существуют четыре типа компонентов [10].

* операции представляют собой один экран с пользовательским интерфейсом;
* службы представляют собой компоненты, которые работают в фоновом режиме и выполняют длительные операции, связанные с работой удаленных процессов; служба не имеет пользовательского интерфейса;
* поставщики контента управляют общим набором данных приложения; данные можно хранить в файловой системе, базе данных, в Интернете или любом другом постоянном месте хранения, к которому у приложения имеется доступ;
* приемники широковещательных сообщений представляют собой компоненты, которые реагируют на сообщения, распространяемые по всей системе; многие из этих сообщений рассылает сама система, однако есть возможность реализации собственных типов.

**1.8** Обзор средства для реализации внешнего интерфейса клиента

Для реализации внешнего интерфейса для работы клиентской части с сетью будет использована библиотека Retrofit 2. Такой выбор обусловлен удобством использования данной библиотеки, высокой скоростью работы, отличной документацией на официальном сайте, а также достаточно большим опытом работы с этой библиотекой.

Retrofit позволяет сделать полноценный REST-клиент, который может выполнять POST, GET, PUT, DELETE запросы. Для обозначения типа и других аспектов запроса используются аннотации. Например, для того, чтобы обозначить, что требуется GET запрос, необходимо написать аннотацию @[GET](https://habrahabr.ru/users/get/) перед методом, для POST запроса – @[POST](https://habrahabr.ru/users/post/), и так далее. Нотации описываются в стандартном интерфейсе языка Java [11].

**2** СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

В этом будет произведено разделение программного средства, реализуемого в данном дипломном проекте, на структурные блоки. Для каждого блока ниже будут описаны назначение блока, краткое описание и связь с остальными блоками.

Для данного дипломного проекта была выбрана архитектура «клиент- сервер» (рисунок 2.1). Серверная часть является основной частью проекта, так как именно она будет содержать свёрточную нейронную сеть, позволяющую колоризировать полутоновые изображения. Предполагается, что серверная часть будет содержать следующие блоки:

* блок колоризации изображений;
* блок обучения нейронной сети;
* блок хранения обработанных изображений;
* блок внешнего интерфейса серверной части.

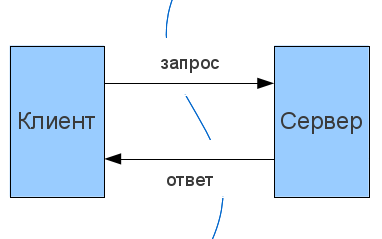


Рисунок 2.1 – Архитектура «клиент-сервер»

Блок колоризации изображений является основным структурным блоком всего дипломного проекта. Этот блок предназначен для колоризации полутоновых изображений. На вход данного блока поступают чёрно-белые изображения из блока внешнего интерфейса серверной части. Колоризованные изображения поступают в блок хранения обработанных изображений. Блок колоризации содержит нейронную сеть, которая в свою очередь не будет функционировать, если не произвести процесс обучения сети. Потому данный блок так же будет взаимодействовать с блоком обучения нейронной сети.

Блок обучения нейронной сети – структурный блок, который является необходимым для обучения нейронной сети. Свёрточные нейронные сети на практике обычно обучаются с использованием «учителя». Обучение с учителем – один из способов [машинного обучения](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D1%83%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), в ходе которого испытуемая система принудительно обучается с помощью примеров «стимул-реакция». Данный блок будет предоставлять блоку, содержащему нейронную сеть, полутоновые изображения, а также идеальный вариант того, как сеть должна обработать изображение. Для реализации подобного блока необходима база изображений, которые являются цветными. Эти изображения будут использоваться для предоставления нейронной сети идеального варианта её работы. В качестве же исходного изображения для сети будут использованы преобразованные в полутон изображения из подготовленной базы – информация о яркости каждого пикселя исходного изображения. Из этого ясно, что блок обучения будет содержать в себе логику, которая позволит попиксельно преобразовать цветное изображение в полутоновое. Алгоритм данного преобразования будет описан ниже.

Также важно понимать, что блок обучения необходим только на стадии проектирования сети и в процессе переобучения нейронной сети. В остальных случаях блок не задействуется и может быть исключён из готового приложения

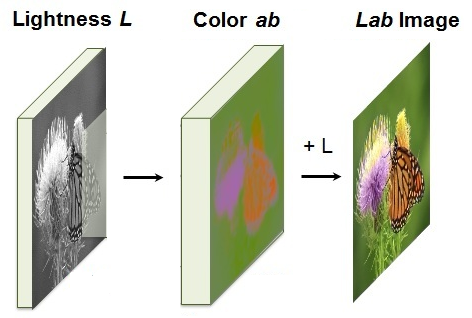


Рисунок 2.2 – Пример входного и выходного интерфейсов свёрточной нейронной сети, выполняющей задачу колоризации

Блок хранения обработанных изображений будет разработан для хранения колоризированных нейронной сетью изображений, которые готовы для передачи на клиентскую часть. Данный блок необходимо реализовать, потому что работа нейронной сети, особенно специализированной на обработку изображений, занимает относительно большое время. Если клиент будет пересылать изображение по сети, используя стандартный HTTP протокол, то вероятнее всего из-за длительной обработки полутонового изображения сервером произойдёт истечение времени для ожидания ответа на запрос. Поэтому было решено, что сервер после принятия сообщения будет сообщать клиенту о начале обработки переданного изображения свёрточной сетью либо же сообщать о невозможности обработать изображение на данный момент. После колоризации данный блок будет принимать для временного хранения обработанные изображения, и оповещать блок внешнего интерфейса о необходимости оповестить клиентское приложение об окончании обработки полутонового изображения серверной частью.

Средства необходимые для реализации такой логики будут подробнее описаны в следующих разделе посвящённом разработке программных модулей пояснительной записки.

Блок интерфейса серверной части является одним из звеньев связывающих серверную часть с клиентским приложением. Данный блок выполняет следующие функции:

* получение полутоновых изображений от клиента;
* оповещение клиента о приёме нейронной сетью изображения на обработку либо о невозможности обработки изображения в данный момент времени;
* оповещение о готовности передачи окрашенного нейронной сетью изображения обратно на клиентское приложение;
* передача обработанного изображения;
* проверка получаемых изображений;
* обработка ошибок, связанных со сбоем сети.

Разработка внешних интерфейсов серверной и клиентской частей представляет построение серверного API для дальнейшего его предоставления клиентской части и проектирование протокола прикладного уровня.

Клиентское мобильное приложение будет включать в себя следующие структурные блоки:

* блок получения изображения;
* блок предобработки изображения;
* блок внешнего интерфейса клиентской части;
* блок пользовательского интерфейса.

Блок получения изображения является источником для изображений, обработанные версии которых будут использоваться в качестве исходных изображений для свёрточной нейронной сети. Как было изложено ранее, если рассматривать данный дипломный проект в качестве средства для реставрации старых чёрно-белых фотографий или киноплёнок, то выбор платформы для клиента, должен зависеть от того, будет ли поддерживать данная платформа оборудование и программное обеспечение для оцифровки аналоговых источников. Так как доступ к подобному обеспечению и оборудованию отсутствует в рамках данного дипломного проекта, было дано предпочтение реализации клиентской части в качестве мобильного, а не прикладного приложения. Такой выбор оправдан, если позиционировать проект в качестве демонстрации работы свёрточной нейронной сети, выполняющей задачу колоризации изображения, так как мобильное приложение имеет возможность получить фотографию с помощью камеры, которой на текущий момент оборудовано практически любое мобильное устройство.

Реализация структурного блока получения изображения будет разниться от платформы, к платформе. Однако, так как предполагается, что именно операционная система Android будет являться платформой для клиентской части, исходя из опыта работы, можно сказать, что данный структурный блок будет представлять собой часть встроенного фреймворка для работы с камерой мобильного устройства, а также получением изображения из хранилища данных смартфона. Кроме того, этот блок будет обрабатывать исключительные ситуации, в случае если на устройство не будет оборудовано встроенной камерой или же доступ к ней будет блокирован системой.

Блок предобработки изображения необходим для преждевременной обработки полученного из предыдущего блока изображения. К подобной обработке можно отнести уменьшение размера изображения, его сжатие для передачи по сети. Кроме того, планируется реализация возможности получить полутоновое изображение из цветного, до его отправления на серверную часть. Данный структурный блок напрямую связан с внешним интерфейсом клиентской части. Более подробное описание реализации будет предоставлено в следующих разделах пояснительной записки.

Блок внешнего интерфейса клиентской части будет выполнять следующие функции:

* передача подготовленных изображений на сервер;
* ожидание оповещений от сервера о конце операции обработки изображения нейронной сетью;
* получение колоризированных нейронной сетью изображений от серверной части;
* обработка исключительных ситуаций, связанных со сбоем сети, или отказом сервера в обслуживании.

Сам по себе блок внешнего интерфейса клиентского приложения будет представлять собой надстройку над сторонней библиотекой Retrofit 2, краткое описание которой было приведено в обзоре литературы. Более подробное описание этой библиотеки, а также пример использования будет приведён в дальнейшем.

Блок пользовательского интерфейса является основным блоком для клиентской части. Данный блок в разной степени связан со всеми остальными структурными блоками мобильного приложения. Основными функциями данного блока являются реагирование на действие пользователя, а также отображение информации в том числе – обработанных изображений, полученных от сервера.

В целом же любой блок клиентской части проекта можно разделить при помощи MVC – схема разделения данных приложения, [пользовательского интерфейса](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D0%BA%D0%B8%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81) и управляющей логики на три отдельных компонента: модель, представление и контроллер (рисунок 2.3).

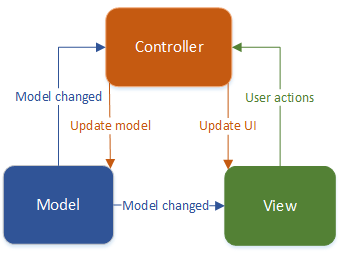


Рисунок 2.3 – Паттерн MVC

**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.

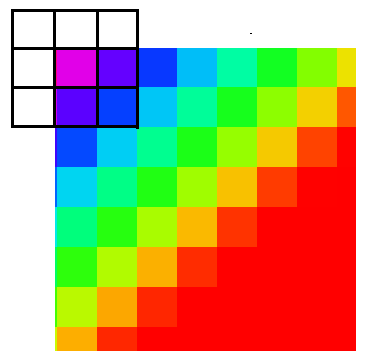


Рисунок 3.1 – Итерация обработки изображения свёрточным фильтром

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.

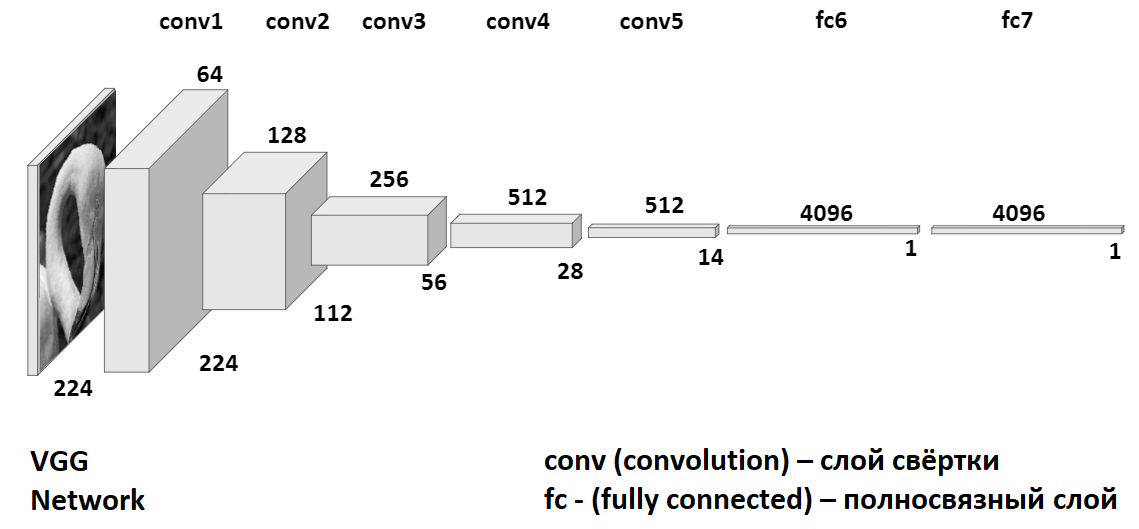


Рисунок 3.2 – Архитектура VGG сети

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.

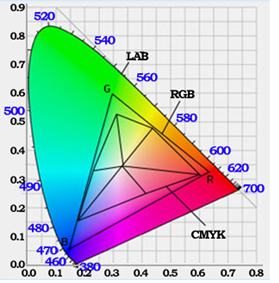


Рисунок 3.3 – Сравнение покрытия цветового пространства разными цветовыми моделями

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.

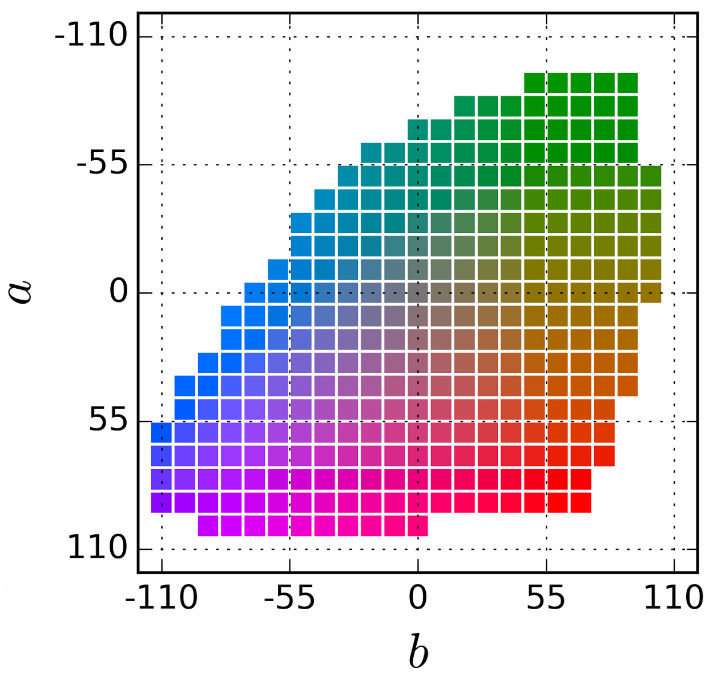


Рисунок 3.4 – Квантование цветового пространства LAB

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).

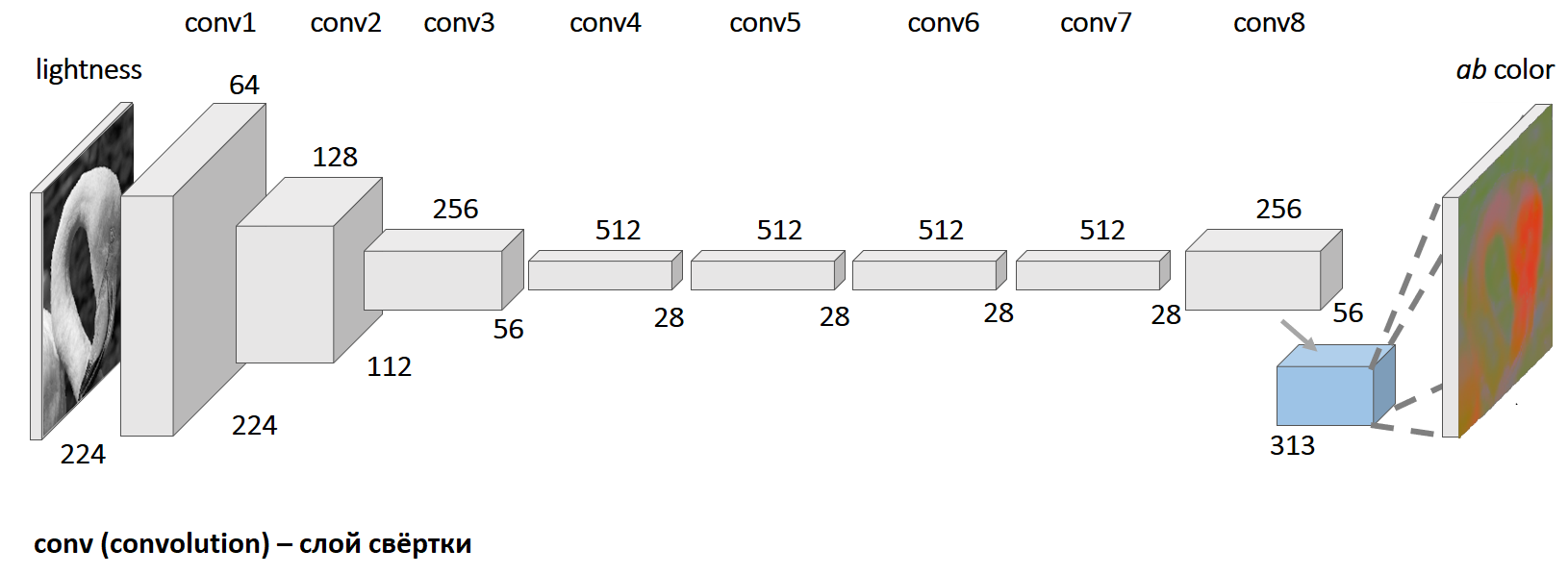


Рисунок 3.5 – Архитектура нейросети для колоризации изображения

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. Данный индентификатор предоставляется самим клиентом.

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.

**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)

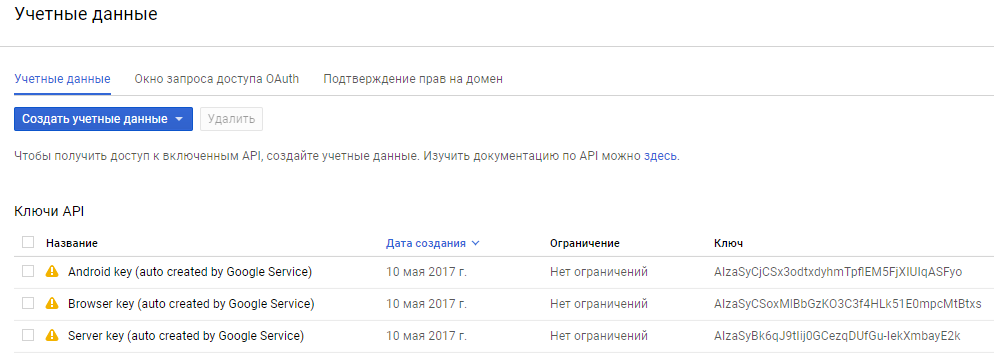


Рисунок 3.6 – Фрагмент веб-страницы Google API Console с учётными данными приложения

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, который генерируется клиентом и передаётся в момент загрузки изображения, и token клиента, предоставляемый самим клиентом.:

def notify(id, token)

Код модуля находится в файле notifier.py, который не имеет дополнительных зависимостей и расположен в каталоге neuralnet 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. Приложения на мобильной платформе Android могут иметь одну или несколько входных точек. 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. Предоставлением камеры устройства приложению по факту является предоставление объекта намерения. 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. Два из трёх классов нужны для взаимодействия с сетью. 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

....// like Toast.show()  
}

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. Для использования этого функционала необходимо добавить в проект сервис, переопределяющий метод onMessageReceived этого класса:

@Override  
public void onMessageReceived(RemoteMessage remoteMessage)

Объект remoteMessage содержит список пар «ключ-значение», которые были заключены в посланном сообщении. В данном проекте создан одноимённый базовому класс, который в переопределённом методе создаёт намерение с определённым фильтром, заключает идентификатор изображения, который был послан сервером, внутрь этого намерения и затем выполняет его широковещательную посылку. Для работы данного сервиса необходимо описать его фильтр com.google.firebase.MESSAGING\_EVENT в файле AndroidManifest.xml, а также добавить сгенерированный веб-приложением Firebase файл google-services.json, содержащий API-ключ, в корневой каталог модуля проекта. Сам сервис находится в одноимённом файле и является частью пакета gcm проекта.

Как было указано в одном из подразделов функционального проектирования приёмником широковещательных сообщений от GCM-сервиса является статический класс PushHandler заключённый внутри активности ColorizationActivity. Конструктор внутреннего класса имеет следующий интерфейс:

public PushHandler(ColorizationActivity activity)

Таким образом приёмник широковещательных сообщений получает доступ к полям активности, которая его использует. Для функционирования приёмника необходимо переопределить метод onReceive:

@Override  
public void onReceive(Context context, final Intent intent)

Аргументами метода являются контекст, с помощью которого было послано намерение, а также сам объект намерения, содержащий идентификатор обработанного сообщения. Единственной функцией приёмника является вызов DownloadingTask, для загрузки и отображения изображения. Стоит заметить, что для функционирования широковещательного приёмника при создании активности необходимо зарегистрировать объект класса с фильтром намерений, который совпадает с тем, что указан в GCM-сервисе. В проекте для этого используется строка «PUSH\_MESSAGE\_RECEIVED», которая заключается внутрь объекта IntentFilter.

**3.15** Передача изображения между активностями

Как было указано ранее PickerActivity отображает предобработанное изображение, а затем передаёт его активности ColorizationActivity. Проблема состоит в том, что объект Bitmap содержащий изображение может быть слишком большого размера для его передачи через Intent. Для того, чтобы передать изображение в другую активность в пакете utils был создан класс BitmapHolder. Класс содержит статическое поле для хранения изображения и два статических метода: save и get:

public static void save(Bitmap bitmap)   
public static Bitmap get()

Метод save сохраняет объект Bitmap в статическое поле и вызывается активностью, которая собирается передать изображение. Метод get предоставляет изображение другой активности и обнуляет ссылку на его в статическом поле во избежание утечек памяти.

**4** РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

В данном разделе будет рассмотрена реализация отдельных блоков приложения. Для реализации клиентского приложения для платформы Android использовались среда разработки Android Studio и язык программирования Java. Также была использована система автоматической сборки Gradle, встроенная в Android Studio, в качестве основного модуля сборки проекта. Для реализации серверной части были использованы:

* язык программирования Python;
* интерпретатор языка Python версии 2.7;
* среда разработки JetBrains PyCharm Community Edition, которая имеет интерфейс схожий с Android Studio и позволяет проверить синтаксис скрипта на языке Python;
* текстовый редактор Sublime Text 3, который использовался в качестве более удобного способа изменения исходного кода.

Распределённая система управления версиями Git использовалась, для хранения версий исходного кода клиентского приложения и серверной части. Стоит также отметить, что Git позволила более удобно переносить код внутрь Docker-контейнера, что крайне затруднительно при использовании Windows-хоста, из-за различия в структурах файловых систем.

**4.1** Обучение нейронной сети

Для обучения нейронной сети необходимы:

* База изображений в формате LMDB файлов;
* Видеокарта, для ускорения процесса обучения;
* Время, для исполнения достаточного количества итераций обучения.

Согласно статье [12] для подобной сети требуется около 450000 итераций, что при отсутствии графического процессора, может занимать около от 55 до 90 часов обучения при использовании фреймворка Caffe. Ввиду ограниченного количества времени на реализацию проекта, отсутствия графического процессора на персональном компьютере, места для хранения базы изображений, а также стабильного Интернет-соединения для загрузки изображений, в готовом проекте использована предобученная нейронная сеть. Однако, в данном подразделе всё же будет описан процесс обучения нейронной сети на основе информации из первоисточника [13].

Для обучения нейронной сети необходимо выгрузить некоторые файлы из удалённого git-репозитория первоисточника. Все команды, которые необходимо выполнить в процессе обучения вводятся с помощью интерфейса командной строки Linux-системы.

wget eecs.berkeley.edu/~rich.zhang/projects/2016\_colorization/files/train/init\_v2.caffemodel -O ./init.caffemodel

Исполнение данной команды загрузит Caffe-модель, которую инициализировали методом k-средних, в корневой каталог выгруженного проекта.

wget eecs.berkeley.edu/~rich.zhang/projects/2016\_colorization/files/train/caffe-colorization.tar.gz -O ./caffe-colorization.tar.gz

tar -xvf ./caffe-colorization.tar.gz

rm ./caffe-colorization.tar.gz

Выполнение данных команд загрузит модифицированный под задачу обучения конкретной нейронной сети фреймворк Caffe, упакованный в архив с расширением tar.gz. Затем произойдёт распаковка архива и его удаление.

Затем следует добавить каталог resources проекта в системную переменную $PYTHONPATH. Данный шаг может различаться для разных дистрибутивов Linux. Здесь приведён способ добавить системную переменную в операционной системе Ubuntu:

export VAR\_NAME=VAR\_VALUE

После добавления системной переменной необходимо переопределить пути к базе изображений в файле models/colorization\_train\_val\_v2.prototxt. Затем само обучение запускается при помощи следующей команды:

./caffe-colorization/build/tools/caffe train -solver ./train/solver.prototxt -weights ./init.caffemodel -gpu $1

Следующая команда использует модифицированный Caffe. В процессе обучения будет использована графический процессор, идентификатор которого указан флагом gpu. Файл solver.prototxt содержит следующие параметры:

net: "./models/colorization\_train\_val\_v2.prototxt"

test\_iter: 1 # Количество итераций за шаг

test\_interval: 10000000

test\_initialization: False

base\_lr: 3.16e-5

lr\_policy: "step"

gamma: 0.316

stepsize: 215000

display: 1

max\_iter: 500000

momentum: 0.9

momentum2: 0.99

weight\_decay: 0.001

snapshot: 1000

snapshot\_prefix: "./train/models/colornet"

solver\_mode: GPU

type: "Adam"

average\_loss: 1000

Обучение заканчивается после 500000 итераций. Само обучение происходит на случайных частях изображений, которые также могут быть зеркальными по отношению к оригиналу. Снимки, отображающие процесс обучения, будут сохраняться каждые 1000 итерации по пути, который указан в параметре snapshot\_prefix файла solver. Сами снимки имеют расширение solverstate.

**4.2** Описание архитектуры сети

В данном подразделе приведена часть файла deploy.prototxt, который находится в каталоге neuralnet проекта и используется модулем colorization.py. Данный файл содержит описание архитектуры свёрточной нейронной сети, предназначенной для решения задачи колоризации.

name: "LtoAB"

layer {

name: "data\_l"

type: "Input"

top: "data\_l"

input\_param {

shape { dim: 1 dim: 1 dim: 224 dim: 224 }

}

}

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# \*\*\*\*\* conv1 \*\*\*\*\*

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

layer {

name: "bw\_conv1\_1"

type: "Convolution"

bottom: "data\_l"

top: "conv1\_1"

convolution\_param {

num\_output: 64

pad: 1

kernel\_size: 3

}

}

layer {

name: "relu1\_1"

type: "ReLU"

bottom: "conv1\_1"

top: "conv1\_1"

}

layer {

name: "conv1\_2"

type: "Convolution"

bottom: "conv1\_1"

top: "conv1\_2"

convolution\_param {

num\_output: 64

pad: 1

kernel\_size: 3

stride: 2

}

}

layer {

name: "relu1\_2"

type: "ReLU"

bottom: "conv1\_2"

top: "conv1\_2"

}

layer {

name: "conv1\_2norm"

type: "BatchNorm"

bottom: "conv1\_2"

top: "conv1\_2norm"

batch\_norm\_param{ }

param {lr\_mult: 0 decay\_mult: 0}

param {lr\_mult: 0 decay\_mult: 0}

param {lr\_mult: 0 decay\_mult: 0}

}

В следующем примере указаны метка-имя архитектуры name, входной слой с именем data\_l, а также составной слой conv\_1, состоящий из 5 слоёв различных типов:

* Convolution – слой свёртки; пример такого слоя с некоторыми его параметрами был описан в разделе функционального проектирования;
* ReLU – слой функции активации после свёрточного слоя; для активации выбирается вместо обычных функций типа гиперболического тангенса или сигмоиды ненасыщаемая функция, которая принимает значение аргумента, если сам аргумент больше нуля, и значение нуля, если аргумент – отрицательное число. Такая функция показывает хорошие результаты при обучении нейронных сетей и отвечает за отсечение ненужных деталей в канале (при отрицательном выходе);
* BatchNorm – слой батч-нормализации.

Из параметров к слою свёртки, кроме уже описанных, здесь приведён параметр stride, обозначающий интервал в пикселях, на который сдвигается ядро свёртки. Наличие слоёв батч-нормализации связано со следующей проблемой: по мере распространения сигнала по сети, даже если нормализовывать его на входе, пройдя через внутренние слои, сигнал может сильно исказиться как по математическому ожиданию, так и по дисперсии (данное явление называется внутренним ковариационным сдвигом), что чревато серьезными несоответствиями между градиентами на различных уровнях.

Одним из последних слоёв нейронной сети является слой Softmax. Особенностью данного слоя является тот факт, что сумма всех значений выходов нейронов этого слоя равна единице:

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# \*\*\*\*\* Softmax \*\*\*\*\*

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

layer {

name: "conv8\_313"

type: "Convolution"

bottom: "conv8\_3"

top: "conv8\_313"

convolution\_param {

num\_output: 313

kernel\_size: 1

stride: 1

dilation: 1

}

}

layer {

name: "conv8\_313\_rh"

type: "Scale"

bottom: "conv8\_313"

top: "conv8\_313\_rh"

scale\_param {

bias\_term: false

filler { type: 'constant' value: 2.606 }

}

}

layer {

name: "class8\_313\_rh"

type: "Softmax"

bottom: "conv8\_313\_rh"

top: "class8\_313\_rh"

}

Последним является слой для декодирования значений результата предыдущего слоя в значения каналов A и B:

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# \*\*\*\*\* Decoding \*\*\*\*\*

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

layer {

name: "class8\_ab"

type: "Convolution"

bottom: "class8\_313\_rh"

top: "class8\_ab"

convolution\_param {

num\_output: 2

kernel\_size: 1

stride: 1

dilation: 1

}

}

layer {

name: "Silence"

type: "Silence"

bottom: "class8\_ab"

}

Именно с выходов слоя class8\_ab снимается значения цветов пикселей. Cлой Silence добавляется в качестве заглушки для готовых моделей сети.

**4.3** Модуль colorization.py

В данном подразделе будет в основном описано использование Python-интерфейса фреймворка Caffe.

Выбор центрального процессора в качестве устройства для вычисления значений при прямом проходе по сети:

caffe.set\_mode\_cpu()

Загрузка модели нейронной сети:

net = caffe.Net(prototxt, caffemodel, caffe.TEST)

Аргументы метода Net:

* prototxt – путь к prototxt-файлу, описывающему архитектуру сети;
* caffemodel – путь к caffemodel-файлу, содержащему обученную модель сети;
* caffe.TEST – значение, которое указывает на необходимость вывода отчёта о работе сети в поток STDOUT.

Получение размеров входного и выходного слоя нейронной сети:

(H\_in, W\_in) = net.blobs['data\_l'].data.shape[2:]

(H\_out, W\_out) = net.blobs['class8\_ab'].data.shape[2:]

Загрузка изображения из файловой системы:

img\_rgb = caffe.io.load\_image(img\_in)

Получение одноканального изображения (L-канал) для последующей конкатенации с результатом нейронной сети:

img\_lab = color.rgb2lab(img\_rgb)

img\_l = img\_lab[:, :, 0]

Получение размеров исходного изображения:

(H\_orig, W\_orig) = img\_rgb.shape[:2]

Получение одноканального изображения (L-канал) для входного слоя нейронной сети (изменения размеров исходного изображения для входного слоя, перевод в цветовую схему LAB и выделения канала яркости:

img\_rs = caffe.io.resize\_image(img\_rgb, (H\_in, W\_in))

img\_lab\_rs = color.rgb2lab(img\_rs)

img\_l\_rs = img\_lab\_rs[:, :, 0]

Загрузка исходного изображения в сеть:

net.blobs['data\_l'].data[0, 0, :, :] = img\_l\_rs - 50

Запуск нейронной сети:

net.forward()

Снятие результата с выходного слоя сети:

ab\_dec = net.blobs['class8\_ab'].data[0, :, :, :].transpose((1, 2, 0))

Увеличение разрешения результата до разрешения оригинального изображения:

ab\_dec\_us = sni.zoom(ab\_dec, (1.\*H\_orig / H\_out, 1.\*W\_orig / W\_out, 1))

Получение обработанного изображения в цветовой модели LAB и последующий перевод в цветовую модель RGB:

img\_lab\_out = np.concatenate((img\_l[:, :, np.newaxis], ab\_dec\_us), axis = 2)

img\_rgb\_out = (255 \* np.clip(color.lab2rgb(img\_lab\_out), 0, 1)).astype('uint8')

Сохранение изображения в файловую систему:

plt.imsave(img\_out, img\_rgb\_out)

Вызов модуля оповещения:

notifier.notify(id, token)

Весь код модуля управления нейронной сетью приложен в листинге.

**4.4** GCM-модуль оповещения:

В данном подразделе указан код GCM-модуля оповещения целиком, так как интерфейс библиотеки python-gcm очень прост и не требует большого количества строк кода:

from gcm import GCM

def notify(id, token):

gcm = GCM("AIzaSyBk6qJ9tIij0GCezqDUfGu-IekXmbayE2k")

data = {'id': id}

gcm.plaintext\_request(registration\_id=token, data=data)

Функция GCM возвращает объект, для дальнейшей работы с GCM библиотекой. Принимает она API-ключ, который был получен при регистрации проекта в веб-приложении Firebase. Объект data заключает в себе данные, которые будут переданы через GCM-сообщение. Вызов функции plaintext\_request инициализирует отправку сообщения клиенту с идентификатором token.

**4.5** Активность для выбора исходного изображения

В данном подразделе будет указан код файла PickerActivity.java, который является входной точкой приложения.

При создании активности вызывается метод onCreate, в котором происходит инициализация интерфейса пользователя:

@Override  
protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.activity\_picker);  
 findViews();  
 setUpViews();  
}  
  
private void findViews() {  
 imageView = (ImageView) findViewById(R.id.iv);  
 progressBar = (ProgressBar) findViewById(R.id.pb);  
 cameraButton = (Button) findViewById(R.id.b\_camera);  
 galleryButton = (Button) findViewById(R.id.b\_gallery);  
 loadButton = (Button) findViewById(R.id.b\_load);  
}  
  
private void setUpViews() {  
 cameraButton.setOnClickListener(this);  
 galleryButton.setOnClickListener(this);  
 loadButton.setOnClickListener(this);  
}

Метод findViewById связывает объект интерфейса с ссылкой на него в активности. Метод setOnClickListener объявляет обработчиком нажатий на контроллеры объект активности, который реализует интерфейс View.OnClickListener:

@Override  
public void onClick(View v) {  
 enableControls(false);  
 switch (v.getId()) {  
 case R.id.b\_camera:  
 Pair<Intent, Uri> pair = CameraProvider.getCameraIntent(this);  
 imageUri = pair.second;  
 startActivityForResult(pair.first, CODE\_REQUEST\_CAMERA);  
 break;  
 case R.id.b\_gallery:  
 Intent pickPhoto = new Intent(Intent.ACTION\_PICK,  
 android.provider.MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI);  
 startActivityForResult(pickPhoto, CODE\_REQUEST\_GALLERY);  
 break;  
 case R.id.b\_load:  
 if (bitmap != null) {  
 BitmapHolder.save(bitmap);  
 startActivity(new Intent(this, ColorizationActivity.class));  
 }  
 enableControls(true);  
 break;  
 }  
}

При нажатии на объект интерфейса с идентификатором R.id.b\_load вызывается метод startActivity, который в свою очередь вызывает активность для передачи изображения на сервер и отображение обработанного изображения. При нажатии на другие элементы интерфейса вызывается метод startActivityForResult, который в свою очередь вызывает стороннюю активность, возвращающую URI выбранного изображения в метод onActivityResult:

@Override  
protected void onActivityResult(int requestCode, int resultCode, Intent data) {  
 super.onActivityResult(requestCode, resultCode, data);  
 if (resultCode == RESULT\_OK) {  
 Uri uri = requestCode == CODE\_REQUEST\_CAMERA ? imageUri  
 : requestCode == CODE\_REQUEST\_GALLERY ? data.getData() : null;  
 imageUri = null;  
 if (uri != null) {  
 AsyncTaskCompat.executeParallel(new GetBitmapTask(this, uri) {  
 @Override  
 protected void onPostExecute(Throwable tr) {  
 super.onPostExecute(tr);  
 enableControls(true);  
 if (tr == null) {  
 PickerActivity.this.bitmap = getBitmap();  
 imageView.setImageBitmap(bitmap);  
 } else Toast.makeText(PickerActivity.this, tr.getMessage(),  
 Toast.LENGTH\_SHORT).show();  
 }  
 });  
 return;  
 }  
 }  
 enableControls(true);  
}

При получении URI от сторонней активности вызывается метод AsyncTaskCompat.executeParallel, который запускает объект AsyncTask, переданный в метод в качестве параметра. При получении изображения из асинхронной задачи вызывается метод ImageView.setImageBitmap, который отображает полученное изображение на экран.

**4.6** Модуль предобработки изображения

В данном подразделе описан код файла ImageProcessor.java, предоставляющий интерфейс для обработки изображений:

public static Bitmap getGrayScale(Bitmap original) {  
 int width, height;  
 height = original.getHeight();  
 width = original.getWidth();  
 Bitmap grayscale = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB\_8888);  
 Canvas c = new Canvas(grayscale);  
 Paint paint = new Paint();  
 ColorMatrix cm = new ColorMatrix();  
 cm.setSaturation(0);  
 ColorMatrixColorFilter f = new ColorMatrixColorFilter(cm);  
 paint.setColorFilter(f);  
 c.drawBitmap(original, 0, 0, paint);  
 return grayscale;  
}  
  
public static Bitmap resize(Bitmap original, float scale) {  
 int width = original.getWidth();  
 int height = original.getHeight();  
 int newWidth = (int) (width \* scale);  
 int newHeight = (int) (height \* scale);  
 float scaleWidth = ((float) newWidth) / width;  
 float scaleHeight = ((float) newHeight) / height;  
 Matrix matrix = new Matrix();  
 matrix.postScale(scaleWidth, scaleHeight);  
 return Bitmap.createBitmap(original, 0, 0, width, height, matrix, false);  
}

Для получения полутонового изображения используется метод getGrayScale. Для создания нового Bitmap-объекта используется метод Bitmap.createBitmap. Для того, чтобы получить значения яркости изображения используется объект ColorMatrix, методы ColorMatrix.setSaturation, и Canvas.drawBitmap.

В методе resize cначала вычисляются scaleWidth и scaleHeight обработанного изображения на основании аргумента. Затем с помощью объекта matrix и метода createBitmap можно получить изображение с новым разрешением.

**4.7** Активность для отображения колоризованного изображения

При создании активности вызывается метод onCreate, в котором происходит получение изображения из BitmapHolder, отображение полученного изображения на экран, регистрация объекта PushHandler и вызов AsyncTask.executeParallel на объекте UploadingTask:

@Override  
protected void onCreate(@Nullable Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.activity\_colorization);  
 findViews();  
 backButton.setOnClickListener(this);  
 bitmap = BitmapHolder.get();  
 imageView.setImageBitmap(bitmap);  
 handler = new PushHandler(this);  
 LocalBroadcastManager.getInstance(this).registerReceiver(handler,  
 new IntentFilter(INTENT\_ACTION));  
 currentTask = new UploadingTask(bitmap) {  
 @Override  
 protected void onPreExecute() {  
 super.onPreExecute();  
 progressBar.setVisibility(View.VISIBLE);  
 }  
  
 @Override  
 protected void onPostExecute(Throwable tr) {  
 super.onPostExecute(tr);  
 if (tr != null) progressBar.setVisibility(View.GONE);  
 Toast.makeText(ColorizationActivity.this, tr == null ? LOADING\_SUCCESS\_MESSAGE  
 : tr.getMessage(), Toast.LENGTH\_SHORT).show();  
 }  
 };  
 AsyncTaskCompat.executeParallel(currentTask);  
}

При уничтожении активности вызывается метод onDestroy, внутри которого происходит отмена регистрации объекта PushHandler:

@Override  
protected void onDestroy() {  
 LocalBroadcastManager.getInstance(this).unregisterReceiver(handler);  
 super.onDestroy();  
}

В случае нажатия на кнопку «BACK» или на соответствующий контроллер смартфона вызывается отмена текущего AsyncTask-объекта:

@Override  
public void onClick(View v) {  
 switch (v.getId()) {  
 case R.id.b\_back:  
 if (currentTask != null && !currentTask.isCancelled())  
 currentTask.cancel(true);  
 finish();  
 break;  
 }  
}  
@Override  
public void onBackPressed() {  
 if (currentTask != null && !currentTask.isCancelled())  
 currentTask.cancel(true);  
 super.onBackPressed();  
}

**4.7** Задача загрузки изображения

В данном подразделе приведён код файла UploadingTask.java. В данном файле кроме посылки запроса на сервер заключены перевод в формат изображения в строку формата Base64 и назначения идентификаторов изображениям:

@Override  
protected final Throwable doInBackground(Object... params) {  
 try {  
 String base64 = getBase64(bitmap);  
 String id = getRandomId();  
 API.getInterface().loadImage(FirebaseInstanceId.getInstance().getToken(), id, base64)  
 .execute();  
 return null;  
 } catch (IOException e) {  
 return e;  
 }  
}  
  
private String getBase64(Bitmap original) {  
 ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();  
 original.compress(Bitmap.CompressFormat.PNG, 100, byteArrayOutputStream);  
 byte[] byteArray = byteArrayOutputStream.toByteArray();  
 return Base64.encodeToString(byteArray, Base64.DEFAULT);  
}  
  
private String getRandomId() {  
 return String.valueOf((new Random()).nextLong());  
}

Для назначения изображениям случайных идентификаторов используется класс Random. При переводе объекта Bitmap в строку происходит его псевдосжатие в формат PNG при помощи метода Bitmap.compress. Для перевода массива байтов в строку формата Base64 используется одноимённый класс.

**4.8** Обработка входящих GCM-сообщений

Для данной цели служат классы FirebaseMessagingService и PushHandler:

public class FirebaseMessagingService extends com.google.firebase.messaging.FirebaseMessagingService {  
  
 private static final String INTENT\_ACTION = "PUSH\_MESSAGE\_RECEIVED";  
  
 @Override  
 public void onMessageReceived(RemoteMessage remoteMessage) {  
 Intent intent = new Intent(INTENT\_ACTION);  
 intent.putExtra("id", remoteMessage.getData().get("id"));  
 LocalBroadcastManager.getInstance(this).sendBroadcast(intent);  
 }  
}

Класс FirebaseMessagingService при получении входящего сообщения вызывает метод LocalBroadcastManager.sendBroadcast, куда передаётся объект намерения, содержащий идентификатор изображения и строку-фильтр PUSH\_MESSAGE\_RECEIVED. Объект класса PushHandler, созданный активностью ColorizationActivity, принимает широковещательные сообщение именно с этим фильтром и затем вызывает запуск задачи DownloadingTask:

@Override  
public void onReceive(Context context, final Intent intent) {  
 activity.runOnUiThread(new Runnable() {  
 @Override  
 public void run() {  
 String id = intent.getStringExtra("id");  
 activity.currentTask = new DownloadingTask(id) {  
 @Override  
 protected void onPostExecute(Throwable tr) {  
 super.onPostExecute(tr);  
activity.progressBar.setVisibility(View.GONE);  
 if (tr == null) {  
 activity.bitmap = getBitmap();  
 activity.imageView.setImageBitmap(activity.bitmap);  
 } else Toast.makeText(activity, tr.getMessage(),  
 Toast.LENGTH\_SHORT).show();  
 }  
 };  
 AsyncTaskCompat.executeParallel(activity.currentTask);  
 }  
 });  
}

При получении обработанного изображение оно отображается на экран при помощи метода ImageView.setImageBitmap. В случае любой ошибки, сообщение, которое она содержит, выводится на экран при помощи метода Toast.makeText.

**4.9** Модуль предоставляющий интерфейс для взаимодействия с сервером

В данном подразделе приведён код класса API, который в некоторой степени реализует паттерн Singleton.

public final class API {  
  
 private static final String SERVER\_ADDRESS = "http://192.168.0.104:8080/";  
  
 private static ServerInterface server;  
  
 private API() {  
  
 }  
 public static ServerInterface getInterface() {  
 if (server == null) server = new Retrofit.Builder()  
 .baseUrl(SERVER\_ADDRESS)  
 .addConverterFactory(ScalarsConverterFactory.create())  
 .build()  
 .create(ServerInterface.class);  
 return server;  
 }  
}

Данный класс содержит адрес сервера, заключённый в строку SERVER\_ADDRESS. Получение объекта, реализующего интерфейс ServerInterface происходит при помощи метода Retrofit.Builder.create, который является частью фреймворка Retrofit.

**4.10** Файлы AndroidManifest.xml и build.gradle

Файл манифеста используется при разработке Android-приложения, для описания основных компонентов приложения и их свойств, требующихся приложению ресурсов, разрешения на использование которых будет получено у пользователя при установке. Также данный файл содержит имя приложения и ссылки на «иконки» для приложения:

<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package="by.evgeniyshilov.colorizationclient">  
  
 <uses-permission android:name="android.permission.INTERNET" />  
 <uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />  
  
 <application  
 android:allowBackup="true"  
 android:icon="@mipmap/ic\_launcher"  
 android:label="@string/app\_name"  
 android:roundIcon="@mipmap/ic\_launcher\_round"  
 android:supportsRtl="true"  
 android:theme="@style/AppTheme">  
  
 <service android:name=".gcm.FirebaseMessagingService">  
 <intent-filter>  
 <action android:name="com.google.firebase.MESSAGING\_EVENT" />  
 </intent-filter>  
 </service>  
  
 <activity android:name=".activities.PickerActivity">  
 <intent-filter>  
 <action android:name="android.intent.action.MAIN" />  
 <category android:name="android.intent.category.LAUNCHER" />  
 </intent-filter>  
 </activity>  
  
 <activity android:name=".activities.ColorizationActivity" />  
  
 </application>  
  
</manifest>

Файл build.gradle используется для указания информации о целевой и минимальной версии Android для разрабатываемого приложения, версии самого приложения. Также файл содержит зависимости, которые должна подгрузить Gradle, а также некоторые дополнительные параметры для системы сборки:

apply plugin: 'com.android.application'  
  
android {  
 compileSdkVersion 25  
 buildToolsVersion "25.0.3"  
 defaultConfig {  
 applicationId "by.evgeniyshilov.colorizationclient"  
 minSdkVersion 21  
 targetSdkVersion 25  
 versionCode 1  
 versionName "1.0"  
 testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"  
 }  
 buildTypes {  
 release {  
 minifyEnabled false  
 proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'  
 }  
 }  
}  
  
dependencies {  
 compile fileTree(dir: 'libs', include: ['\*.jar'])  
 androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {  
 exclude group: 'com.android.support', module: 'support-annotations'  
 })  
 compile 'com.android.support:appcompat-v7:25.3.1'  
 compile 'com.google.firebase:firebase-messaging:10.2.4'  
 compile 'com.squareup.retrofit2:retrofit:2.2.0'  
 compile 'com.squareup.retrofit2:converter-scalars:2.2.0'  
 testCompile 'junit:junit:4.12'  
}  
  
apply plugin: 'com.google.gms.google-services'

Версии указываются внутри структуры defaultConfig. Дополнительные библиотеки указаны внутри структуры dependencies.

**5** ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

Тестирование приложения является неотъемлемой частью разработки, а также одним из важнейших шагов на пути реализации того или иного функционала в программе. Тестирование программного обеспечения позволяет установить соответствие между реальным поведением программы, и ожидаемым.

**5.1** Верификация нейронной сети

Для верификации нейронной сети необходимо определить критерии оценки результата колоризации. Среди различных критериев оценки можно выделить следующие:

* соответствие результата колоризации оригинальному изображению, до его преобразования в полутоновое (при наличии такового);
* субъективное мнение пользователей; результат колоризации может не соответствовать действительности, но при этом быть приемлемым для пользователя (пример на рисунке 5.1).



Рисунок 5.1 – Пример ошибочной, но приемлемой колоризации

Несмотря на то, что субъективное мнение пользователей, является неплохой оценкой для работы нейронной сети, этот критерий не подходит для данного дипломного проекта. Главным недостатком является тот факт, что для того, чтобы получить адекватную оценку, необходимо набрать достаточное количество уникальных пользователей клиентского приложения, что невозможно без его выпуска, на стадии тестирования. Также стоит заметить, что для использования данного критерия оценки необходимо дополнительно реализовать пользовательский интерфейс, для назначения оценки, что не было запланировано на этапе функционального проектирования.

Соответствие результата колоризации исходному изображению до преобразования в полутоновое больше подходит для этапа тестирования продукта. Для того, чтобы оценка приемлемой, вместо относительно большого количества уникальных пользователей приложения необходимо достаточное количество запросов на колоризацию изображений. Несмотря на тот факт, что для реализации верификации данным способом не требуется построение дополнительного интерфейса для пользователя, необходим дополнительный модуль для оценки колоризации на клиентском приложении. Дополнительный модуль должен реализовывать алгоритм, принимающий исходное изображение, не подверженное никакой обработке, результат колоризации. В качестве результата алгоритм должен возвращать оценку результату колоризации.

Для данной цели был создан размещённый в одноимённом файле класс ColorizationEvaluator, который является частью java-пакета utils клиентского приложения. Данный класс имеет только статические методы и поля, что упрощает его интеграцию в разные части приложения:

public class ColorizationEvaluator {  
  
 private static Bitmap original;  
 private static Bitmap result;  
  
 public static void setOriginal(Bitmap original) {  
 ColorizationEvaluator.original = original;  
 }  
  
 public static void setResult(Bitmap result) {  
 ColorizationEvaluator.result = result;  
 }  
  
 public static Float getEvaluation() {  
 if (original == null || result == null  
 || original.getHeight() != result.getHeight()  
 || original.getWidth() != result.getWidth())  
 return null;  
 int height = original.getHeight();  
 int width = original.getWidth();  
 float evaluationSum = 0;  
 for (int i = 0; i < width; i++)  
 for (int j = 0; j < height; j++) {  
 int originalPixel = original.getPixel(i, j);  
 int resultPixel = result.getPixel(i, j);  
 int originalR = getRChannel(originalPixel);  
 int resultR = getRChannel(resultPixel);  
 int originalG = getGChannel(originalPixel);  
 int resultG = getGChannel(resultPixel);  
 int originalB = getBChannel(originalPixel);  
 int resultB = getBChannel(resultPixel);

evaluationSum += getEvaluationForChannel(originalR, resultR);  
 evaluationSum += getEvaluationForChannel(originalG, resultG);  
 evaluationSum += getEvaluationForChannel(originalB, resultB);  
 }  
 return evaluationSum / (height \* width \* 3);  
 }  
  
 private static float getEvaluationForChannel(int original, int result) {  
 float maxDelta = original > 127 ? original : 255 - original;  
 float error = Math.abs(result - original);  
 return (maxDelta - error) / maxDelta;  
 }  
  
 private static int getRChannel(int pixel) {  
 return (pixel & 0x00FF0000) >> 16;  
 }  
  
 private static int getGChannel(int pixel) {  
 return (pixel & 0x0000FF00) >> 8;  
 }  
  
 private static int getBChannel(int pixel) {  
 return (pixel & 0x000000FF);  
 }  
  
 public static void clearBitmaps() {  
 original = null;  
 result = null;  
 }  
}

Данный класс содержит следующие методы:

* setOriginal – метод для сохранения исходного изображения до его преобразования в полутон; вызывается классом GetBitmapTask после изменения разрешения изображения;
* setResult – метод для сохранения результата колоризации; вызывается классом DownloadingTask после декодирования изображения из строки формата Base64;
* getEvaluation – метод, возвращающий среднюю оценку точности колоризации пикселя по всему изображению. Именно данная величина используется в качестве оценки для колоризации;
* getEvaluationForChannel – инкапсулированный метод, возвращающий оценку точности колоризации для отдельного канала определённого пикселя;
* getRChannel – инкапсулированный метод, возвращающий значение канала, ответственного за наличие красного цвета; использует бинарные операторы;
* getGChannel – инкапсулированный метод, возвращающий значение канала, ответственного за наличие зелёного цвета; использует бинарные операторы;
* getBChannel – инкапсулированный метод, возвращающий значение канала, ответственного за наличие зелёного цвета; использует бинарные операторы;
* clearBitmaps – метод для очистки ссылок на Bitmap-объекты; вызывается после получения оценки классом DownloadingTask.

В качестве оценки точности колоризации изображения используется среднее значение точности колоризации отдельно взятых каналов изображения. Для одного канала, ответственного за наличие цветовой составляющей пикселя, оценка вычисляется как отношение модуля разности оригинального значения и результата колоризации к максимально возможной разности двух каналов.

Для сбора данных используется Firebase Analytics. Так как Firebase Cloud Messaging уже был имплементирован в данный проект, установка сбора данных с помощью Firebase Analytics не потребовало больших временных затрат: достаточно лишь добавить дополнительную зависимость в файл build.gradle:

// ...

dependencies {  
 // ...  
 compile 'com.google.firebase:firebase-core:10.2.4'  
 // ...  
}

// ...

Для отправки новой величины оценки колоризации, необходимо поместить её в объект типа Bundle c ключом VALUE и передать в метод сущности FirebaseAnalytics:

Bundle report = new Bundle();  
report.putFloat(FirebaseAnalytics.Param.VALUE, evaluation);  
FirebaseAnalytics.getInstance(activity)  
 .logEvent(EVENT\_NAME, report);

Для просмотра статистики необходимо перейти в Firebase Console. При выбранном диапазоне дат будут отображены количество событий с определённым именем и суммарная VALUE от их.



Рисунок 5.2 – Фрагмент статистики от Firebase Analytics

Так, например, для первых десяти колоризаций средняя точность составила 88,06% (рисунок 5.2). Недостатком данной статистики является, её время обновления. Максимальное время, через которое возможно обновление статистики в веб-приложении составляет 24 часа по заявлению разработчиков. При получении данных для проекта задержка была равна около 8 часам.

На основании полученных данных средняя величина колоризации составляет 88.03%. Данная оценка составлена на основании 110 результатов колоризации изображений различных объектов. Стоит отметить, что нейронная сеть лучше справляется с колоризацией отдельных объектов, имеющих определённый характерный цвет, текстур дерева, растительности. Худшие результаты даёт колоризация фотографий, имеющих большое количество мелких деталей, колоризация объектов, имеющих редкий цвет.

**5.2** Затраты времени на колоризацию

Для измерения затрат времени на колоризацию производились замеры времени работы метода colorize на стороне сервера. Во процессе экспериментальных замеров времени был установлен факт: время колоризации пропорционально разрешению исходного изображения. Для проведения тестов необходимо было выбрать изображения определённого разрешения. Поэтому в дальнейшем было решено производить замеры времени только на изображениях, сделанных с помощью камеры с разрешением матрицы 3120x4160, уменьшенных в процессе предобработки в 4 раза до разрешения 780x1040. На основании двадцати колоризаций было вычислено среднее её время, равное 4.67 секундам. Наибольшее и наименьшее затраченное на колоризацию время составило соответственно 4.8 и 4.57 секунды.

Затраты времени на загрузку и выгрузку изображений, посылку-приём GCM-сообщения не были измерены, так как данные величины сильно зависят от типа подключения и состояния сети в определённый момент времени.

**5.3** Ручное функциональное тестирование приложения

После реализации клиентского приложения было проведено функциональное тестирование приложения

Функциональные тесты, проведенные над разработанным приложением, представлены в таблице 5.1. Таблица содержит информацию о том, как должно выглядеть тестирование, каким должен быть результат тестирования, а также информацию о том, прошло ли готовое приложение тест.

Таблица 5.1 ­ Тестирование программы

|  |  |  |
| --- | --- | --- |
| Содержание теста | Ожидаемый результат | Тест пройден |
| Приложение запущено, но исходное изображение не выбрано. | Изображение не отображено. По нажатию кнопки «LOAD» новая активность не запускается | Да |
| Активность для получения фотографии с помощью камеры закрыта пользователем, изображение не выбрано | Изображение не отображено. По нажатию кнопки «LOAD» новая активность не запускается | Да |
| Активность для получения изображения из галереи закрыта пользователем, изображение не выбрано | Изображение не отображено. По нажатию кнопки «LOAD» новая активность не запускается | Да |

*Продолжение таблицы 5.1*

|  |  |  |
| --- | --- | --- |
| Содержание теста | Ожидаемый результат | Тест пройден |
| Активность для получения фотографии с помощью камеры закрыта пользователем, изображение выбрано | Изображение отображено. По нажатию кнопки «LOAD» запускается новая активность | Да |
| Активность для получения изображения из галереи закрыта пользователем, изображение выбрано | Изображение отображено. По нажатию кнопки «LOAD» запускается новая активность | Да |
| Колоризация начата, пользователь нажимает кнопку «BACK» на экране | Активность закрывается, последнее изображение отображено на экране | Да |
| Колоризация начата, пользователь нажимает кнопку «BACK» на устройстве | Активность закрывается, последнее изображение отображено на экране | Да |
| Колоризация начата, но отсутствует подключение к Интернету | На экран выводится сообщение об ошибке | Да |
| Колоризация начата, но сервер недоступен | На экран выводится сообщение об ошибке | Да |
| Колоризация начата, проблем с соединением нет, сервер запущен | Через определённое время на экране отображается результат колоризации | Да |

Как можно увидеть из представленной таблицы приложение успешно проходит все запланированные функциональные тесты.

**6** РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

В данном разделе описан как процесс установки и настройки серверной стороны программного средства колоризации изображения, так и процесс создания установочного файла клиентского приложения для платформы Android. При описании процесса установки и настройки серверной части будет описан более простой способ, использующий готовый Docker-образ. Также возможен более сложный вариант, предполагающий установку и настройку официального Docker-образа с предустановленным Caffe.

**6.1** Установка Docker

Перед выполнением дальнейших пунктов необходимо установить ПО Docker, которое можно загрузить с официального сайта разработчика.

Текущими поддерживаемыми настольными операционными системами являются:

* Mac;
* Windows.

Docker поддерживает следующие серверные ОС:

* Windows Server;
* CentOS;
* Debian;
* Fedora;
* Oracle Linux;
* RHEL;
* SLES;
* Ubuntu.

Также присутствует поддержка облачных платформ:

* AWS;
* Azure.

В данном руководстве будет описан процесс установки Docker для Windows 10.

Для установки Docker для операционной системы Windows 10 необходимо загрузить из Docker Store msi-файл для программного обеспечения Windows Installer и запустить его. Данное действие запустит процесс установки Docker. После установки программного обеспечения возможно потребуется перезагрузка операционной системы. Следует также отметить что Docker использует Hyper-V - систему аппаратной виртуализации для x64-систем на основе гипервизора, что может повлиять на другое ПО установленное на компьютере.

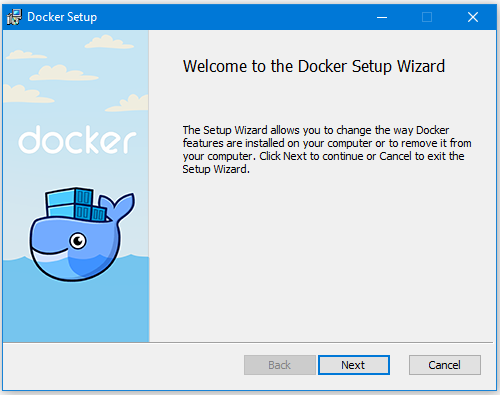


Рисунок 6.1 – Окно установки ПО Docker

После установки Docker логотип этого программного обеспечения появится в системном трее (см. рисунок 6.2).

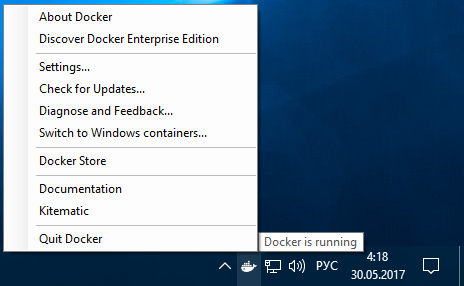


Рисунок 6.2 – Логотип Docker в системном трее

Для верификации установки необходимо выполнить команду docker version с помощью CLI-интерфейса системы.

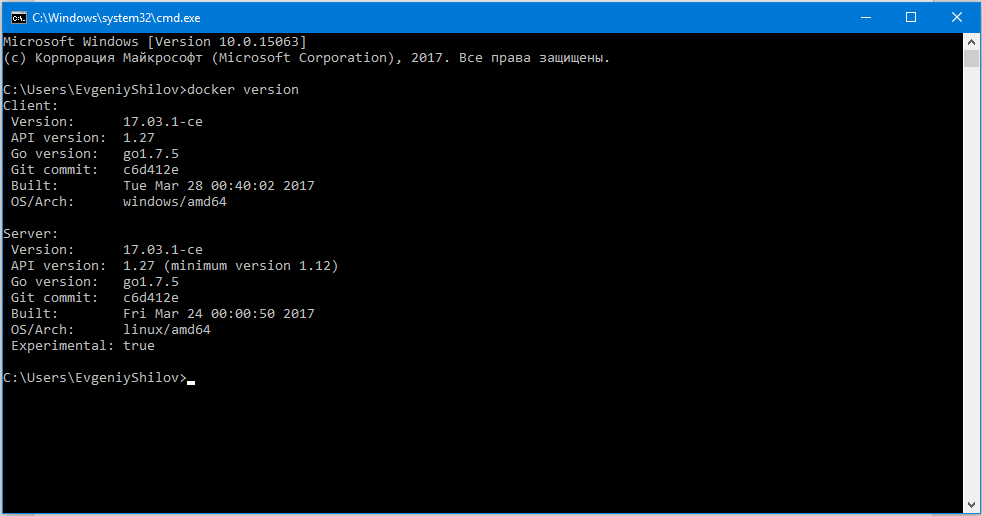


Рисунок 6.3 – Пример выполнения команды docker version

**6.2** Установка сервера из сконфигурированного образа.

В данном подразделе описана установка и настройка серверной части из подготовленного Docker-образа. Для этого необходимо выполнить команду docker pull, указав адрес Docker Hub репозитория.

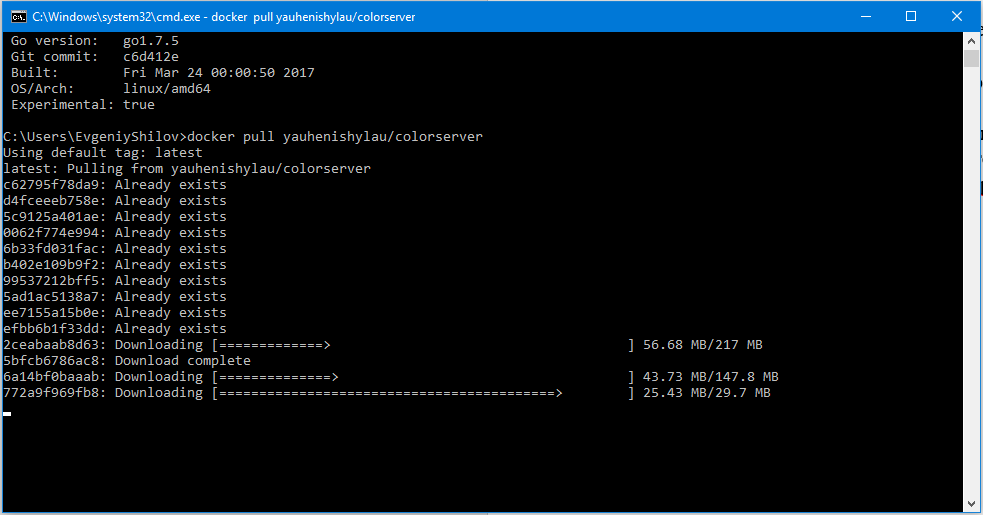


Рисунок 6.4 – Пример выполнения команды docker pull

В качестве адреса удалённого репозитория необходимо указать yauhenishylau/colorserver. Данная команда загрузит образ, содержащий все необходимые исходные файлы. После загрузки образа, перед созданием контейнера необходимо узнать, адрес IP-адрес интерфейса, по которому клиентское приложение будет получать доступ к серверу. Для этого в ОС Windows необходимо выполнить команду ipconfig (см рисунок 6.5).

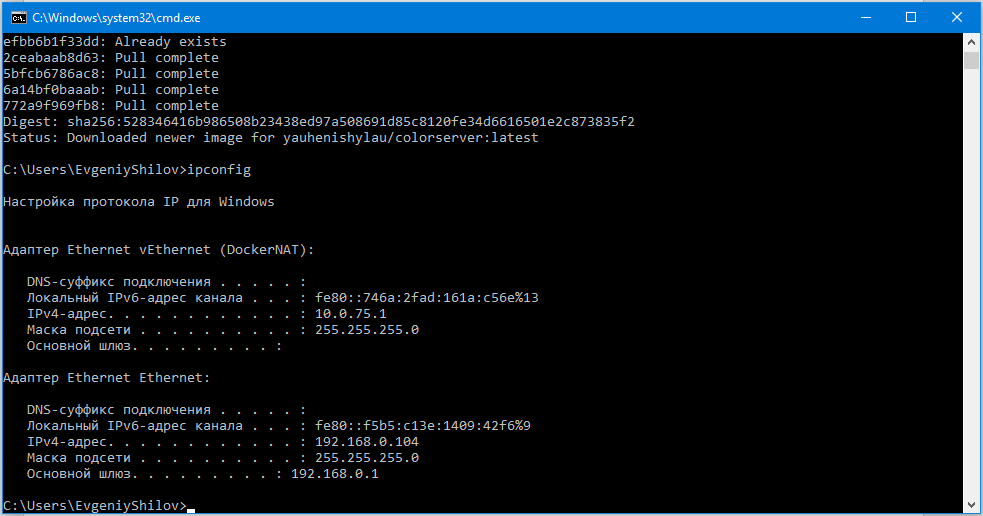


Рисунок 6.5 – Пример выполнения команды ipconfig

Следующим шагом является создание Docker-контейнера при помощи команды docker run. При этом необходимо добавить флаг –p для того, чтобы связать порт ОС-хоста с внутренним портом контейнера.

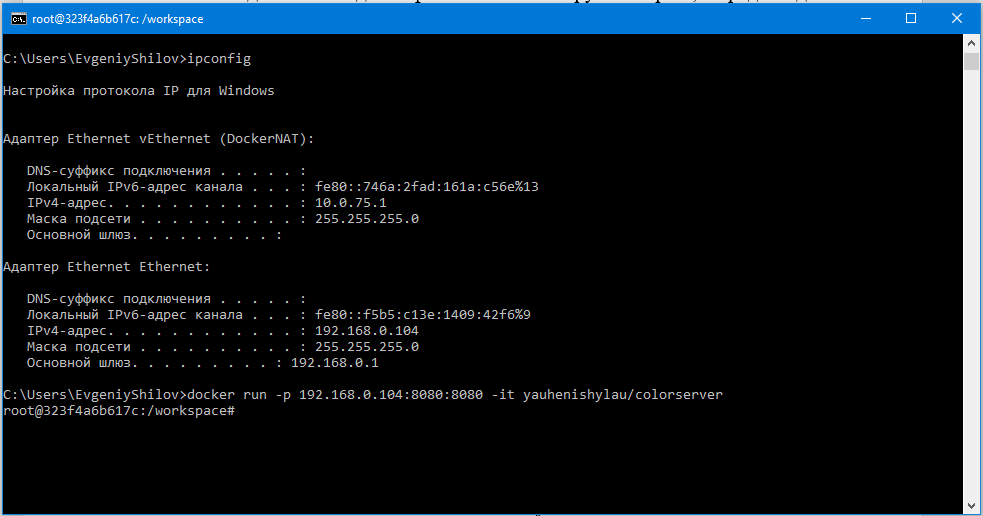


Рисунок 6.6 – Создание контейнера Docker при помощи команды docker run

После создания контейнера необходимо запустить файл server.py (см. рисунок 6.7).

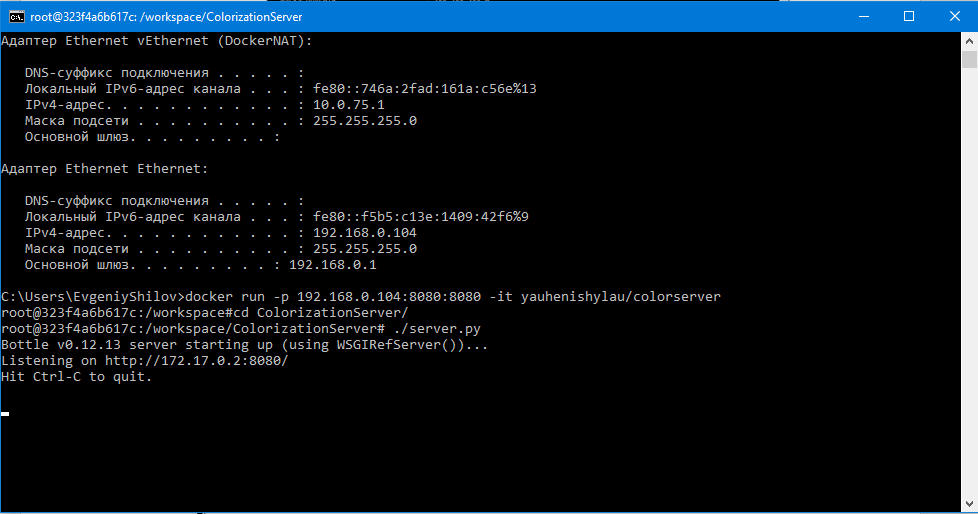


Рисунок 6.7 – Запуск сервера

После выполнения данной команды сервер готов к использованию. Доступ к серверу можно получить по паре адрес-порт указанному при создании контейнера. Завершить работу контейнера можно командой exit, выполнив её внутри контейнера. При завершении работы контейнера, его можно в дальнейшем перезапустить при помощи команды docker start, указав при этом имя созданного контейнера. Для того, чтобы узнать имя контейнера необходимо выполнить команду docker container ls (см рисунок 6.8).

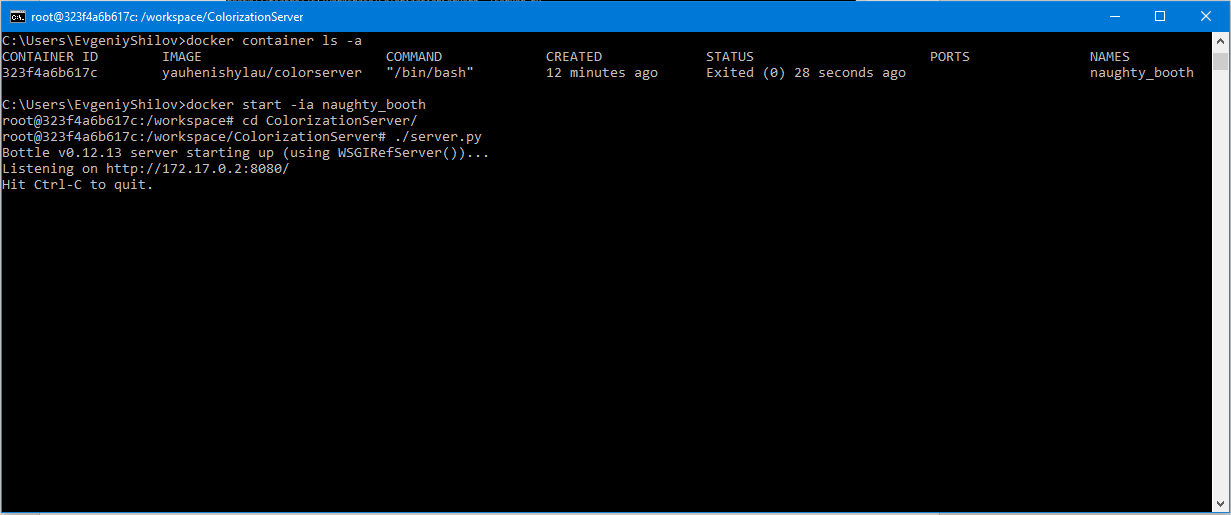


Рисунок 6.8 – Перезапуск сервера

**6.3** Создание установочного файла для платформы Android

Для выполнения данного шага необходимо установить следующее ПО:

* Java Developer Kit – ПО компании Oracle, необходимое для разработки программ для JVM;
* Android Studio – IDE для создания приложений для платформы Android;
* Android SDK – комплект средств разработки, устанавливаемый вместе со средой разработки Android Studio.

Для создания APK-файла необходимо запустить Android Studio и выбрать пункт «Open an existing Android Studio project». Затем необходимо указать путь к каталогу с исходным кодом проекта (см рисунок 6.9). Исходный код проекта содержится в архиве на носителе, прикреплённом к данному дипломному проекту. Структура проекта должна соответствовать стандартной структуре проекта Android Studio. Кроме исходных файлов проект содержит файлы конфигурации, необходимые как для среды разработки, так и для самого проекта.

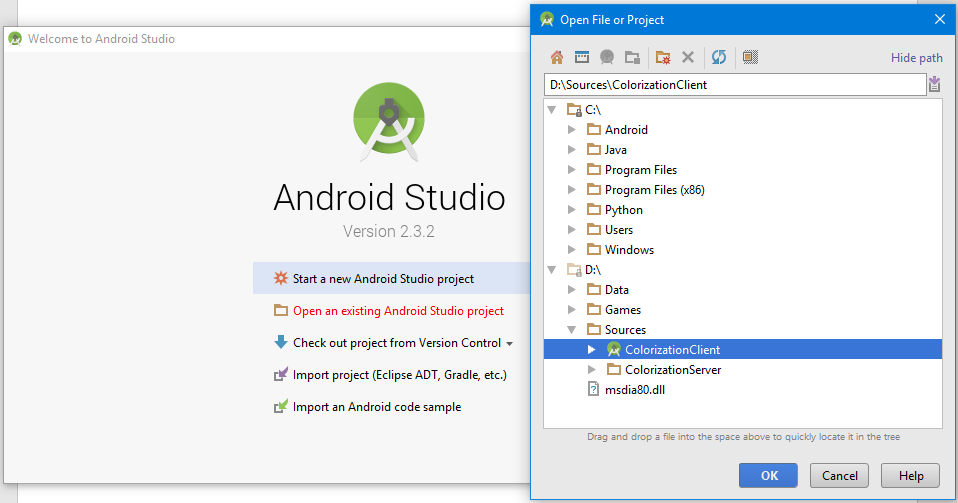


Рисунок 6.9 – Открытие проекта Android Studio

После открытия проекта подобным способом система сборки Gradle сгенерирует недостающие конфигурационные файлы. После этого, как Gradle построит необходимые конфигурационные файлы, необходимо указать адрес сервера в классе API. Данный класс размещён в пакете api, который должен быть размещён в каталоге проекта Android Studio, который размещён по пути srс\main\java\by\evgeniyshilov\colorizationclient\api.

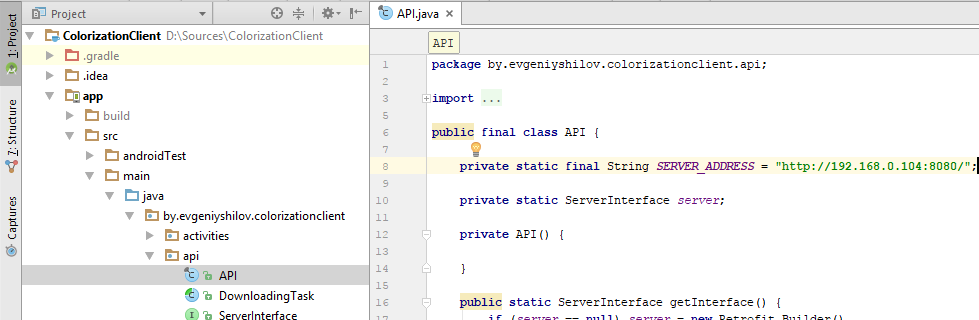


Рисунок 6.10 – Расположение файла API.java

В данном файле для указания адреса сервера используется строка SERVER\_ADDRESS. После изменения адреса сервера можно приступить к сборке APK-файла. Для этого необходимо выбрать пункт меню Build и в появившемся контекстном меню выбрать пункт Build APK.

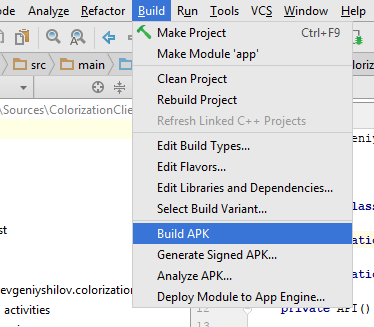


Рисунок 6.11 – Выбор пункта меню для построения APK-файла

Данное действие приведёт к построению APK файла. После завершения пользователь получит уведомление об успешном окончании генерации, а сам файл появится в каталоге app\build\outputs\apk проекта.

**6.4** Руководство по использованию Android-приложения

После установки APK-файла на устройство, необходимо запустить приложение из основного меню системы.

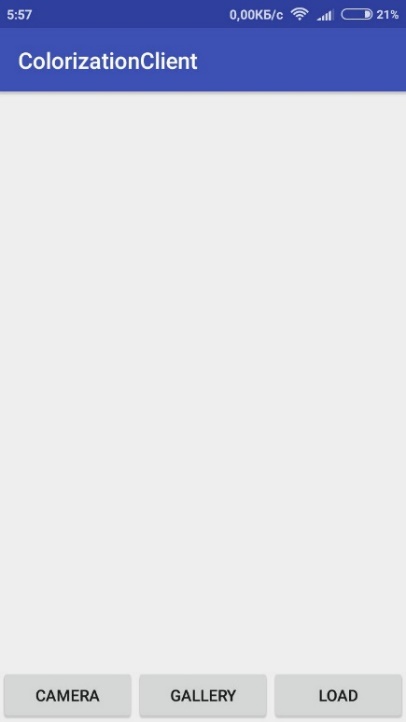


Рисунок 6.12 – Экран для выбора изображения

На данном экране пользователю доступны три кнопки (см. рисунок 6.12). Кнопка «LOAD» не приводит к выполнению какого-либо действия, пока пользователь не выберет изображение.

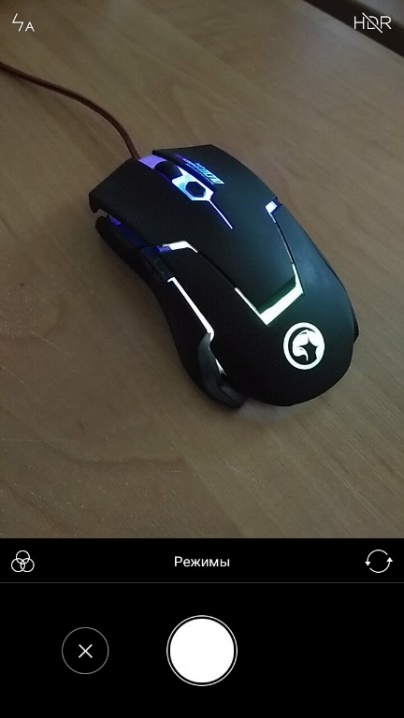


Рисунок 6.13 – Получение изображения с помощью камеры устройства.

После того, как пользователь выберет изображение, его полутоновая версия будет отображена на экране (см. рисунок 6.14)



Рисунок 6.14 – Просмотр полутоновой версии выбранного изображения

Если пользователь сочтёт нужным выбрать другое изображение, то он может снова воспользоваться кнопками «CAMERA» и «GALLERY». В противном случае пользователь должен нажать кнопку «LOAD» для начала колоризации.



Рисунок 6.15 – Экран ожидания результата колоризации

После окончания колоризации индикатор загрузки будет скрыт, а на экране будет отображено обработанное изображение (см рисунок 6.16).



Рисунок 6.16 – Отображение результата колоризации

Для колоризации другого изображения пользователю необходимо нажать на кнопку «BACK» или соответствующий контроллер своего устройства. Для выхода из приложения пользователю необходимо повторно нажать на клавишу «BACK» своего устройства.