Приложение

```
#ifndef ACTFUN_H
#define ACTFUN_H
#include <math.h>
class ActFun
{
public:
    //активационная функция - гиперболический тангенс
    static float activation(float value);
};
#endif // ACTFUN_H
#include "actfun.h"
float ActFun::activation(float value) {
    return 1.7159 * tanh(0.6666 * value);
}
#ifndef FCFILE_H
#define FCFILE_H
#include <QString>
#include <QFile>
#include <QTextStream>
#include <QStringList>
#include <QDir>
#include <QFileInfoList>
#include <QImage>
#include <QDebug>
#include <fcvector.h>
```

```
#include <imaging.h>
class FCFile
{
public:
    static list_vfloat2d openFaces(QString open_directory);
    static void save_net(vfloat data, QString path_save);
    static vfloat load_net(QString path_load);
    static QStringList read_file(QString path);
};
#endif // FCFILE_H
#include "fcfile.h"
void FCFile::save_net(vfloat data, QString path_save) {
    QFile file(path_save);
    file.open(QIODevice::WriteOnly | QIODevice::Text);
    QTextStream out(&file);
    for(int i=0; i<data.size(); i++) {
        out<<QString::number(data[i])<<"|";
    }
    file.close();
}
vfloat FCFile::load_net(QString path_load) {
    QStringList file = read_file(path_load);
    QString line = file.at(0);
    QStringList line_splitted = line.split("|");
    vfloat data;
    for(int i=0; i<line_splitted.size(); i++) {
        QString val = line_splitted.at(i);
        data.append(val.toFloat());
    }
    return data;
```

```
}
QStringList FCFile::read_file(QString path) {
    QFile file(path);
    file.open(QIODevice::ReadOnly);
    QStringList StrList;
    while(!file.atEnd()) {
        StrList<<file.readLine();
    }
    file.close();
    return StrList;
}
list_vfloat2d FCFile::openFaces(QString open_directory) {
   QDir dir(open_directory);
    QFileInfoList dirContent = dir.entryInfoList(QStringList() << "*.jpg", QDir::Dirs |
QDir::Files | QDir::NoDotAndDotDot);
    list_vfloat2d all_faces;
    for(int i=0; i<dirContent.length(); i++) {
        QImage face;
        face.load(dirContent.at(i).absoluteFilePath());
        face = Imaging::toNeuro(face,32,36);
        vfloat2d one_face = Imaging::open_image(face);
        all_faces.append(one_face);
    }
    qDebug()<<all_faces.size();
    return all_faces;
}
#ifndef FCVECTOR_H
#define FCVECTOR_H
#include <math.h>
#include <QVector>
```

```
typedef QVector<float> vfloat;
typedef QVector<QVector<float> > vfloat2d;
typedef QList<vfloat> list_vfloat;
typedef QList<vfloat2d> list_vfloat2d;
typedef QList<list_vfloat2d> list2d_vfloat2d;
typedef QVector<int> vint;
typedef QVector<vint> vint2d;
typedef QList<vint> list_vint;
class FCVector
{
public:
    static vfloat2d vfloat2d_create(const int M, const int N);
    static vfloat vfloat_sum(vfloat &vector1, vfloat &vector2);
    static float vfloat_sum_elem(vfloat &vector);
    static vfloat vfloat_pow(vfloat &vector, double power);
    static vfloat vfloat_sub(vfloat &vector1, vfloat &vector2);
};
#endif // FCVECTOR_H
#include "fcvector.h"
vfloat FCVector::vfloat_sum(vfloat &vector1, vfloat &vector2) {
    if(vector1.size() != vector2.size()) {
        exit(9);
    }
    int i=0;
    vfloat ret(vector1.size());
    for(i=0; i<vector1.size(); i++) {
        ret[i] = vector1[i] + vector2[i];
    }
    return ret;
}
```

```
float FCVector::vfloat_sum_elem(vfloat &vector) {
    float sum;
    for(int i=0; i<vector.size(); i++) {
        sum = sum + vector[i];
    }
    return sum;
}
vfloat FCVector::vfloat_sub(vfloat &vector1, vfloat &vector2) {
    if(vector1.size() != vector2.size()) {
        exit(9);
    }
    int i=0;
    vfloat ret(vector1.size());
    for(i=0; i<vector1.size(); i++) {
        ret[i] = vector1[i] - vector2[i];
    }
    return ret;
}
vfloat FCVector::vfloat_pow(vfloat &vector, double power) {
    int i=0;
    vfloat ret(vector.size());
    for(i=0; i<vector.size(); i++) {
        ret[i] = pow(vector[i],power);
    }
    return ret;
}
vfloat2d FCVector::vfloat2d_create(const int M, const int N) {
    vfloat2d vector;
    vector.resize(M);
    for(int i=0; i<M; i++) {
```

```
        vector[i].resize(N);
    }
    return vector;
}
#ifndef IMAGING_H
#define IMAGING_H
#include <fcvector.h>
#include <QImage>
#include <QColor>
class Imaging
{
public:
    static QImage toNeuro(QImage img, int height, int width);
    static vfloat2d open_image(QImage image);
    static QImage toGrayscale(QImage img);
    static QImage drawRect(QImage &img, int width, int height, int length);
    static QImage skinFilter(QImage image);
    static bool isSkin(QColor color);
    static int minRgb(int r, int g, int b);
    static int maxRgb(int r, int g, int b);
    static QImage toSobel(QImage img);
};
#endif // IMAGING_H
#include "imaging.h"
vfloat2d Imaging::open_image(QImage image) {
    vfloat2d ret_val;
    ret_val.resize(image.height());
    for(int i=0; i<image.height(); i++) {
        ret_val[i].resize(image.width());
        for(int j=0; j<image.width(); j++) {
```

```
            QColor color = image.pixel(j,i);
            int r;
            color.getRgb(&r,&r,&r);
            float pix = (r/100)  - 1.275;
            ret_val[i][j] = pix;
        }
    }
    return ret_val;
}
QImage Imaging::toGrayscale(QImage img) {
    img = img.convertToFormat(QImage::Format_Indexed8);
    QVector<int> transform_table(img.numColors());
    for(int i=0; i<img.numColors() ;i++) {
        QRgb c1=img.color(i);
        int avg=qGray(c1);
        transform_table[i]=avg;
    }
    img.setNumColors(256);
    for(int i=0; i<256; i++) {
        img.setColor(i,qRgb(i,i,i));
    }
    for(int i=0; i<img.numBytes();i++) {
        img.bits()[i]=transform_table[img.bits()[i]];
    }
    return img;
}
QImage Imaging::toNeuro(QImage img, int height, int width) {
    img = toGrayscale(img);
    img = img.scaled(width,height);
    return img;
```

```
}
QImage Imaging::drawRect(QImage &img, int width, int height, int length) {
    for(int i=0; i<length; i++) {
        img.setPixel(width+i,height,-1000);
        img.setPixel(width+i,height+3,-1000);
        img.setPixel(width+i,height+length,-1000);
        img.setPixel(width+i,height+length+3,-1000);
        img.setPixel(width,height+i,-1000);
        img.setPixel(width+3,height+i,-1000);
        img.setPixel(width+length,height+i,-1000);
        img.setPixel(width+length+3,height+i,-1000);
    }
    return img;
}
QImage Imaging::skinFilter(QImage image) {
    for(int i=0; i<image.width(); i++) {
        for(int j=0; j<image.height(); j++) {
            if(!isSkin(image.pixel(i,j))) {
                image.setPixel(i,j,0);
            }
        }
    }
    return image;
}
bool Imaging::isSkin(QColor color) {
    int r,g,b;
    color.getRgb(&r,&g,&b);
    if((r>95) && (g>40) && (b>20) && ((maxRgb(r,g,b) - minRgb(r,g,b)) > 15) &&
(qAbs(r-g) > 15) && (r>g) && (r>b) && ( ((r*100)/(r+g+b)) < 57 ) && ( ((g*100)/
(r+g+b)) < 35 ) && ( ((b*100)/(r+g+b)) < 35 )) {
```

```
        return true;
    }
    return false;
}
int Imaging::minRgb(int r, int g, int b) {
    int min = r;
    if(g<min) {
        min = g;
    }
    if(b<min) {
        min = b;
    }
    return min;
}
int Imaging::maxRgb(int r, int g, int b) {
    int max = r;
    if(g>max) {
        max = g;
    }
    if(b>max) {
        max = b;
    }
    return max;
}
QImage Imaging::toSobel(QImage img) {
    int mask[3][3];
    mask[0][0]=1;  mask[1][0]=2;  mask[2][0]=1;
    mask[0][1]=0;  mask[1][1]=0;  mask[2][1]=0;
    mask[0][2]=-1; mask[1][2]=-2; mask[2][2]=-1;
    for(int i=1; i<img.width()-2; i++) {
```

```
        for(int j=1; j<img.height()-2; j++) {
            int c1 = 0;
            int c2 = 0;
            for(int k=0; k<3; k++) {
                for(int l=0; l<3; l++) {
                    QColor col = img.pixel(i+k,j+l);
                    int r,g,b;
                    col.getRgb(&r,&g,&b);
                    c1 = c1 + r * mask[k][l];
                    c2 = c2 + r * mask[l][k];
                }
                int c = sqrt(pow(c1,2) + pow(c2,2));
                if(c>255) {
                    c = 255;
                }
                c = round(c);
                img.setPixel(i,j,c);
            }
        }
    }
    return img;
}
#include <opencv/cv.h>
#include <opencv/highgui.h>
#include <stdio.h>
#include <assert.h>
#include <QApplication>
#include <QWidget>
#include <QVBoxLayout>
#include "QOpenCVWidget.h"
```

```
#include "MyCameraWindow.h"
#include <QTextCodec>
int main(int argc, char **argv) {
    QTextCodec *cyrillicCodec = QTextCodec::codecForName("UTF-8");
    QTextCodec::setCodecForTr(cyrillicCodec);
    QTextCodec::setCodecForLocale(cyrillicCodec);
    QTextCodec::setCodecForCStrings(cyrillicCodec);
    CvCapture * camera = cvCreateCameraCapture(0);
    assert(camera);
    IplImage * image=cvQueryFrame(camera);
    assert(image);
    printf("Image depth=%i\n", image->depth);
    printf("Image nChannels=%i\n", image->nChannels);
    QApplication app(argc, argv);
    MyCameraWindow *mainWin = new MyCameraWindow(camera);
    mainWin->setWindowTitle("Face Recognition");
    mainWin->show();
    int retval = app.exec();
    cvReleaseCapture(&camera);
    return retval;
}
#ifndef MYCAMERAWINDOW_H_
#define MYCAMERAWINDOW_H_
#include <QWidget>
#include <QVBoxLayout>
#include <QTableWidget>
#include <QList>
#include <QCheckBox>
#include <QProgressBar>
#include <QComboBox>
```

```
#include <opencv/cv.h>
#include <opencv/highgui.h>
#include "QOpenCVWidget.h"
#include <trainer.h>
class MyCameraWindow : public QWidget
{
   Q_OBJECT
   private:
      QOpenCVWidget *cvwidget;
      CvCapture *camera;
   public:
      MyCameraWindow(CvCapture *cam, QWidget *parent=0);
      QTableWidget *image_table;
      QList<QCheckBox*> list_of_cbox;
      QStringList names;
      QProgressBar *progress;
      QComboBox *learn_type;
   protected:
      void timerEvent(QTimerEvent*);
   private slots:
      void screen_shot();
      void delete_images();
      void start_train();
      void show_settings();
};
#endif /*MYCAMERAWINDOW_H_*/
#include "MyCameraWindow.h"
#include <QPushButton>
#include <QDebug>
#include <QTableWidget>
```

```
#include <QCheckBox>
#include <QDir>
#include <QTime>
#include <QProgressBar>
#include <QComboBox>
#include <CNN/cnn.h>
#include <QtConcurrentRun>
#include <QLineEdit>
MyCameraWindow::MyCameraWindow(CvCapture *cam, QWidget *parent) :
QWidget(parent) {
    camera = cam;
    QVBoxLayout *layout = new QVBoxLayout;
    cvwidget = new QOpenCVWidget(this);
    learn_type = new QComboBox;
    learn_type->addItem("Режим обучения");
    learn_type->addItem("Режим распознавания");
    layout->addWidget(learn_type);
    layout->addWidget(cvwidget);
    QHBoxLayout *main_lay = new QHBoxLayout;
    QHBoxLayout *btn_lay = new QHBoxLayout;
    QPushButton *btn_shot = new QPushButton;
    btn_shot->setText("Снимок");
    QObject::connect(btn_shot, SIGNAL(clicked()), this, SLOT(screen_shot()));
    QPushButton *btn_learn = new QPushButton;
    btn_learn->setText("Старт обучения");
    connect(btn_learn, SIGNAL(clicked()), this, SLOT(start_train()));
    btn_lay->addWidget(btn_learn);
    btn_lay->addWidget(btn_shot);
    layout->addLayout(btn_lay);
    QPushButton *settings = new QPushButton;
```

```
    settings->setText("Настройки программы");
    layout->addWidget(settings);
    connect(settings, SIGNAL(clicked()),this,SLOT(show_settings()));
   QVBoxLayout *image_lay = new QVBoxLayout;
    image_table = new QTableWidget;
    image_table->setColumnCount(2);
    image_table->setColumnWidth(0,60);
    ///открываем изображения из папки
    QDir dir(QDir::currentPath() + "/img_for_train/");
    QFileInfoList dirContent = dir.entryInfoList(QStringList() << "*.jpg", QDir::Dirs |
QDir::Files | QDir::NoDotAndDotDot);
    for(int i=0; i<dirContent.length(); i++) {
        QImage temp_face;
        temp_face.load(dirContent.at(i).absoluteFilePath());
        temp_face = temp_face.scaled(60,60);
        QString name = dirContent.at(i).absoluteFilePath();
        names.append(name);
        image_table->setRowCount(image_table->rowCount()+1);
        QLabel *temp_item = new QLabel;
        temp_item->setPixmap(QPixmap::fromImage(temp_face));
        image_table->setCellWidget(image_table->rowCount()-1,0,temp_item);
        image_table->setRowHeight(image_table->rowCount()-1,60);
        QCheckBox *temp_check = new QCheckBox;
        image_table->setCellWidget(image_table->rowCount()-1,1,temp_check);
        list_of_cbox.append(temp_check);
    }
    QTableWidgetItem *header_item1 = new QTableWidgetItem;
    header_item1->setText("Снимки");
    QTableWidgetItem *header_item2 = new QTableWidgetItem;
    header_item2->setText("*");
```

```
    image_table->setHorizontalHeaderItem(0,header_item1);
    image_table->setHorizontalHeaderItem(1,header_item2);
    image_table->setFixedWidth(200);
    image_lay->addWidget(image_table);
    QPushButton *image_del = new QPushButton;
    image_del->setText("Удалить выделенные");
    image_lay->addWidget(image_del);
    connect(image_del, SIGNAL(clicked()),this,SLOT(delete_images()));
    main_lay->addLayout(layout);
    main_lay->addLayout(image_lay);
    setLayout(main_lay);
    resize(400, 300);
    startTimer(10);  // 0.1-second timer
 }
void MyCameraWindow::timerEvent(QTimerEvent*) {
    IplImage *image=cvQueryFrame(camera);
    if(learn_type->currentIndex() == 0) {
        cvwidget->recognizeImage(image);
    } else {
        cvwidget->putImage(image);
    }
}
void MyCameraWindow::screen_shot() {
    QImage copy;
    copy = cvwidget->image.copy(cvwidget->old_point.x(), cvwidget->old_point.y(),
120, 120);
    QString name = QDir::currentPath() + "/img_for_train/img" +
QTime::currentTime().toString() + ".jpg";
    copy.save(name);
    names.append(name);
```

```
    copy = copy.scaled(60,60);
    image_table->setRowCount(image_table->rowCount()+1);
    copy.scaled(copy.width()/2, copy.height()/2);
    QLabel *temp_item = new QLabel;
    temp_item->setPixmap(QPixmap::fromImage(copy));
    image_table->setCellWidget(image_table->rowCount()-1,0,temp_item);
    image_table->setRowHeight(image_table->rowCount()-1,60);
    QCheckBox *temp_check = new QCheckBox;
    image_table->setCellWidget(image_table->rowCount()-1,1,temp_check);
    list_of_cbox.append(temp_check);
}
void MyCameraWindow::delete_images() {
    QList<int> indexes;
    for(int i=0; i<image_table->rowCount(); i++) {
        if(list_of_cbox.at(i)->isChecked()) {
            delete image_table->cellWidget(i,0);
            delete image_table->cellWidget(i,1);
            indexes.append(i);
        }
    }
    for(int i=0; i<indexes.length(); i++) {
        list_of_cbox.removeAt(indexes.at(i-i));
        image_table->removeRow(indexes.at(i-i));
        QFile::remove(names.at(indexes.at(i-i)));
        names.removeAt(indexes.at(i-i));
    }
}
void MyCameraWindow::start_train() {
    QString faces_dir = QDir::currentPath() + "/img_for_train/";
    QString no_faces_dir = QDir::currentPath() + "/nofaces/";
```

```
    Trainer train;
    qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));
    train.image_rga_train(faces_dir,no_faces_dir);
}
void MyCameraWindow::show_settings() {
    QWidget *settings_widget = new QWidget;
    QVBoxLayout *set_main_lay = new QVBoxLayout;
    QLabel *conf_label = new QLabel;
    conf_label->setText("Параметры обучения");
    conf_label->setAlignment(Qt::AlignCenter);
    set_main_lay->addWidget(conf_label);
    QHBoxLayout *set_lay1 = new QHBoxLayout;
    QLabel *mut_val = new QLabel;
    mut_val->setText("Величина мутации:          ");
    QLineEdit *mut_val_edit = new QLineEdit;
    set_lay1->addWidget(mut_val);
    set_lay1->addWidget(mut_val_edit);
    set_main_lay->addLayout(set_lay1);
    QHBoxLayout *set_lay2 = new QHBoxLayout;
    QLabel *mut_ch = new QLabel;
    mut_ch->setText("Вероятность мутации:     ");
    QLineEdit *mut_ch_edit = new QLineEdit;
    set_lay2->addWidget(mut_ch);
    set_lay2->addWidget(mut_ch_edit);
    set_main_lay->addLayout(set_lay2);
    QHBoxLayout *set_lay3 = new QHBoxLayout;
    QLabel *pop_sz = new QLabel;
    pop_sz->setText("Размер популяции:          ");
    QLineEdit *pop_sz_edit = new QLineEdit;
    set_lay3->addWidget(pop_sz);
```

```
    set_lay3->addWidget(pop_sz_edit);
    set_main_lay->addLayout(set_lay3);
    QHBoxLayout *set_lay4 = new QHBoxLayout;
    QLabel *init_temp = new QLabel;
    init_temp->setText("Начальная температура: ");
    QLineEdit *init_temp_edit = new QLineEdit;
    set_lay4->addWidget(init_temp);
    set_lay4->addWidget(init_temp_edit);
    set_main_lay->addLayout(set_lay4);
    QHBoxLayout *set_lay5 = new QHBoxLayout;
    QLabel *final_temp = new QLabel;
    final_temp->setText("Конечная температура:   ");
    QLineEdit *final_temp_edit = new QLineEdit;
    set_lay5->addWidget(final_temp);
    set_lay5->addWidget(final_temp_edit);
    set_main_lay->addLayout(set_lay5);
    QPushButton *save_btn = new QPushButton;
    save_btn->setText("Сохранить настройки");
    set_main_lay->addWidget(save_btn);
    settings_widget->setWindowTitle("Settings");
    settings_widget->setLayout(set_main_lay);
    settings_widget->show();
}
#ifndef QOPENCVWIDGET_H
#define QOPENCVWIDGET_H
#include <opencv/cv.h>
#include <QPixmap>
#include <QLabel>
#include <QWidget>
#include <QVBoxLayout>
```

```
#include <QImage>
#include <fcfile.h>
#include <imaging.h>
class QOpenCVWidget : public QWidget {
   private:
      QLabel *imagelabel;
      QVBoxLayout *layout;
   public:
      QImage image;
      QPoint old_point;
      QOpenCVWidget(QWidget *parent = 0);
      ~QOpenCVWidget(void);
      void putImage(IplImage *);
      void recognizeImage(IplImage *cvimage);
      QPoint find_face(QImage image, QString path_net);
};
#endif
#include "QOpenCVWidget.h"
#include <QGraphicsEffect>
#include <QDir>
#include <CNN/cnn.h>
#include <QtConcurrentRun>
#include <QPushButton>
#include <QTableWidget>
#include <omp.h>
// Constructor
QOpenCVWidget::QOpenCVWidget(QWidget *parent) : QWidget(parent) {
   layout = new QVBoxLayout;
   imagelabel = new QLabel;
   QImage dummy(100,100,QImage::Format_RGB32);
```

```
    image = dummy;
    layout->addWidget(imagelabel);
    for (int x = 0; x < 100; x ++) {
        for (int y =0; y < 100; y++) {
            image.setPixel(x,y,qRgb(x, y, y));
        }
    }
    imagelabel->setPixmap(QPixmap::fromImage(image));
    old_point.setX(999);
    old_point.setY(999);
    setLayout(layout);
}
QOpenCVWidget::~QOpenCVWidget(void) {}
void QOpenCVWidget::putImage(IplImage *cvimage) {
    int cvIndex, cvLineStart;
    switch (cvimage->depth) {
        case IPL_DEPTH_8U:
            switch (cvimage->nChannels) {
                case 3:
                    if ( (cvimage->width != image.width()) || (cvimage->height !=
image.height()) ) {
                        QImage temp(cvimage->width, cvimage->height,
QImage::Format_RGB32);
                        image = temp;
                    }
                    cvIndex = 0; cvLineStart = 0;
                    for (int y = 0; y < cvimage->height; y++) {
                        unsigned char red,green,blue;
                        cvIndex = cvLineStart;
                        for (int x = 0; x < cvimage->width; x++) {
```

```
                    // DO it
                    red = cvimage->imageData[cvIndex+2];
                    green = cvimage->imageData[cvIndex+1];
                    blue = cvimage->imageData[cvIndex+0];
                    image.setPixel(x,y,qRgb(red, green, blue));
                    cvIndex += 3;
                }
                cvLineStart += cvimage->widthStep;
            }
            break;
        default:
            printf("This number of channels is not supported\n");
            break;
    }
    break;
  default:
    printf("This type of IplImage is not implemented in QOpenCVWidget\n");
    break;
}
image = image.scaled(image.width()/2, image.height()/2);
if(old_point.x() != 999) {
  image = Imaging::drawRect(image,old_point.x(),old_point.y(),120);
}
int secs = QTime::currentTime().second();
if((secs%3) == 0) {
  QString net = QDir::currentPath() + "/NET.txt190";
  QPoint point = find_face(image,net);
  if(point.x() != 0) {
    image = Imaging::drawRect(image,point.x(),point.y(),120);
    old_point.setX(point.x());
```

```
            old_point.setY(point.y());
        }
    }
    imagelabel->setPixmap(QPixmap::fromImage(image));
}
void QOpenCVWidget::recognizeImage(IplImage *cvimage) {
    int cvIndex, cvLineStart;
    switch (cvimage->depth) {
        case IPL_DEPTH_8U:
            switch (cvimage->nChannels) {
                case 3:
                    if ( (cvimage->width != image.width()) || (cvimage->height !=
image.height()) ) {
                        QImage temp(cvimage->width, cvimage->height,
QImage::Format_RGB32);
                        image = temp;
                    }
                    cvIndex = 0; cvLineStart = 0;
                    for (int y = 0; y < cvimage->height; y++) {
                        unsigned char red,green,blue;
                        cvIndex = cvLineStart;
                        for (int x = 0; x < cvimage->width; x++) {
                            // DO it
                            red = cvimage->imageData[cvIndex+2];
                            green = cvimage->imageData[cvIndex+1];
                            blue = cvimage->imageData[cvIndex+0];
                            image.setPixel(x,y,qRgb(red, green, blue));
                            cvIndex += 3;
                        }
                        cvLineStart += cvimage->widthStep;
```

```
                }
                break;
            default:
                printf("This number of channels is not supported\n");
                break;
        }
        break;
    default:
        printf("This type of IplImage is not implemented in QOpenCVWidget\n");
        break;
  }
  image = image.scaled(image.width()/2, image.height()/2);
  old_point.setX(100);
  old_point.setY(70);
  image = Imaging::drawRect(image,old_point.x(),old_point.y(),120);
  imagelabel->setPixmap(QPixmap::fromImage(image));
}
QPoint QOpenCVWidget::find_face(QImage image, QString path_net) {
  QPoint point;
  point.setX(0);
  point.setY(0);
  vfloat vfloat_data = FCFile::load_net(path_net);
  cnn_data data = CNN::vfloat_to_cnn_data(vfloat_data);
  omp_set_dynamic(0);
  omp_set_num_threads(6);
  int mask = 120;
    for(int i=0; i<image.width()-mask; i=i+mask/4) {
      int j=0;
#pragma omp parallel shared(point, image) private(j)
    {
```

```
#pragma omp for
        for(j=0; j<image.height()-mask; j=j+mask/4) {
            QImage on_net = image.copy(i,j,mask,mask);
            on_net = Imaging::toNeuro(on_net,32,36);
            vfloat2d one_face = Imaging::open_image(on_net);
            float output = QtConcurrent::run(CNN::convolution_net,one_face,data);
            if(output>0.7) {
                point.setX(i);
                point.setY(j);
                qDebug()<<output;
            }
        }
#pragma omp barrier
    }
        if(point.x() != 0) {
            return point;
        }
    }
#pragma omp barrier
  return point;
}
#ifndef CNN_H
#define CNN_H
#include "CNN_global.h"
#include <QVector>
#include <QList>
#include <qdebug.h>
#include <math.h>
#include <QTime>
#include <QFile>
```

```
#include <QString>
#include <QStringList>
#include <QImage>
#include <QColor>
#include <QDir>
#include <fcvector.h>
#include <actfun.h>
//общий слой
struct cnn_data {
    list_vfloat2d kernel1;
    vfloat bias1;
    vfloat bias2;
    list_vfloat2d kernel3;
    vfloat bias3;
    vfloat weight4;
    vfloat bias4;
    vfloat bias5;
    list_vfloat2d weight5;
    vfloat weight6;
    float bias6_1;
};
typedef QList<layer*> c_net;
class CNNSHARED_EXPORT CNN {
public:
    //функция свертки
    static vfloat2d convolution(vfloat2d &kernel, float bias, vfloat2d &input);
    //функция субдескретизации
    static vfloat2d subsampling(vfloat2d &input, float bias, float weight);
    //свертка + субдескретизация
    static vfloat2d conv_and_subs(vfloat2d &input, vfloat2d &c_kernel, float c_bias,
```

```
float s_bias, float s_weight);
    static vfloat2d sum_fmaps(vfloat2d &fmap1, vfloat2d &fmap2);
    static float subs_to_neuron(vfloat2d &weight, vfloat2d &fmap, float bias);
    static void append_vfloat2d(vfloat2d &kernel, vfloat &data);
    static vfloat2d append_data_to_vfloat2d(int kern_height, int kern_width, vfloat
&data, int &counter);
    static float convolution_net(vfloat2d &input, cnn_data &data);
    static vfloat cnn_data_to_vfloat(cnn_data &data);
    static cnn_data vfloat_to_cnn_data(vfloat &data);
};
#endif // CNN_H
#include "cnn.h"
#include <QtConcurrentRun>
#include <omp.h>
//функция свертки
vfloat2d CNN::convolution(vfloat2d &kernel, float bias, vfloat2d &input) {
    //проверяем ядро на пустоту
    if(kernel.empty()) {
        exit(1);
    }
    //проверяем размерность ядра
    if(kernel.size() != kernel.at(0).size()) {
        exit(2);
    }
    //проверяем входные значения на пустоту
    if(input.empty()) {
        exit(3);
    }
    //вычисление исходных размерностей
    //размерность ядра
```

```
    int kernel_size = kernel.size();
    //ширина входного слоя
    int input_width = input.at(0).size();
    //высота входного слоя
    int input_height = input.size();
    //вычисляем размер выходной плоскости
    //ширина выходной плоскости
    int out_width = input_width - kernel_size + 1;
    //высота выходной плоскости
    int out_height = input_height - kernel_size + 1;
    //создаем выходную плоскость
    vfloat2d output;
    for(int i=0; i<out_height; i++) {
        vfloat out_line;
        out_line.resize(out_width);
        output.append(out_line);
    }
    //выполняем свертку для каждого нейрона в плоскости
    for(int i=0; i<out_height; i++) {
        for(int j=0; j<out_width; j++) {
            for(int s=0; s<kernel_size; s++) {
                for(int t=0; t<kernel_size; t++) {
                    output[i][j] = output[i][j] + kernel[s][t] * input[i+s][j+t];
                }
            }
            output[i][j] = output[i][j] + bias;
        }
    }
    return output;
}
```

```
//функция субдескритезации
vfloat2d CNN::subsampling(vfloat2d &input, float bias, float weight) {
   //проверяем входную плоскость на пустоту
   if(input.empty()) {
      exit(4);
   }
   //задаем размеры выходной плоскости, в 2 раза меньше входной
   //ширина
   int out_width = input.at(0).size() / 2;
   //высота
   int out_height = input.size() / 2;
   //создаем выходную плоскость заданной размерностью
   vfloat2d output;
   for(int i=0; i<out_height; i++) {
      vfloat out_line;
      out_line.resize(out_width);
      output.append(out_line);
   }
   //вычисляем среднее 4-х входов, умножаем на синаптический коэффициент и
прибавляем нейронное смещение
   for(int i=0; i<out_height; i++) {
      for(int j=0; j<out_width; j++) {
         output[i][j] = ActFun::activation( ((input[2*i][2*j] + input[2*i+1][2*j] +
input[2*i][2*j+1] + input[2*i+1][2*j+1]) / 4) * weight + bias);
      }
   }
   return output;
}
vfloat2d CNN::conv_and_subs(vfloat2d &input, vfloat2d &c_kernel, float c_bias,
float s_bias, float s_weight) {
```

```
    vfloat2d output = convolution(c_kernel,c_bias,input);
    output = subsampling(output,s_bias,s_weight);
    return output;
}
float CNN::convolution_net(vfloat2d &input, cnn_data &data) {
    list_vfloat2d fmap1;
    for(int i=0; i<5; i++) {
        fmap1.append(convolution(data.kernel1[i],data.bias1[i],input));
    }
    list_vfloat2d fmap2;
    for(int i=0; i<5; i++) {
        fmap2.append(subsampling(fmap1[i],data.bias2[i],data.weight2[i]));
    }
    list_vfloat2d fmap3;
    for(int i=0; i<5; i++) {
        fmap3.append(convolution(data.kernel3[i],data.bias3[i],fmap2[i]));
    }
    for(int i=0; i<5; i++) {
        fmap3.append(convolution(data.kernel3[i+5],data.bias3[i+5],fmap2[i]));
    }
    list_vfloat2d temp;
    temp.append(convolution(data.kernel3[10],data.bias3[10],fmap2[0]));
    temp.append(convolution(data.kernel3[11],data.bias3[10],fmap2[1]));
    temp.append(convolution(data.kernel3[12],data.bias3[11],fmap2[0]));
    temp.append(convolution(data.kernel3[13],data.bias3[11],fmap2[2]));
    temp.append(convolution(data.kernel3[14],data.bias3[12],fmap2[0]));
    temp.append(convolution(data.kernel3[15],data.bias3[12],fmap2[3]));
    temp.append(convolution(data.kernel3[16],data.bias3[13],fmap2[0]));
    temp.append(convolution(data.kernel3[17],data.bias3[13],fmap2[4]));
    temp.append(convolution(data.kernel3[18],data.bias3[14],fmap2[1]));
```

```
    temp.append(convolution(data.kernel3[19],data.bias3[14],fmap2[2]));
    temp.append(convolution(data.kernel3[20],data.bias3[15],fmap2[1]));
    temp.append(convolution(data.kernel3[21],data.bias3[15],fmap2[3]));
    temp.append(convolution(data.kernel3[22],data.bias3[16],fmap2[1]));
    temp.append(convolution(data.kernel3[23],data.bias3[16],fmap2[4]));
    temp.append(convolution(data.kernel3[24],data.bias3[17],fmap2[2]));
    temp.append(convolution(data.kernel3[25],data.bias3[17],fmap2[3]));
    temp.append(convolution(data.kernel3[26],data.bias3[18],fmap2[2]));
    temp.append(convolution(data.kernel3[27],data.bias3[18],fmap2[4]));
    temp.append(convolution(data.kernel3[28],data.bias3[19],fmap2[3]));
    temp.append(convolution(data.kernel3[29],data.bias3[19],fmap2[4]));
    for(int i=0; i<20; i=i+2) {
        fmap3.append(sum_fmaps(temp[i],temp[i+1]));
    }
    list_vfloat2d fmap4;
    for(int i=0; i<20; i++) {
        fmap4.append(subsampling(fmap3[i],data.bias4[i],data.weight4[i]));
    }
    vfloat neuron_out5;
    for(int i=0; i<20; i++) {
        neuron_out5.append(subs_to_neuron(data.weight5[i],fmap4[i],data.bias5[i]));
    }
    vfloat mul6;
    for(int i=0; i<20; i++) {
        mul6.append(data.weight6[i] * neuron_out5[i]);
    }
    float sum = vfloat_sum_elem(mul6);
    sum = activation(sum + data.bias6_1);
    return sum;
}
```

```
float CNN::subs_to_neuron(vfloat2d &weight, vfloat2d &fmap, float bias) {
    float sum;
    for(int i=0; i<weight.size(); i++) {
        for(int j=0; j<weight.at(0).size(); j++) {
            sum = sum + (weight[i][j] * fmap[i][j]);
        }
    }
    sum = ActFun::activation(sum+bias);
    return sum;
}
vfloat CNN::cnn_data_to_vfloat(cnn_data &data) {
    vfloat new_data;
    for(int i=0; i<5; i++) {
        append_vfloat2d(data.kernel1[i],new_data);
    }
    for(int i=0; i<5; i++) {
        new_data.append(data.bias1[i]);
    }
    for(int i=0; i<5; i++) {
        new_data.append(data.weight2[i]);
    }
    for(int i=0; i<5; i++) {
        new_data.append(data.bias2[i]);
    }
    for(int i=0; i<10; i++) {
        append_vfloat2d(data.kernel3[i],new_data);
    }
    for(int i=0; i<10; i++) {
        new_data.append(data.bias3[i]);
    }
```

```
    for(int i=10; i<30; i++) {
        append_vfloat2d(data.kernel3[i],new_data);
    }
    for(int i=10; i<20; i++) {
        new_data.append(data.bias3[i]);
    }
    for(int i=0; i<20; i++) {
        new_data.append(data.weight4[i]);
    }
    for(int i=0; i<20; i++) {
        new_data.append(data.bias4[i]);
    }
    for(int i=0; i<20; i++) {
        new_data.append(data.bias5[i]);
    }
    for(int i=0; i<20; i++) {
        append_vfloat2d(data.weight5[i],new_data);
    }
    for(int i=0; i<20; i++) {
        new_data.append(data.weight6[i]);
    }
    new_data.append(data.bias6_1);
    return new_data;
}
 cnn_data CNN::vfloat_to_cnn_data(vfloat &data) {
    int counter = 0;
    cnn_data new_data;
    int kernel_1_size = 5;
    for(int i=0; i<5; i++) {
        new_data.kernel1.append(append_data_to_vfloat2d(kernel_1_size,kernel_1_size
```

```
,data,counter));
    }
    for(int i=0; i<5; i++) {
        new_data.bias1.append(data[counter]);
        counter++;
    }
    for(int i=0; i<5; i++) {
        new_data.weight2.append(data[counter]);
        counter++;
    }
    for(int i=0; i<5; i++) {
        new_data.bias2.append(data[counter]);
        counter++;
    }
    int kernel_3_size = 3;
    for(int i=0; i<10; i++) {
        new_data.kernel3.append(append_data_to_vfloat2d(kernel_3_size,kernel_3_size
,data,counter));
    }
    for(int i=0; i<10; i++) {
        new_data.bias3.append(data[counter]);
        counter++;
    }
    for(int i=10; i<30; i++) {
        new_data.kernel3.append(append_data_to_vfloat2d(kernel_3_size,kernel_3_size
,data,counter));
    }
    for(int i=10; i<20; i++) {
        new_data.bias3.append(data[counter]);
        counter++;
```

```
    }
    for(int i=0; i<20; i++) {
        new_data.weight4.append(data[counter]);
        counter++;
    }
    for(int i=0; i<20; i++) {
        new_data.bias4.append(data[counter]);
        counter++;
    }
    for(int i=0; i<20; i++) {
        new_data.bias5.append(data[counter]);
        counter++;
    }
    int weight_5_height = 6;
    int weight_5_width = 7;
    for(int i=0; i<20; i++) {
        new_data.weight5.append(append_data_to_vfloat2d(weight_5_height,weight_5
_width,data,counter));
    }
    for(int i=0; i<20; i++) {
        new_data.weight6.append(data[counter]);
        counter++;
    }
    new_data.bias6_1 = data[counter];
    counter++;
    return new_data;
}
vfloat2d CNN::sum_fmaps(vfloat2d &fmap1, vfloat2d &fmap2) {
    vfloat2d ret_fmap;
    ret_fmap.resize(fmap1.size());
```

```
    for(int i=0; i<fmap1.size(); i++) {
        ret_fmap[i].resize(fmap1.at(i).size());
        for(int j=0; j<fmap1.at(0).size(); j++) {
            ret_fmap[i][j] = fmap1[i][j] + fmap2[i][j];
        }
    }
    return ret_fmap;
}
vfloat2d CNN::append_data_to_vfloat2d(int kern_height, int kern_width, vfloat
&data, int &counter) {
    vfloat2d ret_val;
    ret_val.resize(kern_height);
    for(int i=0; i<kern_height; i++) {
        ret_val[i].resize(kern_width);
        for(int j=0; j<kern_width; j++) {
            ret_val[i][j] = data[counter];
            counter++;
        }
    }
    return ret_val;
}
void CNN::append_vfloat2d(vfloat2d &kernel, vfloat &data) {
    for(int i=0; i<kernel.size(); i++) {
        for(int j=0; j<kernel.at(i).size(); j++) {
            data.append(kernel[i][j]);
        }
    }
}
#ifndef RGA_H
#define RGA_H
```

```
#include <fcvector.h>
#include <rnd.h>
#include <srv.h>
#include <QDebug>
class RGA
{
public:
    static int best_chromosome(vfloat2d &outputs, vfloat &target);
    static vfloat2d random_blx(vfloat2d &population, double blx_koef);
    static vfloat2d tournament_selection(vfloat2d &population, vfloat &values);
    static vfloat calculate_fitness(vfloat2d &outputs, vfloat &target);
    static vfloat blx_a_crossover(vfloat &chromosome1, vfloat &chromosome2, float
A);
    static vfloat2d population_mutation(vfloat2d &population, int chance);
    static float random_mutation(float value);
    static float fitness_neural(vfloat &output, vfloat &target);
};
#endif // RGA_H
#include "rga.h"
vfloat2d RGA::random_blx(vfloat2d &population, double blx_koef) {
    int population_size = population.size();
    vfloat2d new_population;
    for(int i=0; i<population_size; i++) {
        int rand_ch1 = RND::rand_A_B(0,population_size-1);
        int rand_ch2 = RND::rand_A_B(0,population_size-1);
        new_population.append(blx_a_crossover(population[rand_ch1
],population[rand_ch2],blx_koef));
    }
    return new_population;
}
```

```
int RGA::best_chromosome(vfloat2d &outputs, vfloat &target) {
    vfloat errors = calculate_fitness(outputs,target);
    float min = errors[0];
    int ret_val = 0;
    for(int i=0; i<errors.size(); i++) {
        if(errors[i]<=min) {
            ret_val = i;
        }
    }
    return ret_val;
}
vfloat RGA::calculate_fitness(vfloat2d &outputs, vfloat &target) {
    vfloat ret_val;
    ret_val.resize(outputs.size());
    for(int i=0; i<outputs.size(); i++) {
        ret_val[i] = fitness_neural(outputs[i],target);
    }
    return ret_val;
}
vfloat2d RGA::tournament_selection(vfloat2d &population, vfloat &values) {
    int population_size = population.size();
    vfloat2d new_population;
    for(int i=0; i<population_size; i++) {
        int rand_ch1 = RND::rand_A_B(0,population_size-1);
        int rand_ch2 = RND::rand_A_B(0,population_size-1);
        if(values[rand_ch1] < values[rand_ch2]) {
            new_population.append(population[rand_ch1]);
        } else {
            new_population.append(population[rand_ch2]);
        }
```

```
    }
    return new_population;
}
float RGA::fitness_neural(vfloat &output, vfloat &target) {
    if(output.size() != target.size()) {
        qDebug()<<"Несовпадение размеров выхода сети и требуемых выходов";
        exit(3);
    }
    vfloat sub = FCVector::vfloat_sub(output,target);
    vfloat power = FCVector::vfloat_pow(sub,2);
    float sum = FCVector::vfloat_sum_elem(power);
    float fitness = sum/output.size();
    return fitness;
}
vfloat RGA::blx_a_crossover(vfloat &chromosome1, vfloat &chromosome2, float A)
{
    if(chromosome1.size()!=chromosome2.size()) {
        qDebug()<<"Не совпадают размеры хромосом";
        exit(1);
    }
    vfloat new_chromosome(chromosome1.size());
    for(int i=0; i<new_chromosome.size(); i++) {
        float c_max = SRV::max(chromosome1[i],chromosome2[i]);
        float c_min = SRV::min(chromosome1[i],chromosome2[i]);
        float delta = c_max - c_min;
        float range_min = c_min - delta*A;
        float range_max = c_max + delta*A;
        new_chromosome[i] = RND::random_float(range_min,range_max);
    }
    return new_chromosome;
```

```
}
float RGA::random_mutation(float value) {
    float range_min = value - value*0.2;
    float range_max = value + value*0.2;
    float ret_val = RND::random_float(range_min,range_max);
    return ret_val;
}
vfloat2d RGA::population_mutation(vfloat2d &population, int chance) {
    for(int i=0; i<population.size(); i++) {
        int rand_val = RND::rand_A_B(0,100-chance);
        if(rand_val == 0) {
            for(int j=0; j<population.at(i).size(); j++) {
                population[i][j] = random_mutation(population.at(i).at(j));
            }
        }
    }
    return population;
}
#ifndef RND_H
#define RND_H
#include <CNN/cnn.h>
#include <fcvector.h>
#include <QTime>
class RND
{
public:
    static cnn_data get_randon_data();
    static vfloat2d random_vfloat2d(int height, int width, float range_min, float
range_max);
    static float random_float(float range_min, float range_max);
```

```
    static float rand_A_B(int A, int B);
};
#endif // RND_H
#include "rnd.h"
float RND::rand_A_B(int A, int B) {
    //qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));
    return qrand()%(B-A+1)+A;
}
cnn_data CNN::get_randon_data() {
    float range_min = -0.5;
    float range_max = 0.5;
    int kernel_1_size = 5;
    cnn_data data;
    for(int i=0; i<5; i++) {
        data.kernel1.append(random_vfloat2d(kernel_1_size,kernel_1_size,range_min,r
ange_max));
    }
    for(int i=0; i<5; i++) {
        data.bias1[i] = random_float(range_min,range_max);
    }
    for(int i=0; i<5; i++) {
        data.weight2[i] = random_float(range_min,range_max);
    }
    for(int i=0; i<5; i++) {
        data.bias2[i] = random_float(range_min,range_max);
    }
    int kernel_3_size = 3;
    for(int i=0; i<30; i++) {
        data.kernel3[i] =
random_vfloat2d(kernel_3_size,kernel_3_size,range_min,range_max);
```

```
    }
    for(int i=0; i<20; i++) {
        data.bias3[i] = random_float(range_min,range_max);
    }
    for(int i=0; i<20; i++) {
        data.weight4[i] = random_float(range_min,range_max);
    }
    for(int i=0; i<20; i++) {
        data.bias4[i] = random_float(range_min,range_max);
    }
    for(int i=0; i<20; i++) {
        data.bias5[i] = random_float(range_min,range_max);
    }
    int weight_5_height = 6;
    int weight_5_width = 7;
    for(int i=0; i<20; i++) {
        data.weight5[i] =
random_vfloat2d(weight_5_height,weight_5_width,range_min,range_max);
    }
    for(int i=0; i<20; i++) {
        data.weight6[i] = random_float(range_min,range_max);
    }
    data.bias6_1 = random_float(range_min,range_max);
    return data;
}
vfloat2d RND::random_vfloat2d(int height, int width, float range_min, float
range_max) {
    //qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));
    vfloat2d ret_val;
    ret_val.resize(height);
```

```
    for(int i=0; i<ret_val.size(); i++) {
        ret_val[i].resize(width);
        for(int j=0; j<ret_val.at(i).size(); j++) {
            ret_val[i][j] = random_float(range_min,range_max);
        }
    }
    return ret_val;
}
float RND::random_float(float range_min, float range_max) {
    //qsrand(QTime(0,0,0).secsTo(QTime::currentTime()));
    float random_val = (float) qrand()/RAND_MAX;
    float ret_val = range_min + (range_max - range_min) * random_val;
    return ret_val;
}
#ifndef SRV_H
#define SRV_H
class SRV
{
public:
    static float min(float a, float b);
    static float max(float a, float b);
};
#endif // SRV_H
#include "srv.h"
float SRV::max(float a, float b) {
    if(a>b) {
        return a;
    } else {
        return b;
    }
```

```
}
float SRV::min(float a, float b) {
    if(a<b) {
        return a;
    } else {
        return b;
    }
}
#ifndef TRAINER_H
#define TRAINER_H
#include <CNN/cnn.h>
#include <rga.h>
#include <fcvector.h>
#include <fcfile.h>
#include <rnd.h>
#include <omp.h>
class Trainer
{
public:
    cnn_data rga_train(list_vfloat2d &in, vfloat2d &target, vfloat2d &population, float
rms_error, float blx_koef, int step, int max_step);
    cnn_data image_rga_train(QString faces_dir, QString no_faces_dir);
};
#endif // TRAINER_H
#include "trainer.h"
n_net neural::rga_train(n_net &neuro_net, v2double &in, v2double &target,
v2double &population, double rms_error, double blx_koef, int step, int max_step) {
    qDebug()<<"Train Started";
    qDebug()<<"Real Coded GA + Simulate Annealing";
    double T = 100000;
```

```
double alfa = 0.99;
double P_start = 100;
double P = P_start;
double E = 0.5;
n_net ret_net;
step = 0;
double error = 0;
vdouble all_errors(population.size());
for(int j=0; j<in.size(); j++) {
    //qDebug()<<"calc errors";
    v2double outputs;
    for(int i=0; i<population.size(); i++) {
        n_net temp_net = set_W(neuro_net,population[i]);
        temp_net = start_net(in[j],temp_net);
        outputs.append(temp_net.last()->output);
    }
    vdouble errors = neural_rga.calculate_fitness(outputs,target[j]);
    all_errors = vectr.vdouble_sum(all_errors,errors);
    error = error + vectr.vdouble_sum_elem(errors) / errors.size();
}
while(error>E) {
    population = neural_rga.tournament_selection(population,all_errors);//ТУРНИР
    population = neural_rga.random_blx(population, blx_koef);
    population = neural_rga.population_mutation(population,P);
    error = 0;
    all_errors.clear();
    all_errors.resize(population.size());
    int best;
    for(int j=0; j<in.size(); j++) {
        //qDebug()<<"calc errors";
```

```
            v2double outputs;
            for(int i=0; i<population.size(); i++) {
                n_net temp_net = set_W(neuro_net,population[i]);
                temp_net = start_net(in[j],temp_net);
                outputs.append(temp_net.last()->output);
            }
            best = neural_rga.best_chromosome(outputs,target[j]);
            ret_net = set_W(neuro_net,population[best]);
            vdouble errors = neural_rga.calculate_fitness(outputs,target[j]);
            all_errors = vectr.vdouble_sum(all_errors,errors);
            error = error + vectr.vdouble_sum_elem(errors) / errors.size();
        }
        P = P_start * exp(-1/T);
        T = T * alfa;
        qDebug()<<"Error="<<error<<"Temperature="<<T<<"Chance="<<P;
    }
    return ret_net;
}
cnn_data Trainer::image_rga_train(QString faces_dir, QString no_faces_dir) {
    float rms_error = 0.5;
    float blx_a_koef = 0.5;
    float rnd_start_range = -0.3;
    float rnd_end_range = 0.3;
    int populatin_size = 20;
    int chromosome_size = 1351;
    vfloat2d population;
    for(int i=0; i<populatin_size; i++) {
        vfloat chromosome(chromosome_size);
        for(int j=0; j<chromosome.size(); j++) {
            chromosome[j] = RND::random_float(rnd_start_range,rnd_end_range);
```

```
    }
    population.append(chromosome);
}
list_vfloat2d faces_data = FCFile::openFaces(faces_dir);
vfloat2d faces_target;
for(int i=0; i<faces_data.size(); i++) {
    vfloat target;
    target.append(1);
    faces_target.append(target);
}
list_vfloat2d no_faces_data = FCFile::openFaces(no_faces_dir);
vfloat2d no_faces_target;
for(int i=0; i<no_faces_data.size(); i++) {
    vfloat target;
    target.append(-1);
    no_faces_target.append(target);
}
list_vfloat2d input_data;
for(int i=0; i<faces_data.size(); i++) {
    input_data.append(faces_data.at(i));
}
for(int i=0; i<no_faces_data.size(); i++) {
    input_data.append(no_faces_data.at(i));
}
vfloat2d target_data;
for(int i=0; i<faces_target.size(); i++) {
    target_data.append(faces_target.at(i));
}
for(int i=0; i<no_faces_target.size(); i++) {
    target_data.append(no_faces_target.at(i));
```

```
    }
    cnn_data net =
rga_train(input_data,target_data,population,rms_error,blx_a_koef,0,1001);
    return net;
}
```