Приложение 1

**Общество с ограниченной ответственностью «СК-Роботикс»**

**(ООО «СК-Роботикс»)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | УТВЕРЖДАЮ |
|  |  |  |  | Генеральный директор |
|  |  |  |  | Шумов М.Б. |
|  |  |  |  | 16.01 2023 г.  М.П. |

**ПРОГРАММА ОБНАРУЖЕНИЯ И КЛАССИФИКАЦИИ ДЕФЕКТОВ**

**«DEFECTOSCOPE»**

**ТЕКСТ ПРОГРАММЫ**

**ЛИСТ УТВЕРЖДЕНИЯ**

**РОФ.** **93241908.01-01 12 01-ЛУ**

**Листов 2**

| Наименование организации | Должность исполнителя | Ф.И.О. | Подпись | Дата |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

УТВЕРЖДЕН

РОФ. 93241908.01-01 12 01

**ПРОГРАММА ОБНАРУЖЕНИЯ И КЛАССИФИКАЦИИ ДЕФЕКТОВ**

**«DEFECTOSCOPE»**

**ТЕКСТ ПРОГРАММЫ**

**РОФ.** **93241908.01-01 12 01**

**Листов 45**

Блок формирования отчетов Reports.

Файл **main.css**.

body {

padding: 10px;

background-color: #ccc;

}

#main-container {

position: relative;

top: 200px;

}

.chosen-container {

font-size: 18px;

}

.chosen-container-single .chosen-single {

height: 38px;

line-height: 38px;

}

.chosen-container-single .chosen-single div b {

background: url("chosen-sprite.png") no-repeat 2px 9px;

}

.chosen-container-active.chosen-with-drop .chosen-single div b {

background-position: -16px 9px;

}

.chosen-container .chosen-results li {

line-height: 20px;

}

.chosen-container-single .chosen-single abbr {

top: 13px;

}

Файл **scripts.js**.

var $select = $('#plane-selector');

var $report\_btn = $('#get-report-btn');

var list = [];

$select.chosen({

width: '100%',

no\_results\_text: 'Совпадений не обнаружено',

search\_contains: true,

placeholder\_text\_single: 'Выберите или введите параметр для поиска',

allow\_single\_deselect: true

});

$(document).ready(function() {

$.ajax({

url: '/all',

type: 'POST',

success: function (response) {

list = JSON.parse(response);

$.each( list['name'], function(i, value) {

var $item = '<option value="'+ i +'">'+ value +' #'+ list['serial'][i] +'</option>';

$select.append($item);

});

$select.trigger('chosen:updated');

}

});

});

$report\_btn.on('click', function() {

var id = $select.val();

if (id) {

$.ajax({

url: '/report',

type: 'POST',

contentType: 'application/json',

data: JSON.stringify({

'name': list['name'][id],

'serial': list['serial'][id]

}),

success: function () {

}

});

}

});

Файл **main.html**.

<html>

<head>

<meta http-equiv="content-type" content="text/html; charset=utf-8">

<link rel="stylesheet" type="text/css" href="{{ url\_for('static', filename='css/chosen.css') }}">

<link rel="stylesheet" type="text/css" href="{{ url\_for('static', filename='css/font-awesome.min.css') }}">

<link rel="stylesheet" type="text/css" href="{{ url\_for('static', filename='css/bootstrap/css/bootstrap.css') }}">

<link rel="stylesheet" type="text/css" href="{{ url\_for('static', filename='css/main.css') }}">

<title></title>

<script src="{{ url\_for('static', filename='lib/jquery-3.3.1.min.js') }}"></script>

<script src="{{ url\_for('static', filename='lib/chosen.jquery.min.js') }}"></script>

</head>

<body>

<div id="main-container" class="container">

<div class="row">

<div class="col"></div>

<div class="col-6">

<select id="plane-selector" class="chosen-select">

<option></option>

</select>

</div>

<div class="col-2">

<button id="get-report-btn" type="button" class="btn btn-primary">Создать отчет</button>

</div>

<div class="col"></div>

</div>

</div>

</body>

<script src="{{ url\_for('static', filename='js/scripts.js') }}"></script>

</html>

Файл **\_\_init\_\_\_.py**.

from flask import Flask

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

from app import views

Файл **views.py**.

from flask import render\_template, request

from database import defects\_base

from app import app

import json

@app.route('/')

@app.route('/main')

def main():

return render\_template("main.html")

@app.route('/all', methods=['POST'])

def all():

defect\_database = defects\_base.DefectsBase('database\defects\_base.db')

plane\_list = defect\_database.all()

defect\_database.close()

return json.dumps(plane\_list)

@app.route('/report', methods=['POST'])

def report():

data = request.get\_json()

defect\_database = defects\_base.DefectsBase('database\defects\_base.db')

defect\_database.report('reports', data['name'], data['serial'])

defect\_database.close()

return ''

Файл **defects\_base.py**.

import datetime

import os.path

import pickle

import sqlite3

import cv2

from reportlab.lib.pagesizes import A4

from reportlab.lib.utils import ImageReader

from reportlab.pdfbase import pdfmetrics

from reportlab.pdfbase.pdfmetrics import stringWidth

from reportlab.pdfbase.ttfonts import TTFont

from reportlab.pdfgen import canvas

from database import defects\_list

def add\_text\_to\_pdf\_center(canvas, text, y):

width, height = A4

text\_width = stringWidth(text, fontName='GOST', fontSize=14)

pdf\_text\_object = canvas.beginText((width - text\_width) / 2.0, y)

pdf\_text\_object.textOut(text)

def add\_text\_to\_pdf\_left(canvas, text, x, y):

pdf\_text\_object = canvas.beginText(x, y)

pdf\_text\_object.textOut(text)

def footer(canvas, date, width, page):

canvas.drawCentredString(width / 2, 40, '{}.{}.{}'.format(date.day,

date.month,

date.year))

canvas.drawCentredString(width / 2, 60, 'ООО СК-"Роботикс"')

canvas.drawCentredString(width / 2, 20, 'стр. {}'.format(page))

canvas.line(width / 2 - 150, 53, width / 2 + 150, 53)

class DefectsBase:

def \_\_init\_\_(self, path\_to\_base='defects\_base.db'):

try:

self.db\_connection = sqlite3.connect(path\_to\_base)

self.cursor = self.db\_connection.cursor()

sql\_req = 'CREATE TABLE IF NOT EXISTS airplanes (id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT DEFAULT 0,' \

'name TEXT NOT NULL,' \

'serial TEXT NOT NULL,' \

'comment TEXT DEFAULT "",' \

'UNIQUE (name, serial) ON CONFLICT IGNORE);'

self.cursor.execute(sql\_req)

sql\_req = 'CREATE TABLE IF NOT EXISTS defects (id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT DEFAULT 0,' \

'airplane\_name TEXT NOT NULL,' \

'air\_plane\_serial TEXT NOT NULL,' \

'date TIMESTAMP NOT NULL,' \

'defect\_data BLOB,' \

'comment TEXT DEFAULT "");'

self.cursor.execute(sql\_req)

self.db\_connection.commit()

self.cursor.close()

except sqlite3.Error as e:

print('Error opening/creating database', e)

def all(self):

self.cursor = self.db\_connection.cursor()

sql\_req = 'SELECT name, serial FROM airplanes;'

self.cursor.execute(sql\_req)

data = self.cursor.fetchall()

result = {

'name': [],

'serial': [],

}

for dat in data:

name\_, serial\_ = dat

result['name'].append(name\_)

result['serial'].append(serial\_)

self.cursor.close()

return result

def add(self, aircraft\_defects\_list: defects\_list.AirCraftDefectsList):

self.cursor = self.db\_connection.cursor()

sql\_req = 'INSERT OR IGNORE INTO airplanes (name, serial) VALUES (?,?);'

self.cursor.execute(sql\_req, (aircraft\_defects\_list.name, aircraft\_defects\_list.serial\_num))

date\_now = datetime.datetime.now()

for defect in aircraft\_defects\_list.defects:

sql\_req = 'INSERT INTO defects (airplane\_name, air\_plane\_serial, date, defect\_data, comment) ' \

'VALUES (?, ?, ?, ?, ?);'

jpeg\_defect = defect

\_, jpeg\_defect.img = cv2.imencode('.jpg', jpeg\_defect.image)

self.cursor.execute(sql\_req, (aircraft\_defects\_list.name,

aircraft\_defects\_list.serial\_num,

aircraft\_defects\_list.date,

pickle.dumps(jpeg\_defect),

''))

self.db\_connection.commit()

self.cursor.close()

def get(self, aircraft\_name, aircraft\_serial):

self.cursor = self.db\_connection.cursor()

sql\_req = 'SELECT MAX(date) FROM defects WHERE airplane\_name=? AND air\_plane\_serial=?;'

self.cursor.execute(sql\_req, (aircraft\_name, aircraft\_serial))

date, = self.cursor.fetchone()

sql\_req = 'SELECT defect\_data FROM defects WHERE airplane\_name=? AND air\_plane\_serial=? AND date=?;'

self.cursor.execute(sql\_req, (aircraft\_name, aircraft\_serial, date))

defects\_resp = self.cursor.fetchall()

date = datetime.datetime.strptime(date.split('.')[0], '%Y-%m-%d %H:%M:%S')

air\_craft = defects\_list.AirCraftDefectsList(aircraft\_serial, aircraft\_name)

air\_craft.date = date

for defect\_resp in defects\_resp:

defect\_pickled, = defect\_resp

defect = pickle.loads(defect\_pickled)

defect.img = cv2.imdecode(defect.img, cv2.IMREAD\_COLOR)

air\_craft.defects.append(defect)

self.cursor.close()

return air\_craft

def report(self, report\_path, aircraft\_name, aircraft\_serial):

defects\_table = {'Class\_1': 'не определен', 'Class\_2': '"трещины"', 'Class\_3': '"риски"', 'Class\_4': '"вмятины"',

'Class\_5': '"коррозия"'}

air\_craft = self.get(aircraft\_name, aircraft\_serial)

pdfmetrics.registerFont(TTFont('GOST', 'database\\GOSTtypeB.ttf'))

pdf\_canvas = canvas.Canvas(os.path.join(report\_path,

'air\_craft\_{}\_serial\_{}.pdf'.format(aircraft\_name, aircraft\_serial),

), pagesize=A4)

width, height = A4

pdf\_canvas.setLineWidth(3)

pdf\_canvas.setFont('GOST', size=14)

report\_day = air\_craft.date

page\_num = 1

footer(pdf\_canvas, report\_day, width, page\_num)

pdf\_canvas.drawCentredString(width / 2, 755, 'Отчет по ВС {}'.format(air\_craft.name))

pdf\_canvas.drawCentredString(width / 2, 740, 'серийный номер {}'.format(air\_craft.serial\_num))

pdf\_canvas.line(width / 2 - 240, 732, width / 2 + 240, 732)

pdf\_canvas.line(width / 2 - 230, 730, width / 2 + 230, 730)

total\_defects = [len(defect.types) for defect in air\_craft.defects]

count\_defects = sum(total\_defects)

pdf\_canvas.drawString(75, 650, 'Обнаружено дефектов: {} на {} кадрах'.format(count\_defects, len(total\_defects)))

pdf\_canvas.showPage()

for defect in air\_craft.defects:

pdf\_canvas.setFont('GOST', size=14)

cv2.imwrite('temp.jpg', cv2.resize(defect.img, (int(width) - 100, int(height / 2))))

pic = ImageReader('temp.jpg')

pdf\_canvas.drawImage(pic, 50, 100)

uniq\_defects = {i: defect.types.count(i) for i in defect.types}

str\_defects = []

for defect\_type in uniq\_defects.keys():

str\_defects.append('дефектов типа {} - {}'.format(defects\_table[defect\_type],

uniq\_defects[defect\_type]))

pdf\_canvas.drawString(75, 710, 'Обнаружено дефектов:')

y = 710 - 20

for line in str\_defects:

pdf\_canvas.drawString(100, y, '{}'.format(line))

y -= 20

page\_num += 1

footer(pdf\_canvas, report\_day, width, page\_num)

pdf\_canvas.showPage()

pdf\_canvas.save()

def close(self):

self.db\_connection.close()

Файл **defects\_listp.py**.

import datetime

class DefectFrame:

def \_\_init\_\_(self, image, boxes, scores, types):

self.image = image

self.boxes = boxes

self.scores = scores

self.types = types

class AirCraftDefectsList:

def \_\_init\_\_(self, serial\_num='0000', name='plane'):

self.defects = []

self.serial\_num = serial\_num

self.name = name

self.date = datetime.datetime.now()

def add(self, defect):

self.defects.append(defect)

блок ВИДЕОСЕРВЕРА

#ifndef CAMERA\_PROG\_CAMERA\_H

#define CAMERA\_PROG\_CAMERA\_H

#include "FIFO.h"

#include <opencv2/imgproc/imgproc.hpp>

#include <opencv2/highgui/highgui.hpp>

#include <opencv2/features2d.hpp>

#include <opencv2/opencv.hpp>

#include <thread>

#include <map>

namespace video {

class Camera {

public:

Camera(int camera, cv::Size resolution, utils::FIFO<cv::Mat>\* record\_queue);

Camera(int camera, cv::Size resolution);

virtual ~Camera();

void add\_sink(int id, utils::FIFO<cv::Mat>\* queue);

void remove\_sink(int id);

private:

void run\_loop();

std::thread loop\_thread\_;

int camera\_;

cv::Size resolution\_;

bool stop\_;

bool record\_;

std::map<int, utils::FIFO<cv::Mat>\*> sinks;

utils::FIFO<cv::Mat>\* record\_queue\_;

};

Camera::~Camera() {

stop\_ = true;

if (loop\_thread\_.joinable()) {

loop\_thread\_.join();

}

}

Camera::Camera(int camera, cv::Size resolution) {

camera\_ = camera;

resolution\_ = resolution;

stop\_ = false;

record\_ = false;

record\_queue\_ = nullptr;

loop\_thread\_ = std::thread ([this]{ run\_loop(); });

}

Camera::Camera(int camera, cv::Size resolution, utils::FIFO<cv::Mat> \*record\_queue) {

camera\_ = camera;

resolution\_ = resolution;

stop\_ = false;

record\_ = true;

record\_queue\_ = record\_queue;

loop\_thread\_ = std::thread ([this]{ run\_loop(); });

}

void Camera::run\_loop() {

cv::VideoCapture cap("v4l2src device=/dev/video" + std::to\_string(camera\_) +

" do-timestamp=true ! video/x-raw, width=" + std::to\_string(resolution\_.width) +

", height=" + std::to\_string(resolution\_.height) +

", framerate=30/1, format=YUY2 ! videoscale ! video/x-raw, width=960, height=540 ! " +

"videoconvert ! video/x-raw, format=BGR ! appsink");

if (!cap.isOpened()) {

std::cerr << "Error: Can't open camera!" << std::endl;

exit(EXIT\_FAILURE);

}

cv::Mat frame;

cv::Mat trash\_frame;

while (!stop\_) {

if (!cap.read(frame)) {

continue;

}

for (auto const& sink : sinks) {

if (sink.second->full()) {

sink.second->pop(trash\_frame);

}

sink.second->push(frame);

}

if (record\_) {

if (record\_queue\_->full()) {

record\_queue\_->pop(trash\_frame);

}

record\_queue\_->push(frame);

}

}

cap.release();

}

void Camera::add\_sink(int id, utils::FIFO<cv::Mat>\* queue) {

if (queue != nullptr) {

sinks[id] = queue;

}

}

void Camera::remove\_sink(int id) {

sinks.erase(id);

std::cout << std::to\_string(sinks.size()) << " sinks remain" << std::endl;

}

} // video

#endif //CAMERA\_PROG\_CAMERA\_H

Файл **FIFO.h**.

#ifndef ALGOCPP\_FIFO\_H

#define ALGOCPP\_FIFO\_H

// based on https://stackoverflow.com/a/16075550

#include <queue>

#include <mutex>

#include <condition\_variable>

#include <iostream>

namespace utils {

template <typename T>

class FIFO {

public:

explicit FIFO(bool blocking = false, size\_t size = 10);

~FIFO();

bool push(T item);

bool pop(T &item);

bool empty();

bool full();

private:

std::queue<T> queue\_;

mutable std::mutex mutex\_;

std::condition\_variable flag\_;

bool blocking\_;

size\_t max\_size\_;

};

template<typename T>

FIFO<T>::FIFO(bool blocking, size\_t size) : mutex\_(), flag\_() {

blocking\_ = blocking;

max\_size\_ = size;

queue\_ = std::queue<T>();

};

template<typename T>

FIFO<T>::~FIFO() = default;

template<typename T>

bool FIFO<T>::push(T item) {

std::lock\_guard<std::mutex> lock(mutex\_);

if (queue\_.size() == max\_size\_) {

return false;

}

queue\_.push(item);

flag\_.notify\_one();

return true;

}

template<typename T>

bool FIFO<T>::pop(T &item) {

if (!blocking\_ && queue\_.empty()) {

return false;

}

std::unique\_lock<std::mutex> lock(mutex\_);

while (queue\_.empty()) {

flag\_.wait(lock);

}

item = queue\_.front();

queue\_.pop();

return true;

}

template<typename T>

bool FIFO<T>::empty() {

return queue\_.size() == 0;

}

template<typename T>

bool FIFO<T>::full() {

return queue\_.size() == max\_size\_;

}

} // utils

#endif //ALGOCPP\_FIFO\_H

Файл **utils.h**.

#ifndef ALGOCPP\_UTILS\_H

#define ALGOCPP\_UTILS\_H

#include <chrono>

#include <thread>

namespace utils {

void delay(long long ms);

long long unsigned get\_us();

}

#endif //ALGOCPP\_UTILS\_H

Файл **utils.cpp**.

#include "utils.h"

namespace utils {

void delay(long long ms) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(ms));

}

long long unsigned get\_us() {

return std::chrono::duration\_cast<std::chrono::microseconds>(

std::chrono::high\_resolution\_clock::now().time\_since\_epoch()

).count();

}

}

Файл **ClientWorker.h**.

#ifndef CAMERA\_PROG\_CLIENTWORKER\_H

#define CAMERA\_PROG\_CLIENTWORKER\_H

#include <iostream>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <unistd.h>

#include <string>

#include <arpa/inet.h>

#include <cstring>

#include <cstdio>

#include <thread>

#include <vector>

#include "base64.h"

#include "utils.h"

#include "FIFO.h"

#include <opencv4/opencv2/imgproc/imgproc.hpp>

#include <opencv4/opencv2/highgui/highgui.hpp>

#include <opencv4/opencv2/features2d.hpp>

#include <opencv4/opencv2/opencv.hpp>

#include "opencv4/opencv2/core/core.hpp"

namespace server {

int worker\_id = 0;

class ClientWorker {

public:

ClientWorker(int client, sockaddr\_in addr);

virtual ~ClientWorker();

void run\_loop();

bool isAlive() const;

utils::FIFO<cv::Mat>\* stream();

int id;

private:

utils::FIFO<cv::Mat> queue\_;

int client\_;

sockaddr\_in client\_addr\_{};

std::thread worker\_;

bool stopped\_;

bool stop\_;

};

ClientWorker::ClientWorker(int client, sockaddr\_in addr) : queue\_(true, 10) {

id = worker\_id;

worker\_id++;

client\_ = client;

client\_addr\_ = addr;

stopped\_ = false;

stop\_= false;

worker\_ = std::thread ([this]{run\_loop();});

}

void ClientWorker::run\_loop() {

char buffer[1024 \* 1024];

char ip[INET\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &(client\_addr\_.sin\_addr), ip, INET\_ADDRSTRLEN);

std::cout << ip << std::endl;

struct timeval timeout{};

timeout.tv\_sec = 10;

timeout.tv\_usec = 0;

setsockopt (client\_, SOL\_SOCKET, SO\_RCVTIMEO, (char \*)&timeout,

sizeof(timeout));

std::vector<uchar> buff;//buffer for coding

std::vector<int> param(4);

cv::Mat src = cv::imread("test.jpg");

cv::Mat frame;

cv::resize(src, frame, cv::Size(960, 540));

param[0] = cv::IMWRITE\_JPEG\_QUALITY;

param[1] = 40;//default(95) 0-100

param[2] = cv::IMWRITE\_JPEG\_OPTIMIZE;

param[3] = 1;

std::string encoded\_frame\_str;

std::cout << "Start communication with " << ip << std::endl;

while (!stop\_) {

memset(buffer, 0, 1000);

long n = recv(client\_, buffer, 100, 0);

if (n < 0) {

std::cout << "-1" << std::endl;

continue;

}

if (n == 0) {

std::cout << "disconnected client by timeout: " << ip << std::endl;

break;

}

auto str = std::string(buffer);

//randu(frame, cv::Scalar(0, 0, 0), cv::Scalar(255, 255, 255));

queue\_.pop(frame);

cv::imencode(".jpg", frame, buff, param);

auto \*encoded\_frame = reinterpret\_cast<unsigned char\*>(buff.data());

encoded\_frame\_str = R"({"image": ")" + base64\_encode(encoded\_frame, buff.size()) +

R"(", "timestamp": )" + std::to\_string(utils::get\_us()) + "}\r\n";

send(client\_, encoded\_frame\_str.c\_str(), strlen(encoded\_frame\_str.c\_str()), 0);

if (str == "quit") {

break;

}

if (stop\_) {

break;

}

}

close(client\_);

std::cout << "disconnected client: " << ip << std::endl;

std::cout.flush();

stopped\_ = true;

}

ClientWorker::~ClientWorker(){

std::cout << "deleted" << std::endl;

stop\_ = true;

worker\_.join();

}

bool ClientWorker::isAlive() const {

return !stopped\_;

}

utils::FIFO<cv::Mat> \*ClientWorker::stream() {

return &queue\_;

}

} // server

#endif //CAMERA\_PROG\_CLIENTWORKER\_H

Файл **main.cpp**.

#include <iostream>

#include "ClientWorker.h"

#include <map>

#include <algorithm>

#include <csignal>

#include "Camera.h"

#include "Recorder.h"

const auto SERVER\_PORT = htons(8008);

volatile sig\_atomic\_t exit\_flag = 0;

video::Recorder recorder("log\_video.ts");

video::Camera camera(4, cv::Size(1920, 1080), &recorder.stream);

void clearDeadWorkers(std::map<int, server::ClientWorker\*> &list){

std::vector<int> ids;

for (auto const& w : list) {

if (!w.second->isAlive()) {

auto ww = w.second;

auto id = ww->id;

camera.remove\_sink(id);

ids.push\_back(id);

ids.push\_back(id);

}

}

for (auto x : ids) {

list.erase(x);

}

}

void clearAllWorkers(std::map<int, server::ClientWorker\*> &list){

std::vector<int> ids;

for (auto const& w : list) {

auto id = w.second->id;

camera.remove\_sink(id);

ids.push\_back(id);

delete w.second;

list.erase(list.find(id));

}

for (auto x : ids) {

list.erase(x);

}

}

void exit\_catch(int sig) {

std::cout << "User stop requested: " << std::to\_string(sig) << std::endl;

exit\_flag = 1;

}

void setBreakCallback() {

struct sigaction sigIntHandler{};

sigIntHandler.sa\_handler = exit\_catch;

sigemptyset(&sigIntHandler.sa\_mask);

sigIntHandler.sa\_flags = 0;

sigaction(SIGINT, &sigIntHandler, nullptr);

}

void buffer\_append\_int32(uint8\_t\* buffer, int32\_t number, int32\_t \*index) {

buffer[(\*index)++] = number >> 24;

buffer[(\*index)++] = number >> 16;

buffer[(\*index)++] = number >> 8;

buffer[(\*index)++] = number;

}

int main() {

setBreakCallback();

int serverSock=socket(AF\_INET, SOCK\_STREAM, 0);

sockaddr\_in serverAddr{};

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_port = SERVER\_PORT;

serverAddr.sin\_addr.s\_addr = INADDR\_ANY;

int reusePort = 1;

setsockopt(serverSock, SOL\_SOCKET, SO\_REUSEPORT, &reusePort, sizeof(reusePort));

if (bind(serverSock, (struct sockaddr\*)&serverAddr, sizeof(struct sockaddr)) < 0) {

std::cout << "Error: bind" << std::endl;

abort();

}

listen(serverSock, 4);

char ip[INET\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &(serverAddr.sin\_addr), ip, INET\_ADDRSTRLEN);

std::cout << ip << std::endl;

std::map<int, server::ClientWorker\*> workers;

while(!exit\_flag) {

sockaddr\_in clientAddr{};

socklen\_t sin\_size=sizeof(struct sockaddr\_in);

if(int clientSock=accept(serverSock,(struct sockaddr\*)&clientAddr, &sin\_size)) {

clearDeadWorkers(workers);

inet\_ntop(AF\_INET, &(clientAddr.sin\_addr), ip, INET\_ADDRSTRLEN);

if (std::string(ip) != ("0.0.0.0")) {

std::cout << "Connected to " << ip << std::endl;

auto w = new server::ClientWorker(clientSock, clientAddr);

workers[w->id] = w;

camera.add\_sink(w->id, w->stream());

std::cout << std::to\_string(workers.size()) << " workers used" << std::endl;

}

}

}

std::cout << std::to\_string(workers.size()) << " workers left" << std::endl;

clearAllWorkers(workers);

std::cout << std::to\_string(workers.size()) << " workers left" << std::endl;

close(serverSock);

return 0;

}

Файл **CMakeLists.txt**.

cmake\_minimum\_required(VERSION 3.10)

project(camera\_prog)

set(CMAKE\_CXX\_STANDARD 14)

find\_package (Threads REQUIRED)

find\_package(OpenCV REQUIRED)

include\_directories(${OpenCV\_INCLUDE\_DIRS} "/usr/local/include/opencv4/")

add\_executable(camera\_prog main.cpp base64.cpp utils.cpp)

target\_link\_libraries(${PROJECT\_NAME}

${CMAKE\_THREAD\_LIBS\_INIT}

${OpenCV\_LIBS}

)

БЛОК ГРАФИЧЕСКОГО ИНТЕРФЕЙСА

Файл **video\_cam.py**.

import socket

import base64

import numpy as np

import cv2

import json

import time

import multiprocessing as mp

HOST, PORT = "192.168.136.131", 8008

def receive\_packet(sock):

chunk = ''

data\_list = []

while '\r\n' not in chunk:

chunk = sock.recv(1024).decode()

data\_list.append(chunk)

data = ''.join(data\_list)

pack = json.loads(data)

return pack, len(data)

def run\_loop(cam, queue\_out, stop):

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.connect((HOST, PORT))

s.settimeout(1)

time.sleep(1)

s.send(b'Hello')

packet, \_ = receive\_packet(s)

buf\_decode = base64.b64decode(packet['image'])

timestamp0 = packet['timestamp']

print('Camera started')

while not stop.is\_set():

try:

s.send(b'get')

packet, size\_data = receive\_packet(s)

buf\_decode = base64.b64decode(packet['image'])

timestamp = (packet['timestamp'] - timestamp0) / 1e6

jpg = np.frombuffer(buf\_decode, np.uint8)

frame = cv2.imdecode(jpg, cv2.IMREAD\_UNCHANGED)

if not queue\_out.full():

queue\_out.put(cv2.resize(frame, (1280, 720)))

except Exception:

break

print('Camera stopped')

class VideoCamera:

def \_\_init\_\_(self, cam\_id):

self.cam\_id = cam\_id

self.frame\_buf = mp.Queue(10)

self.stop\_event = mp.Event()

self.cam\_proc = mp.Process(target=run\_loop, args=(self.cam\_id, self.frame\_buf, self.stop\_event))

self.cam\_proc.start()

def get(self):

if self.stop\_event.is\_set():

return None

if not self.frame\_buf.empty():

return self.frame\_buf.get()

else:

return None

def stop(self):

self.stop\_event.set()

self.cam\_proc.terminate()

print('Camera terminated')

Файл **net\_test.py**.

import multiprocessing as mp

ready = mp.Event()

result = mp.Queue()

test\_proc = None

def test\_body(net\_type, model\_path, out, event):

net = net\_type(model\_path)

res = net.verification()

out.put(res)

event.set()

def run\_test(net\_type, model\_path):

global test\_proc

test\_proc = mp.Process(target=test\_body, args=(net\_type, model\_path, result, ready))

test\_proc.start()

def is\_ready():

return ready.is\_set()

def report():

if ready.is\_set():

res = result.get()

test\_proc.join()

return res

else:

return None

Файл **DefectoScopeGUI.py**.

import tkinter as tk

from tkinter import simpledialog as simple\_dlg

from tkinter.dialog import Dialog as tk\_dialog

from PIL import Image as PIL\_Image

from PIL import ImageTk as PIL\_Image\_Tk

import cv2

import numpy as np

import os

import net\_test

from detector import classifier

from detector import detector\_t

import datetime

from detector import analyzer

import video\_cam

import multiprocessing as mp

from database import defects\_list

from database import defects\_base

WINDOW\_SIZE = (1024, 600)

VIDEO\_FRAME\_SIZE = (768, 480) # WVGA

BUTTON\_SIZE = (200, 100)

BACKGROUND\_COLOR = '#8C9787'

MODEL\_DETECTOR\_PATH = 'detector\\models\\rcnn\\300.torch'

MODEL\_CLASSIFIER\_PATH = 'detector\\models\\classification\_model.pth'

defect\_database = defects\_base.DefectsBase()

net\_analyzer = None

camera = None

stop\_event = mp.Event()

capture\_event = mp.Event()

record\_event = mp.Event()

vid\_writer = None

video\_panel\_image = None

analyze\_started = False

bounding\_boxes = []

defect\_types = []

defect\_scores = []

plane\_id = {

'name': "",

'id': '',

'date': '',

'time': '',

'has\_data': False,

}

record\_on = False

def bgr\_to\_tk\_image(image):

b, g, r = cv2.split(image)

img = cv2.merge((r, g, b))

img = PIL\_Image.fromarray(img)

img\_tk = PIL\_Image\_Tk.PhotoImage(image=img)

return img\_tk

def start\_action(root):

global plane\_id

global net\_analyzer

plane\_id['name'] = simple\_dlg.askstring('Input airplane name', 'Name', parent=root)

plane\_id['id'] = simple\_dlg.askstring('Input airplane serial number', 'ID', parent=root)

plane\_id['date'] = datetime.datetime.now().strftime('%d.%m.%Y')

plane\_id['time'] = datetime.datetime.now().strftime('%H:%M:%S')

plane\_dir = os.path.join('data\_analyzed', plane\_id['name'] + '\_' + plane\_id['id'])

if not os.path.exists(plane\_dir):

os.mkdir(plane\_dir)

plane\_dir = os.path.join(plane\_dir, plane\_id['date'] + '\_' + plane\_id['time'].replace(':', '\_'))

os.mkdir(plane\_dir)

plane\_id['results'] = plane\_dir

plane\_id['capture\_counter'] = 0

plane\_id['record\_counter'] = 0

plane\_id['defects'] = defects\_list.AirCraftDefectsList(plane\_id['id'], plane\_id['name'])

print(plane\_id)

if plane\_id['name'] is not None:

capture\_button['state'] = 'normal'

record\_button['state'] = 'normal'

new\_button['state'] = 'disabled'

stop\_button['state'] = 'normal'

test\_button['state'] = 'disabled'

net\_analyzer = analyzer.DefectAnalyzer(MODEL\_CLASSIFIER\_PATH, MODEL\_DETECTOR\_PATH)

def capture\_action():

global capture\_event

capture\_event.set()

print('Capture')

def record\_action():

global record\_on

global record\_event

global vid\_writer

global plane\_id

if not record\_on:

record\_button['image'] = record\_button\_on\_icon

record\_on = True

fourcc = cv2.VideoWriter\_fourcc('m', 'p', '4', 'v')

vid\_writer = cv2.VideoWriter(os.path.join(plane\_id['results'], str(plane\_id['record\_counter']) + '.mp4'),

fourcc, 30, (1280, 720))

record\_event.set()

else:

record\_button['image'] = record\_button\_off\_icon

record\_on = False

record\_event.clear()

vid\_writer.release()

plane\_id['record\_counter'] += 1

def stop\_action():

global net\_analyzer

global plane\_id

capture\_button['state'] = 'disabled'

if record\_on:

record\_action()

record\_button['state'] = 'disabled'

new\_button['state'] = 'normal'

stop\_button['state'] = 'disabled'

test\_button['state'] = 'normal'

if net\_analyzer is not None:

net\_analyzer.stop()

net\_analyzer = None

if plane\_id['has\_data']:

defect\_database.add(plane\_id['defects'])

defect\_database.report(plane\_id['results'], plane\_id['name'], plane\_id['id'])

plane\_id['has\_data'] = False

print('Stop')

def select\_test\_dialog(root):

d = tk\_dialog(root, {'title': 'Choose Test Type',

'text': '',

'bitmap': '',

'default': 0,

'strings': ('Classifier',

'Detector')})

return ('Classifier', 'Detector')[d.num]

def center\_window(w, size):

s\_w = w.winfo\_screenwidth()

s\_h = w.winfo\_screenheight()

s\_xc = int((s\_w / 2) - (size[0] / 2))

s\_yc = int((s\_h / 2) - (size[1] / 2))

w.geometry('{}x{}+{}+{}'.format(size[0], size[1], s\_xc, s\_yc))

return w

def wait\_window(root):

win = tk.Toplevel(root)

win = center\_window(win, (400, 160))

win.resizable(False, False)

win.configure(bg=BACKGROUND\_COLOR)

win.transient()

win.overrideredirect(True)

win.grab\_set()

lbl = tk.Label(win, text='Wait! Test in progress...', bg=BACKGROUND\_COLOR, font=("Arial", 23))

lbl.place(relx=0.5, rely=0.5, anchor="center")

return win

def report\_window(root, text):

win = tk.Toplevel(root)

win = center\_window(win, (900, 500))

win.resizable(False, False)

win.configure(bg=BACKGROUND\_COLOR)

win.transient()

win.title('Results!')

win.grab\_set()

tk.Label(win, text=text, anchor='w', justify=tk.LEFT, bg=BACKGROUND\_COLOR, font=("Courier", 14)).pack()

btn = tk.Button(win, text='Ok', command=win.destroy)

btn.place(x=400, y=400, width=100, height=70, anchor='center')

return win

def wait\_classifier\_test\_end(root, wait\_win):

if net\_test.is\_ready():

res = net\_test.report()

wait\_win.destroy()

report\_text = '\*' \* 80 + '\n' + \

'Testing results:\n' + \

'{:<50}: {:20d}\n'.format('Total images tested', res['total\_images']) + \

'{:<50}: {:20d}\n'.format('True Positive checks', res['tp\_n']) + \

'{:<50}: {:20d}\n'.format('False Negative checks', res['fn\_n']) + \

'{:<50}: {:20d}\n'.format('False Positive checks', res['fp\_n']) + \

'{:<50}: {:19.3f}%\n'.format('Precision', res['precision']) + \

'{:<50}: {:19.3f}%\n'.format('Recall', res['recall']) + \

'{:<50}: {:19.3f}%\n\n'.format('Probability of wrong classification', res['prob\_error']) + \

'{:<50}: {:18.3f}ms\n'.format('Average timing for classification', res['average\_t']) + \

'{:<50}: {:18.3f}ms\n'.format('Maximum timing for classification', res['max\_t']) + \

'{:<50}: {:18.3f}ms'.format('Standard deviation of timing for classification', res['std\_t'])

report\_window(root, report\_text)

test\_button['state'] = 'normal'

else:

root.after(100, wait\_classifier\_test\_end, root, wait\_win)

def wait\_detector\_test\_end(root, wait\_win):

if net\_test.is\_ready():

res = net\_test.report()

wait\_win.destroy()

report\_text = '\*' \* 80 + '\n' + \

'Testing results:\n' + \

'{:<50}: {:20d}\n'.format('Total images tested', res['total\_images']) + \

'{:<50}: {:19.2f}%\n\n'.format('Mean Average Precision (mAP50)', res['mAP50']) + \

'{:<50}: {:18.3f}ms\n'.format('Average timing for classification', res['average\_t']) + \

'{:<50}: {:18.3f}ms\n'.format('Maximum timing for classification', res['max\_t']) + \

'{:<50}: {:18.3f}ms'.format('Standard deviation of timing for classification', res['std\_t'])

report\_window(root, report\_text)

test\_button['state'] = 'normal'

else:

root.after(100, wait\_detector\_test\_end, root, wait\_win)

def test\_action(root):

test\_type = select\_test\_dialog(root)

if test\_type is not None:

if test\_type == 'Classifier':

test\_button['state'] = 'disabled'

net\_test.run\_test(classifier.Classifier, MODEL\_CLASSIFIER\_PATH)

test\_wait\_win = wait\_window(root)

root.after(100, wait\_classifier\_test\_end, root, test\_wait\_win)

elif test\_type == 'Detector':

test\_button['state'] = 'disabled'

net\_test.run\_test(detector\_t.Detector, MODEL\_DETECTOR\_PATH)

test\_wait\_win = wait\_window(root)

root.after(100, wait\_detector\_test\_end, root, test\_wait\_win)

def load\_icon(icon\_name):

return tk.PhotoImage(file=os.path.join('img', icon\_name + '.png'))

def create\_left\_menu\_button(icon, index, action):

button = tk.Button(main\_window,

image=icon, border='0',

bg=BACKGROUND\_COLOR,

activebackground=BACKGROUND\_COLOR)

button.place(x=5 + VIDEO\_FRAME\_SIZE[0] + 20, y=15 + BUTTON\_SIZE[1] \* index + 15 \* index,

width=BUTTON\_SIZE[0], height=BUTTON\_SIZE[1])

button.config(command=action)

return button

def on\_closing(root):

stop\_event.set()

if camera is not None:

camera.stop()

stop\_action()

print('Exit')

root.destroy()

def analyze\_loop(root, video\_panel):

global camera

global analyze\_started

global bounding\_boxes

global defect\_types

global defect\_scores

global plane\_id

global vid\_writer

img = camera.get()

if img is not None:

if net\_analyzer is not None:

if not analyze\_started:

analyze\_started = True

net\_analyzer.test\_start(img)

else:

res = net\_analyzer.get()

if res is not None:

bounding\_boxes = res['bboxes']

defect\_scores = res['probs']

defect\_types = res['types']

analyze\_started = False

for def\_type, def\_score, bbox in zip(defect\_types, defect\_scores, bounding\_boxes):

x1, y1, x2, y2 = bbox

cv2.rectangle(img,

(x1, y1),

(x2, y2),

color=(0, 0, 255),

thickness=2)

cv2.putText(img,

'{:.2f}% {:}'.format(def\_score \* 100.0, def\_type),

(x1, y1 - 10),

fontFace=cv2.FONT\_ITALIC,

fontScale=0.8,

thickness=1,

color=(255, 0, 0))

if capture\_event.is\_set():

cv2.imwrite(os.path.join(plane\_id['results'], str(plane\_id['capture\_counter']) + '.jpg'), img)

plane\_id['capture\_counter'] += 1

if len(bounding\_boxes) > 0:

plane\_id['defects'].add(defects\_list.DefectFrame(img, bounding\_boxes, defect\_scores, defect\_types))

plane\_id['has\_data'] = True

capture\_event.clear()

if record\_event.is\_set():

vid\_writer.write(img)

if len(bounding\_boxes):

plane\_id['defects'].add(defects\_list.DefectFrame(img, bounding\_boxes, defect\_scores, defect\_types))

plane\_id['has\_data'] = True

img = cv2.resize(img, VIDEO\_FRAME\_SIZE)

update\_action(video\_panel, img)

if not stop\_event.is\_set():

root.after(ms=10, func=lambda: analyze\_loop(root, video\_panel))

def update\_action(video\_panel, img):

global video\_panel\_image

video\_panel\_image = bgr\_to\_tk\_image(img)

video\_panel.configure(image=video\_panel\_image)

video\_panel.image = video\_panel\_image

if \_\_name\_\_ == '\_\_main\_\_':

main\_window = tk.Tk()

main\_window = center\_window(main\_window, WINDOW\_SIZE)

main\_window.resizable(False, False)

main\_window.configure(bg=BACKGROUND\_COLOR)

main\_window.title("DefectoScope")

# main\_window.wm\_attributes('-fullscreen', 'True')

frame = np.random.randint(0, 255, size=(VIDEO\_FRAME\_SIZE[1], VIDEO\_FRAME\_SIZE[0], 3), dtype=np.uint8)

video\_panel\_image = bgr\_to\_tk\_image(frame)

video\_out = tk.Label(main\_window,

image=video\_panel\_image,

width=VIDEO\_FRAME\_SIZE[0],

height=VIDEO\_FRAME\_SIZE[1],

borderwidth=2,

relief='ridge')

video\_out.place(x=5, y=5)

new\_button\_icon = load\_icon('button\_new')

new\_button = create\_left\_menu\_button(new\_button\_icon, 0, lambda: start\_action(main\_window))

capture\_button\_icon = load\_icon('button\_capture')

capture\_button = create\_left\_menu\_button(capture\_button\_icon, 1, capture\_action)

capture\_button['state'] = 'disable'

record\_button\_off\_icon = load\_icon('button\_record\_off')

record\_button\_on\_icon = load\_icon('button\_record\_on')

record\_button = create\_left\_menu\_button(record\_button\_off\_icon, 2, record\_action)

record\_button['state'] = 'disable'

stop\_button\_icon = load\_icon('button\_stop')

stop\_button = create\_left\_menu\_button(stop\_button\_icon, 3, stop\_action)

stop\_button['state'] = 'disabled'

test\_button\_icon = load\_icon('button\_test')

test\_button = create\_left\_menu\_button(test\_button\_icon, 4, lambda: test\_action(main\_window))

main\_window.protocol("WM\_DELETE\_WINDOW", lambda: on\_closing(main\_window))

camera = video\_cam.VideoCamera(0)

main\_window.after(ms=10, func=lambda: analyze\_loop(main\_window, video\_out))

main\_window.update()

main\_window.mainloop()

Файл **detector/analyzer.py**.

import multiprocessing as mp

def run\_loop(class\_model, detect\_model, queue\_in, queue\_out, stop):

print('analyzer started!')

from detector import classifier

from detector import detector\_t

net\_detector = detector\_t.Detector(detect\_model)

net\_classifier = classifier.Classifier(class\_model)

while not stop.is\_set():

img = queue\_in.get(block=True)

if img is not None:

result = {

'image': None,

'bboxes': [],

'scores': [],

'types': [],

'probs': [],

}

bboxes, scores = net\_detector.test(img)

result['bboxes'] = bboxes

result['scores'] = scores

for score, bbox in zip(scores, bboxes):

x1, y1, x2, y2 = bbox

img\_small = img[y1:y2, x1:x2]

defect\_type, prob = net\_classifier.test(img\_small)

result['types'].append(defect\_type)

result['probs'].append(prob \* score)

if not queue\_out.full():

queue\_out.put(result)

print('Detector loop stopped!')

class DefectAnalyzer:

def \_\_init\_\_(self, classifier\_model\_path, detector\_model\_path):

self.classifier\_model\_path = classifier\_model\_path

self.detector\_model\_path = detector\_model\_path

self.image\_queue = mp.Queue(10)

self.stop\_event = mp.Event()

self.result\_queue = mp.Queue(10)

self.detector\_proc = mp.Process(target=run\_loop,

args=(

self.classifier\_model\_path,

self.detector\_model\_path,

self.image\_queue,

self.result\_queue,

self.stop\_event)

)

self.detector\_proc.start()

def test\_start(self, image):

if not self.image\_queue.full():

self.image\_queue.put(image)

def get(self):

if not self.result\_queue.empty():

return self.result\_queue.get()

else:

return None

def stop(self):

self.stop\_event.set()

self.image\_queue.put(None)

self.detector\_proc.terminate()

Файл **detector/classifier.py**.

import torch

import cv2

import torchvision.transforms as transforms

import detector.classifier\_net.utils as utils

import time

import os

import torch.nn.functional as nn\_functional

import numpy as np

IMAGE\_SIZE = (64, 64)

class Classifier:

def \_\_init\_\_(self, path\_to\_saved\_model):

self.target\_device = ('cuda' if torch.cuda.is\_available() else 'cpu')

self.model = utils.load\_model(self.target\_device, path\_to\_saved\_model)

self.labels = [

'Class\_1',

'Class\_2',

'Class\_3',

'Class\_4',

'Class\_5',

]

# define preprocess transforms

self.transform = transforms.Compose([

transforms.ToPILImage(),

transforms.Resize(64),

transforms.ToTensor(),

transforms.Normalize(

mean=[0.5, 0.5, 0.5],

std=[0.5, 0.5, 0.5]

)

])

def test(self, image\_to\_test):

image = cv2.resize(image\_to\_test, IMAGE\_SIZE, cv2.INTER\_LINEAR)

image\_bw = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

image\_bw = self.transform(image\_bw)

# add batch dimension

image\_bw = torch.unsqueeze(image\_bw, 0)

with torch.no\_grad():

outputs = self.model(image\_bw.to(self.target\_device))

probs = nn\_functional.softmax(outputs, dim=-1)

predicted\_class\_ = self.labels[torch.argmax(probs)]

probability\_ = float(torch.max(probs).detach().cpu().numpy())

return predicted\_class\_, probability\_

def verification(self):

test\_images\_paths = [

os.path.abspath(os.path.join(os.path.curdir, 'dataset/classification/train')),

os.path.abspath(os.path.join(os.path.curdir, 'dataset/classification/validation'))

]

images\_paths = []

for sub\_folder\_path in test\_images\_paths:

for test\_images\_path in os.listdir(sub\_folder\_path):

images\_folder = os.path.join(sub\_folder\_path, test\_images\_path)

for f in os.listdir(images\_folder):

if os.path.isfile(os.path.join(images\_folder, f)) and '.jpg' in f.lower():

images\_paths.append(os.path.join(images\_folder, f))

infer\_times = []

infer\_valid = []

infer\_non\_valid = []

# pre-heat classifier

image = cv2.imread(images\_paths[0])

\_, \_ = self.test(image)

for im\_path in images\_paths:

image = cv2.imread(im\_path)

# get the ground truth class

gt\_class = im\_path.split('\\')[-2]

t0 = time.time()

prediction, probability = self.test(image)

t1 = time.time()

infer\_times.append((t1 - t0) \* 1000.0)

infer\_valid.append(1 if gt\_class == prediction else 0)

infer\_non\_valid.append(1 if probability < 0.5 else 0)

n\_total = len(infer\_valid)

n\_true\_positives = sum(infer\_valid)

n\_false\_negatives = sum(infer\_non\_valid)

n\_false\_positives = n\_total - sum(infer\_valid)

accuracy = n\_true\_positives / n\_total \* 100.0

precision = n\_true\_positives / (n\_true\_positives + n\_false\_positives) \* 100.0

recall = n\_true\_positives / (n\_true\_positives + n\_false\_negatives) \* 100.0

probability\_error = (n\_false\_positives + n\_false\_negatives) / n\_total \* 100.0

average\_timing = sum(infer\_times) / n\_total

max\_timing = max(infer\_times)

std\_timing = np.std(np.array(infer\_times))

verification\_result = {

'total\_images': n\_total,

'tp\_n': n\_true\_positives,

'fn\_n': n\_false\_negatives,

'fp\_n': n\_false\_positives,

'accuracy': accuracy,

'precision': precision,

'recall': recall,

'prob\_error': probability\_error,

'average\_t': average\_timing,

'max\_t': max\_timing,

'std\_t': std\_timing,

}

return verification\_result

if \_\_name\_\_ == '\_\_main\_\_':

net = Classifier('models\\classification\_model.pth')

res = net.verification()

print('\*' \* 80)

print('Testing results:')

print('Total images tested:', res['total\_images'])

print('True Positive checks: {}'.format(res['tp\_n']))

print('False Negative checks: {}'.format(res['fn\_n']))

print('False Positive checks: {}'.format(res['fp\_n']))

print('Precision: {:.3f}%'.format(res['precision']))

print('Recall: {:.3f}%'.format(res['recall']))

print('Probability of wrong classification: {:.3f}%'.format(res['prob\_error']))

print()

print('Average timing for classification: {:.3f}ms'.format(res['average\_t']))

print('Maximum timing for classification: {:.3f}ms'.format(res['max\_t']))

print('Standard deviation of timing for classification: {:.3f}ms'.format(res['std\_t']))

Файл **detector/detector\_t.py**.

import numpy as np

from torchvision.models.detection.faster\_rcnn import FastRCNNPredictor

from torchvision.models.detection.mask\_rcnn import MaskRCNNPredictor

import cv2

import torchvision.models.segmentation

import torch

from detector.detector\_net.dataset import get\_transform

from PIL import Image

import detector.detector\_net.dataset as labelme\_dataset

from torchmetrics.detection.mean\_ap import MeanAveragePrecision

import time

IMAGE\_SIZE = (600, 600)

class Detector:

def \_\_init\_\_(self, model\_path):

self.device = torch.device('cuda') if torch.cuda.is\_available() else torch.device('cpu')

self.model = torchvision.models.detection.maskrcnn\_resnet50\_fpn(pretrained=True)

in\_features = self.model.roi\_heads.box\_predictor.cls\_score.in\_features

self.model.roi\_heads.box\_predictor = FastRCNNPredictor(in\_features, 2)

in\_features\_mask = self.model.roi\_heads.mask\_predictor.conv5\_mask.in\_channels

hidden\_layer = 256

self.model.roi\_heads.mask\_predictor = MaskRCNNPredictor(in\_features\_mask,

hidden\_layer,

2)

self.model.load\_state\_dict(torch.load(model\_path))

self.model.to(self.device)

self.model.eval()

self.transforms = get\_transform(False)

def test(self, image):

old\_height, old\_width, \_ = image.shape

image = cv2.resize(image, IMAGE\_SIZE, cv2.INTER\_LINEAR)

img = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

im\_pil = Image.fromarray(img)

if self.transforms is not None:

image\_infer, \_ = self.transforms(im\_pil, {})

bboxes, scores = self.infer(image\_infer)

k\_width = old\_width / IMAGE\_SIZE[0]

k\_height = old\_height / IMAGE\_SIZE[1]

new\_bboxes = []

for bbox in bboxes:

x1\_, y1\_, x2\_, y2\_ = bbox

new\_bboxes.append(

(

int(x1\_ \* k\_width),

int(y1\_ \* k\_height),

int(x2\_ \* k\_width),

int(y2\_ \* k\_height)

)

)

return new\_bboxes, scores

def infer(self, image):

images = [image.to(self.device)]

with torch.no\_grad():

pred = self.model(images)

bboxes = []

scores = []

for i in range(len(pred[0]['labels'])):

score = pred[0]['scores'][i].detach().cpu().numpy()

if score > 0.4:

scores.append(float(score))

bbox = pred[0]['boxes'][i].detach().cpu().numpy()

b = list(map(int, bbox.tolist()))

bboxes.append(b)

return bboxes, scores

def verification(self):

dataset = labelme\_dataset.LabelMeDataset('dataset\\detection\_verification', get\_transform(False))

infer\_times = []

image, target = dataset.\_\_getitem\_\_(0)

\_, \_ = self.infer(image)

metric = MeanAveragePrecision()

for i in range(dataset.\_\_len\_\_()):

image, target = dataset.\_\_getitem\_\_(i)

t0 = time.time()

boxes, scores = self.infer(image)

t1 = time.time()

infer\_times.append((t1 - t0) \* 1000.0)

target\_metric = [

dict(

boxes=target['boxes'],

labels=target['labels']

)

]

pred\_metrics = [

dict(

boxes=torch.tensor(boxes),

scores=torch.tensor(scores),

labels=torch.tensor([1] \* len(scores))

)

]

metric.update(pred\_metrics, target\_metric)

res\_m = metric.compute()

mAP = float(res\_m['map\_50']) \* 100.0

verification\_result = {

'total\_images': len(infer\_times),

'mAP50': mAP,

'average\_t': sum(infer\_times) / len(infer\_times),

'max\_t': max(infer\_times),

'std\_t': np.array(infer\_times).std(),

}

return verification\_result

if \_\_name\_\_ == '\_\_main\_\_':

det = Detector('models\\rcnn\\5.torch')

img\_test = cv2.imread('test\_rcnn.jpg')

bb, sc = det.test(img\_test)

for coord, score in zip(bb, sc):

x1, y1, x2, y2 = coord

cv2.rectangle(img\_test,

(x1, y1),

(x2, y2),

color=(0, 0, 255),

thickness=2)

cv2.putText(img\_test,

'{:.2f}% {:}'.format(score \* 100.0, score),

(x1, y1 - 10),

fontFace=cv2.FONT\_ITALIC,

fontScale=0.4,

thickness=1,

color=(255, 0, 0))

cv2.imwrite('test\_rcnn\_detected.jpg', img\_test)

res = det.verification()

print(res)

Файл **detector/detector\_net/dataset.py**.

import json

import base64

import pathlib

import cv2

import numpy

import torch

import torch.utils.data as torch\_data

import detector.detector\_net.transforms as T

from PIL import Image

# reworked from https://pytorch.org/tutorials/intermediate/torchvision\_tutorial.html

def \_create\_masks(shapes, image\_width: int, image\_height: int):

for shape in shapes:

mask = numpy.zeros((image\_height, image\_width), dtype=numpy.uint8)

points = numpy.array(shape['points']).reshape((-1, 1, 2))

points = numpy.round(points).astype(numpy.int32)

cv2.fillPoly(mask, [points], (1, ))

mask = mask.astype(numpy.uint8)

yield mask

def \_create\_bboxs(shapes):

for shape in shapes:

points = numpy.array(shape['points'])

xmin, ymin = numpy.min(points, axis=0)

xmax, ymax = numpy.max(points, axis=0)

yield [xmin, ymin, xmax, ymax]

class LabelMeDataset(torch\_data.Dataset):

def \_\_init\_\_(self, directory, transforms):

self.directory = pathlib.Path(directory)

assert self.directory.exists()

assert self.directory.is\_dir()

self.labelme\_paths = []

for labelme\_path in self.directory.rglob('\*.json'):

with open(labelme\_path, 'r') as labelme\_file:

labelme\_json = json.load(labelme\_file)

required\_keys = ['version', 'flags', 'shapes', 'imagePath', 'imageData', 'imageHeight', 'imageWidth']

assert all(key in labelme\_json for key in required\_keys), (required\_keys, labelme\_json.keys())

self.labelme\_paths += [labelme\_path]

self.transforms = transforms

def \_\_len\_\_(self):

return len(self.labelme\_paths)

def \_\_getitem\_\_(self, idx: int):

labelme\_path = self.labelme\_paths[idx]

image\_path = str(labelme\_path).replace('json', 'jpg')

with open(labelme\_path, 'r') as labelme\_file:

labelme\_json = json.load(labelme\_file)

image\_width = labelme\_json['imageWidth']

image\_height = labelme\_json['imageHeight']

image = Image.open(image\_path).convert('RGB')

labelme\_shapes = [i for i in labelme\_json['shapes'] if len(i['points']) > 2]

assert all(i['shape\_type'] == 'polygon' for i in labelme\_shapes)

masks = list(\_create\_masks(labelme\_shapes, image\_width, image\_height))

bboxes = list(\_create\_bboxs(labelme\_shapes))

target = {}

target['masks'] = torch.as\_tensor(numpy.stack(masks), dtype=torch.uint8)

target['labels'] = torch.ones((len(labelme\_shapes),), dtype=torch.int64)

target['iscrowd'] = torch.zeros\_like(target['labels'], dtype=torch.int64)

target['image\_id'] = torch.tensor([idx], dtype=torch.int64)

bboxes = torch.as\_tensor(bboxes, dtype=torch.float32)

target['area'] = (bboxes[:, 3] - bboxes[:, 1]) \* (bboxes[:, 2] - bboxes[:, 0])

target['boxes'] = bboxes

if self.transforms is not None:

image, target = self.transforms(image, target)

return image, target

def get\_transform(train):

transforms = []

transforms.append(T.PILToTensor())

transforms.append(T.ConvertImageDtype(torch.float))

if train:

transforms.append(T.RandomHorizontalFlip(0.5))

return T.Compose(transforms)

if \_\_name\_\_ == '\_\_main\_\_':

dat = LabelMeDataset('..\\..\\dataset\\detection\_train', get\_transform(True))

print(dat.\_\_len\_\_())

img, tar = dat.\_\_getitem\_\_(1)

print(tar)

print(img.shape)

print(tar['masks'].sum())

Файл **detector/classifier\_net/CNNModel.py**.

import torch.nn as nn

import torch.nn.functional as nn\_functional

class CNNModel(nn.Module):

def \_\_init\_\_(self, num\_classes=5):

super(CNNModel, self).\_\_init\_\_()

# convolution layer #1

# input size [batch size x 3 x 64 x 64]

self.convolution\_1 = nn.Conv2d(in\_channels=3,

out\_channels=18,

kernel\_size=5)

# convolution layer #2

# input size [batch size x 18 x 60 x 60]

self.convolution\_2 = nn.Conv2d(in\_channels=18,

out\_channels=16,

kernel\_size=5)

# linear transformation layer #1

# input size [batch size x 16 x 13 x 13]

self.linear\_transform\_1 = nn.Linear(16 \* 13 \* 13, 120)

# linear transformation layer #2

# input size [batch size x 120]

self.linear\_transform\_2 = nn.Linear(120, 84)

# linear transformation layer #3

# input size [batch size x 84]

self.linear\_transform\_3 = nn.Linear(84, num\_classes)

def forward(self, x):

# convolution layer #1

x = self.convolution\_1(x)

# 2D max pooling over input signals

x = nn\_functional.max\_pool2d(x, kernel\_size=2)

# rectified linear unit function over input signals

x = nn\_functional.relu(x)

# convolution layer #2

x = self.convolution\_2(x)

# 2D max pooling over input signals

x = nn\_functional.max\_pool2d(x, kernel\_size=2)

# rectified linear unit function over input signals

x = nn\_functional.relu(x)

# reshape data

x = x.view(x.shape[0], -1)

# linear transformation layer #1

x = self.linear\_transform\_1(x)

# rectified linear unit function over input signals

x = nn\_functional.relu(x)

# linear transformation layer #2

x = self.linear\_transform\_2(x)

# rectified linear unit function over input signals

x = nn\_functional.relu(x)

# x = nn\_functional.softmax(x, dim=1)

# linear transformation layer #3

out = self.linear\_transform\_3(x)

# out - array of probabilities of "x" is in target classes

return out

Файл **detector/classifier\_net/config.py**.

BATCH\_SIZE = 4

IMAGE\_SIZE = (64, 64)

LEARNING\_RATE = 1e-3

TRAIN\_EPOCHS = 70

Файл **detector/classifier\_net/dataset\_utils.py**.

import torchvision.transforms as transforms

import torchvision.datasets as datasets

from torch.utils.data import DataLoader

from detector.classifier\_net.config import BATCH\_SIZE, IMAGE\_SIZE

def get\_transforms():

return transforms.Compose([

transforms.Resize(IMAGE\_SIZE),

transforms.ToTensor(),

transforms.Normalize(

mean=[0.5, 0.5, 0.5],

std=[0.5, 0.5, 0.5]

)

])

def get\_transforms\_for\_train():

return transforms.Compose([

transforms.Resize(IMAGE\_SIZE),

transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),

transforms.RandomAffine(degrees=15, scale=(0.8, 1.3)),

transforms.RandomHorizontalFlip(p=0.5),

transforms.RandomVerticalFlip(p=0.5),

transforms.ToTensor(),

transforms.Normalize(

mean=[0.5, 0.5, 0.5],

std=[0.5, 0.5, 0.5]

)

])

def folder\_to\_dataset(path\_to\_folder\_with\_images):

return datasets.ImageFolder(

root=path\_to\_folder\_with\_images,

transform=get\_transforms\_for\_train()

)

def dataset\_to\_loader(dataset):

return DataLoader(

dataset, batch\_size=BATCH\_SIZE, shuffle=True,

num\_workers=1, pin\_memory=True

)

Файл **detector/classifier\_net/utils.py**.

import torch

from detector.classifier\_net.CNNModel import CNNModel

import matplotlib

#matplotlib.use('Agg')

import matplotlib.pyplot as plt

def save\_model(path\_to\_model, epochs, model, optimizer, criterion):

torch.save(

{

'epoch': epochs,

'model\_state\_dict': model.state\_dict(),

'optimizer\_state\_dict': optimizer.state\_dict(),

'loss': criterion,

},

path\_to\_model

)

def load\_model(target\_device, path\_to\_model):

model = CNNModel().to(target\_device)

checkpoint = torch.load(path\_to\_model, map\_location=target\_device)

model.load\_state\_dict(checkpoint['model\_state\_dict'])

model.eval()

return model

def save\_plot(title, train\_metric, validation\_metric):

plt.figure(figsize=(10, 7))

plt.plot(

train\_metric, color='green', linestyle='-',

label='train'

)

plt.plot(

validation\_metric, color='blue', linestyle='-',

label='validation'

)

plt.xlabel('Epoch count')

plt.ylabel(title)

plt.legend()

plt.savefig(title + '.png')

БЛОК ОБУЧЕНИЯ ДЕТЕКТОРА

Файл **train\_detector.py**.

import detector.detector\_net.dataset as labelme\_dataset

import torch

import torchvision

from torchvision.models.detection.faster\_rcnn import FastRCNNPredictor

from torchvision.models.detection.mask\_rcnn import MaskRCNNPredictor

import torchvision.models.segmentation

from detector.detector\_net.engine import train\_one\_epoch, evaluate

import detector.detector\_net.utils as utils

from detector.detector\_net.dataset import get\_transform

IMAGE\_SIZE = (600, 600)

TORCH\_DEVICE = torch.device('cuda') if torch.cuda.is\_available() else torch.device('cpu')

print('Train using {}'.format(TORCH\_DEVICE))

def get\_model\_instance\_segmentation(num\_classes):

# load an instance segmentation model pre-trained on COCO

model = torchvision.models.detection.maskrcnn\_resnet50\_fpn(pretrained=True)

# get number of input features for the classifier

in\_features = model.roi\_heads.box\_predictor.cls\_score.in\_features

# replace the pre-trained head with a new one

model.roi\_heads.box\_predictor = FastRCNNPredictor(in\_features, num\_classes)

# now get the number of input features for the mask classifier

in\_features\_mask = model.roi\_heads.mask\_predictor.conv5\_mask.in\_channels

hidden\_layer = 256

# and replace the mask predictor with a new one

model.roi\_heads.mask\_predictor = MaskRCNNPredictor(in\_features\_mask,

hidden\_layer,

num\_classes)

return model

def train\_detector():

dataset = labelme\_dataset.LabelMeDataset('dataset\\detection\_train', get\_transform(True))

dataset\_test = labelme\_dataset.LabelMeDataset('dataset\\detection\_train', get\_transform(False))

indices = torch.randperm(len(dataset)).tolist()

dataset = torch.utils.data.Subset(dataset, indices[:-50])

dataset\_test = torch.utils.data.Subset(dataset\_test, indices[-50:])

data\_loader = torch.utils.data.DataLoader(

dataset, batch\_size=2, shuffle=True, num\_workers=1,

collate\_fn=utils.collate\_fn)

data\_loader\_test = torch.utils.data.DataLoader(

dataset\_test, batch\_size=1, shuffle=False, num\_workers=1,

collate\_fn=utils.collate\_fn)

model = get\_model\_instance\_segmentation(2)

model.to(TORCH\_DEVICE)

params = [p for p in model.parameters() if p.requires\_grad]

optimizer = torch.optim.SGD(params, lr=0.005,

momentum=0.9, weight\_decay=0.0005)

lr\_scheduler = torch.optim.lr\_scheduler.StepLR(optimizer,

step\_size=3,

gamma=0.1)

num\_epochs = 301

for i in range(num\_epochs):

train\_one\_epoch(model, optimizer, data\_loader, TORCH\_DEVICE, i, print\_freq=10)

lr\_scheduler.step()

evaluate(model, data\_loader\_test, device=TORCH\_DEVICE)

if i % 50 == 0:

torch.save(model.state\_dict(), 'detector\\models\\rcnn\\' + str(i) + ".torch")

print("Saved model to:", str(i) + ".torch")

if \_\_name\_\_ == '\_\_main\_\_':

train\_detector()

БЛОК ОБУЧЕНИЯ КЛАССИФИКАТОРА

Файл **train\_classifier.py**.

from detector.classifier\_net.config import \*

import detector.classifier\_net.dataset\_utils as datasets

import contextlib

import time

import torch

import torch.nn as nn

import torch.optim as optim

from detector.classifier\_net.CNNModel import CNNModel

from detector.classifier\_net import utils

TARGET\_DEVICE = ('cuda' if torch.cuda.is\_available() else 'cpu')

def train(cnn\_model, data\_loader, train\_criterion, train\_optimizer=None, is\_validation=False):

if is\_validation:

print('Start validation process...')

cnn\_model.eval()

else:

print('Start training process...')

cnn\_model.train()

loss = 0.0

correct = 0

counter = 0

performance\_time = time.time()

with torch.no\_grad() if is\_validation else contextlib.suppress():

# doesn't need to calc gradients when validation is in progress

for data in data\_loader:

counter += 1

image, labels = data

image = image.to(TARGET\_DEVICE)

labels = labels.to(TARGET\_DEVICE)

if not is\_validation:

# clear gradients on optimizer

train\_optimizer.zero\_grad()

# do forward pass

outputs = cnn\_model(image)

# calculate the loss between estimation and target

loss\_ = train\_criterion(outputs, labels)

# calculate the accuracy

\_, predictions = torch.max(outputs.torch\_data, 1)

correct += sum([1 if prediction == label else 0 for prediction, label in zip(predictions, labels)])

loss += sum([0 if prediction == label else 1 for prediction, label in zip(predictions, labels)])

if not is\_validation:

# do loss backpropagation

loss\_.backward()

# update the optimizer

train\_optimizer.step()

performance\_time = time.time() - performance\_time

# loss and accuracy for the epoch

epoch\_loss = 100.0 \* float(loss) / len(data\_loader.dataset)

epoch\_acc = 100.0 \* float(correct) / len(data\_loader.dataset)

return epoch\_loss, epoch\_acc, performance\_time

if \_\_name\_\_ == '\_\_main\_\_':

print('Model will be trained using: {}'.format(TARGET\_DEVICE.capitalize()))

model = CNNModel().to(TARGET\_DEVICE)

TOTAL\_MODEL\_PARAMETERS = sum(p.numel() for p in model.parameters())

print('Total model parameters: {}'.format(TOTAL\_MODEL\_PARAMETERS))

TOTAL\_TRAINABLE\_PARAMETERS = sum(

p.numel() for p in model.parameters() if p.requires\_grad

)

print('Number of parameters to train:{}'.format(TOTAL\_TRAINABLE\_PARAMETERS))

# Adam Method for Stochastic Optimization.

optimizer = optim.Adam(model.parameters(), lr=LEARNING\_RATE)

# loss function for training - cross entropy loss between estimation and target

criterion = nn.CrossEntropyLoss()

train\_loss\_history = []

validation\_loss\_history = []

train\_accuracy\_history = []

validation\_accuracy\_history = []

# prepare datasets

train\_dataset = datasets.folder\_to\_dataset('dataset/classification/train')

validation\_dataset = datasets.folder\_to\_dataset('dataset/classification/validation')

train\_loader = datasets.dataset\_to\_loader(train\_dataset)

validation\_loader = datasets.dataset\_to\_loader(validation\_dataset)

performance = 0.0

# train model

for epoch in range(TRAIN\_EPOCHS):

print(f"[INFO]: Epoch {epoch + 1} of {TRAIN\_EPOCHS}")

train\_epoch\_loss, train\_epoch\_acc, train\_timer = train(model, train\_loader,

criterion, optimizer, is\_validation=False)

validation\_epoch\_loss, validation\_epoch\_acc, validation\_timer = train(model, validation\_loader,

criterion, is\_validation=True)

performance += train\_timer + validation\_timer

print('[INFO]: Epoch {} of {} done: train in {:.1f}ms, validation in {:.1f}ms'.format(epoch + 1,

TRAIN\_EPOCHS,

train\_timer \* 1000.0,

validation\_timer \* 1000.0)

)

print('Training loss: {:.3f}, training acc: {:.3f}'.format(train\_epoch\_loss, train\_epoch\_acc))

print('Validation loss: {:.3f}, validation acc: {:.3f}'.format(validation\_epoch\_loss, validation\_epoch\_acc))

print('-' \* 60)

train\_loss\_history.append(train\_epoch\_loss)

validation\_loss\_history.append(validation\_epoch\_loss)

train\_accuracy\_history.append(train\_epoch\_acc)

validation\_accuracy\_history.append(validation\_epoch\_acc)

# save trained model

utils.save\_model('detector\\models\\classification\_model.pth', TRAIN\_EPOCHS, model, optimizer, criterion)

# save loss and accuracy plots

utils.save\_plot('Accuracy', train\_accuracy\_history, validation\_accuracy\_history)

utils.save\_plot('Loss', train\_loss\_history, validation\_loss\_history)

print('Training complete!')

print('Train performance: {:.2f}fps'.format(

(len(train\_loader.dataset) + len(validation\_loader.dataset)) \* TRAIN\_EPOCHS / performance)

)

print('Model saved to classification\_model.pth')