计算机视觉期末项目

16340220 王培钰 电子政务

目录：

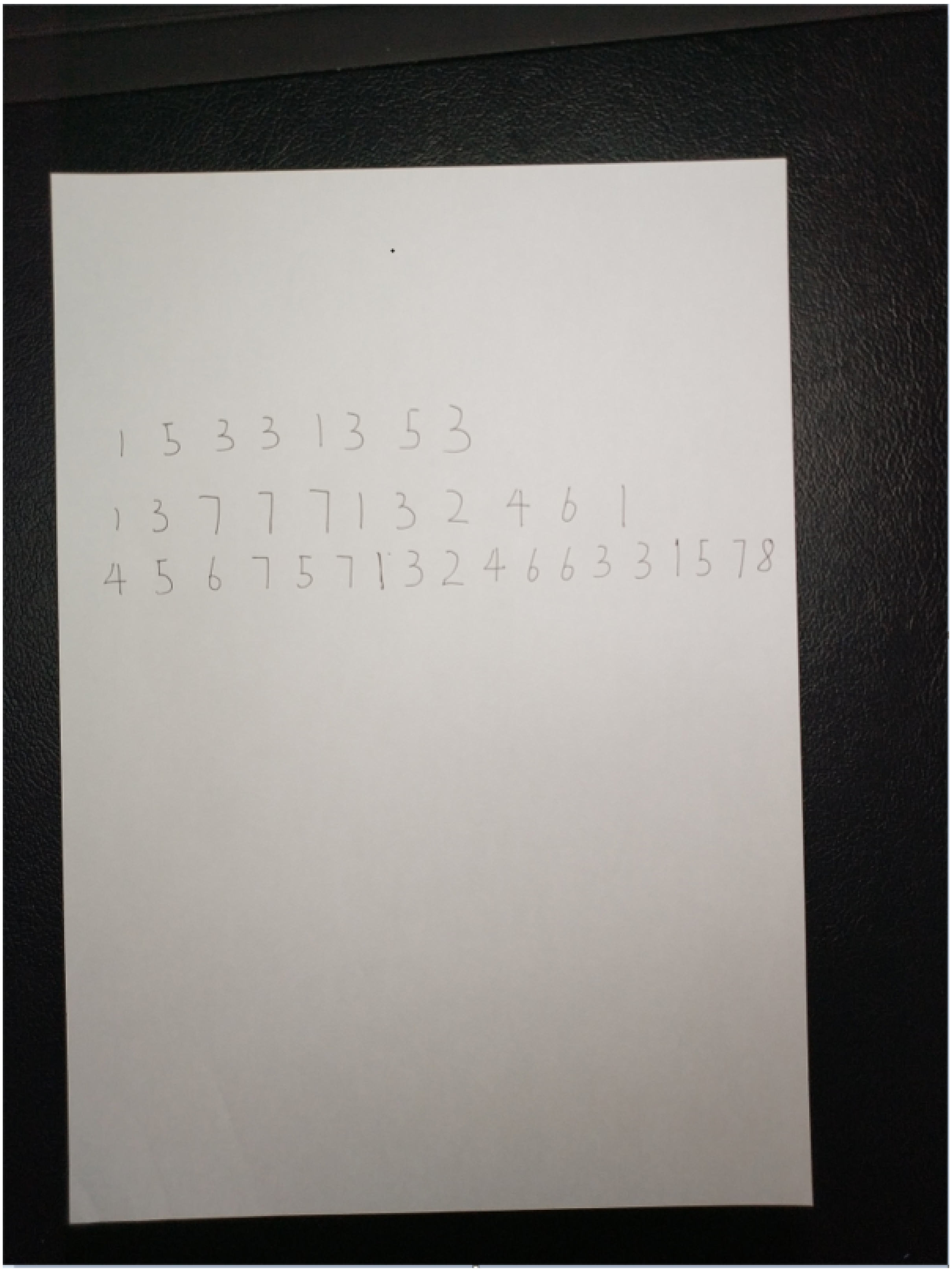
一. 实验内容

1. 输入图像

普通 **A4** 打印纸，上面有手写的如下信息:

1. 学号
2. 手机号
3. 身份证号

所有输入格式一致, 数字号码不粘连, 但是拍照时可能角度不正。



1 5 3 3 1 3 5 3

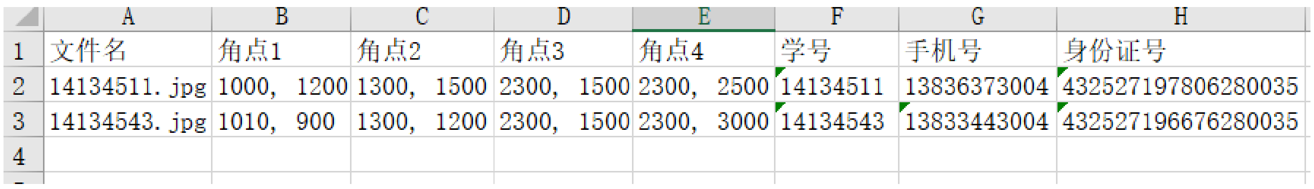
1 3 7 7 7 1 3 2 4 6 1

4 5 6 7 5 7 1 3 2 4 6 6 3 3 1 5 7 8

2. 输出要求

1. 根据标准流程输出每一个主要步骤的结果，包括 A4 纸张的矫正结果，行数据（包括学号、手机号、身份证号）的切割，单个字符的切割结果。
2. 对上面的 A4 纸的四个角、学号、手机号、身份证号进行识别, 识别结果保存到Excel 表格(xlsx 格式), 对于手写体数字字符的训练数据可以使用 MNIST.

格式入下:



	A	B	C	D	E	F	G	H
1	文件名	角点1	角点2	角点3	角点4	学号	手机号	身份证号
2	14134511.jpg	1000，1200	1300，1500	2300，1500	2300，2500	14134511	13836373004	432527197806280035
3	14134543.jpg	1010，900	1300，1200	2300，1500	2300，3000	14134543	13833443004	432527196676280035
4								

二. 开发环境

1. 操作系统

- windows系统
- MinGW g++(CImg)
- Anaconda python3.5(tensorflow)

2. 编译命令

- **c++:**

```
g++ main.cpp canny.cpp canny.h hough.cpp hough.h warp.cpp warp.h ImageSegmentation.cpp
ImageSegmentation.h function.cpp function.h -O2 -lgdi32 -std=gnu++11
```

- **python:**
  - **训练model：** `python train.py`
  - **验证结果：** `python inference.py`

三. 实验过程

1. 边缘检测

(1). 灰度变换

```
CImg<float> rgb2gray(CImg<float> picture) {
    CImg<float> graypic;
    graypic.resize(picture._width, picture._height, 1, 1, 0);
    cimg_forXY(graypic, x, y) {
        float R = picture(x,y,0);
        float G = picture(x,y,1);
        float B = picture(x,y,2);
        float Gray = (R * 299 + G * 587 + B * 114 + 500) / 1000;
        graypic(x,y) = Gray;
    }
    return graypic;
}
```

(2). 高斯滤波

```
blurImg = grayImg.get_blur(BLUR);
```

(3). sobel算子计算梯度

```
void canny::sobel() {
    sobelImg.resize(blurImg._width, blurImg._height, 1, 1, 0);
    angleImg = sobelImg;
    CImg_3x3(I, float);
    cimg_for3x3(blurImg, x, y, 0, 0, I, float) {
        float ix = (Inp + 2 * Inc + Inn) - (Ipp + 2 * Ipc + Ipn);
        float iy = (Ipp + 2 * Icp + Inp) - (Ipn + 2 * Icn + Inn);
        float grad = sqrt(ix * ix + iy * iy);
        angleImg(x, y) = (ix == 0) ? 90 : atan(iy / ix);
        if (grad > 255) {
            grad = 255;
        }
        if (grad < 0) {
            grad = 0;
        }
        sobelImg(x, y) = grad;
    }
}
```

(4). 非极大值抑制

```
void canny::nonMaxSupp() {
    nonImg = sobelImg;
    cimg_for_insideXY(nonImg, x, y, 1) {
        // horizontal edge
        if (((-22.5 < angleImg(x, y)) && (angleImg(x, y) <= 22.5)) || ((157.5 < angleImg(x, y))
&& (angleImg(x, y) <= -157.5))) {
            if (sobelImg(x, y) < sobelImg(x + 1, y) || sobelImg(x, y) < sobelImg(x - 1, y))
                nonImg(x, y) = 0;
        }
        // vertical edge
        if (((-112.5 < angleImg(x, y)) && (angleImg(x, y) <= -67.5)) || ((67.5 < angleImg(x, y))
&& (angleImg(x, y) <= 112.5))) {
            if (sobelImg(x, y) < sobelImg(x, y + 1) || sobelImg(x, y) < sobelImg(x, y - 1))
                nonImg(x, y) = 0;
        }
        // -45 degree edge
        if (((-67.5 < angleImg(x, y)) && (angleImg(x, y) <= -22.5)) || ((112.5 < angleImg(x, y))
&& (angleImg(x, y) <= 157.5))) {
            if (sobelImg(x, y) < sobelImg(x + 1, y - 1) || sobelImg(x, y) < sobelImg(x - 1, y +
1))
                nonImg(x, y) = 0;
        }
        // 45 degree edge
        if (((-157.5 < angleImg(x, y)) && (angleImg(x, y) <= -112.5)) || ((22.5 < angleImg(x,
y)) && (angleImg(x, y) <= 67.5))) {
```

```
            if (sobelImg(x, y) < sobelImg(x + 1, y + 1) || sobelImg(x, y) < sobelImg(x - 1, y - 
1))
                nonImg(x, y) = 0;
        }
    }
}
```

(5). 双阈值检测

```
void canny::threshold(int low_thres, int high_thres) {
    thresImg = nonImg;
    cimg_forXY(thresImg, x, y) {
        if (thresImg(x, y) > high_thres) {
            thresImg(x, y) = 255;
        }
        else if (thresImg(x, y) < low_thres) {
            thresImg(x, y) = 0;
        }
        else {
            bool anyHigh = false;
            bool anyBetween = false;
            for (int i = x - 1; i <= x + 1; ++i) {
                for (int j = y - 1; j <= y + 1; ++j) {
                    if (i < 0 || i >= thresImg._width || j < 0 || j >= thresImg._height) {
                        continue;
                    }
                    else {
                        if (thresImg(i, j) > high_thres) {
                            thresImg(x, y) = 255;
                            anyHigh = true;
                            break;
                        }
                        else if (thresImg(i, j) <= high_thres && thresImg(i, j) >= low_thres) {
                            anyBetween = true;
                        }
                    }
                }
                if (anyHigh) break;
            }
            if (!anyHigh && anyBetween) {
                for (int i = x - 2; i <= x + 2; ++i) {
                    for (int j = y - 2; j <= y + 2; ++j) {
                        if (i < 0 || i >= thresImg._width || j < 0 || j >= thresImg._height) {
                            continue;
                        }
                        else {
                            if (thresImg(i, j) > high_thres) {
                                thresImg(x, y) = 255;
                                anyHigh = true;
                                break;
                            }
                        }
                    }
                    if (anyHigh) {
                        break;
                    }
                }
            }
            if (!anyHigh) {
                thresImg(x, y) = 0;
            }
        }
    }
}
```

2. 霍夫变换检测直线

(1). Hough空间变换

```
void hough::Hough_Statistics() {
    double w = Img._width;
    double h = Img._height;
    double center_x = w/2;
    double center_y = h/2;
    double hough_h = ((sqrt(2.0) * (double)(h>w?h:w)) / 2.0);
    houghImg.resize(180, hough_h * 2, 1, 1, 0);
    cimg_forXY(cannyImg, x, y) {
        if (cannyImg(x,y) != 0) {
            cimg_forX(houghImg, angle) {
                double _angle = (double)PI*angle / 180.0f;
                int polar = (int)((((double)x - center_x)*cos(_angle) + ((double)y -
center_y)*sin(_angle)) + hough_h);
                houghImg(angle, polar) += 1;
            }
        }
    }
}
```

(2). 投票算法选出四条直线

```
void hough::GetLine() {
    resultImg = Img;
    //剔除掉可能出现的重合线，方法是取9x9空间内的霍夫最大值
    int hough_h = houghImg._height;
    int img_h = Img._height;
    int img_w = Img._width;
    const int y_min = 0;
    const int y_max = Img._height - 1;
    const int x_min = 0;
    const int x_max = Img._width - 1;
    cimg_forXY(houghImg, angle, polar) {
        if (houghImg(angle, polar) >= Min_thres) {
            int max = houghImg(angle, polar);
            for(int ly=-DIFF;ly<=DIFF;ly++) {
                for(int lx=-DIFF;lx<=DIFF;lx++) {
                    if( (ly+polar>=0 && ly+polar<houghImg._height) && (lx+angle>=0 &&
lx+angle<houghImg._width) ) {
                        if( (int)houghImg(angle + lx, polar + ly ) > max ) {
                            max = houghImg(angle + lx, polar + ly );
                            ly = lx = DIFF + 1;
                        }
                    }
                }
            }
            if (max > (int)houghImg(angle, polar) )
                continue;
            peaks.push_back(pair< pair<int, int>, int >(pair<int, int>(angle, polar),
houghImg(angle, polar)));
        }
    }
    sort(peaks.begin(), peaks.end(), sortCmp);
    for (int i = 0; lines.size() != 4; i++) {
        int angle = peaks[i].first.first;
        int polar = peaks[i].first.second;
        int x1, y1, x2, y2;
        x1 = y1 = x2 = y2 = 0;
        double _angle = (double)PI*angle / 180.0f;
        if(angle >= 45 && angle <= 135) {
            x1 = 0;
            y1 = ((double)(polar-(hough_h/2)) - ((x1 - (img_w/2) ) * cos(_angle))) / sin(_angle)
+ (img_h / 2);
            x2 = img_w;
            y2 = ((double)(polar-(hough_h/2)) - ((x2 - (img_w/2) ) * cos(_angle))) / sin(_angle)
+ (img_h / 2);
        }
        else {
            y1 = 0;
            x1 = ((double)(polar-(hough_h/2)) - ((y1 - (img_h/2) ) * sin(_angle))) / cos(_angle)
+ (img_w / 2);
            y2 = img_h;
            x2 = ((double)(polar-(hough_h/2)) - ((y2 - (img_h/2) ) * sin(_angle))) / cos(_angle)
+ (img_w / 2);
        }
        bool flag = true;
        for (int k = 0; k < lines.size(); k++) {
            if (distance((float)(lines[k].first.first - x1), (float)(lines[k].first.second -
y1)) < 100 && distance((float)(lines[k].second.first - x2), (float)(lines[k].second.second -
y2)) < 100) {
                flag = false;
                break;
            }
        }
        if (flag == true) {
            lines.push_back(pair< pair<int, int>, pair<int, int> >(pair<int, int>(x1, y1), pair<int, int>(x2, y2)));
        }
    }
    for (int i = 0; i < lines.size(); i++) {
        resultImg.draw_line(lines[i].first.first, lines[i].first.second, lines[i].second.first, lines[i].second.second, Red);
    }
}
```

(3). 绘制角点

```
void hough::GetVertexs() {
    for (int i = 0; i < lines.size(); i++) {
        double k0, b0;
        if (lines[i].first.first == lines[i].second.first) {
            k0 = DBL_MAX;
            b0 = lines[i].first.first;
        }
        else {
            k0 = (double) (lines[i].first.second - lines[i].second.second) / (lines[i].first.first - lines[i].second.first);
            b0 = (double) (lines[i].first.second * lines[i].second.first - lines[i].second.second * lines[i].first.first) / (lines[i].second.first - lines[i].first.first);
        }
        for (int j = i + 1; j < lines.size(); j++) {
            double k1, b1;
            if (lines[j].first.first == lines[j].second.first) {
                k1 = DBL_MAX;
                b1 = lines[j].first.first;
            }
            else {
                k1 = (double) (lines[j].first.second - lines[j].second.second) / (lines[j].first.first - lines[j].second.first);
                b1 = (double) (lines[j].first.second * lines[j].second.first - lines[j].second.second * lines[j].first.first) / (lines[j].second.first - lines[j].first.first);
            }
            if (k0 == k1)
                continue;
            if (k0 == DBL_MAX) {
                int _x = b0, _y = k1 * b0 + b1;
                if (_x >= 0 && _x < Img._width && _y >= 0 && _y < Img._height)
                    vertex.push_back(make_pair(_x, _y));
                continue;
            }
            if (k1 == DBL_MAX) {
                int _x = b1, _y = k0 * b1 + b0;
                if (_x >= 0 && _x < Img._width && _y >= 0 && _y < Img._height)
                    vertex.push_back(make_pair(_x, _y));
                continue;
            }
            int _x = (b0 - b1) / (k1 - k0);
```

```
                int _y = (k0 * b1 - k1 * b0) / (k0 - k1);
                if (_x >= 0 && _x < Img._width && _y >= 0 && _y < Img._height)
                    vertex.push_back(make_pair(_x, _y));
            }
        }
        for (int i = 0; i < vertex.size(); i++) {
            cout << vertex[i].first << "   ...   " << vertex[i].second << endl;
            resultImg.draw_circle(vertex[i].first, vertex[i].second, 50, Red);
        }
    }
```

3. A4纸矫正

(1). 角点排序

```
void warp::orderVertexs() {
    sort(vertex.begin(), vertex.end(), sortCmp);
    double w = distance(vertex[0].first - vertex[1].first, vertex[0].second - vertex[1].second);
    double h = distance(vertex[0].first - vertex[2].first, vertex[0].second - vertex[2].second);
    //纸张是横向的
    if (vertex[1].first < vertex[2].first && h > w) {
        swap(vertex[1], vertex[2]);
        swap(vertex[2], vertex[3]);
        vertex.push_back(vertex[0]);
        vertex.erase(vertex.begin());
    }
    //纸张是竖向的
    else {
        swap(vertex[2], vertex[3]);
    }
}
```

(2). 获取矫正的特征矩阵

```
void warp::calcMatrix() {
  double x0 = vertex[0].first, x1 = vertex[1].first, x2 = vertex[2].first, x3 = vertex[3].first;
  double y0 = vertex[0].second, y1 = vertex[1].second, y2 = vertex[2].second, y3 =
vertex[3].second;
  double dx3 = x0 - x1 + x2 - x3;
  double dy3 = y0 - y1 + y2 - y3;
  if (fabs(dx3) < 10e-5 && fabs(dy3) < 10e-5) {
    M[0] = x1 - x0, M[1] = y1 - y0, M[2] = 0;
    M[3] = x2 - x1, M[4] = y2 - y1, M[5] = 0;
    M[6] = x0, M[7] = y0, M[8] = 1;
  }
  else {
    double dx1 = x1 - x2, dx2 = x3 - x2, dy1 = y1 - y2, dy2 = y3 - y2;
    double det = dx1 * dy2 - dx2 * dy1;
    double a13 = (dx3 * dy2 - dx2 * dy3) / det;
    double a23 = (dx1 * dy3 - dx3 * dy1) / det;
    M[0] = x1 - x0 + a13 * x1, M[1] = y1 - y0 + a13 * y1, M[2] = a13;
    M[3] = x3 - x0 + a23 * x3, M[4] = y3 - y0 + a23 * y3, M[5] = a23;
    M[6] = x0, M[7] = y0, M[8] = 1;
  }
}
```

(3). 比例变换

```
void warp::warping() {
    double P[3];
    resultImg.resize(paper_width, paper_height);
    double width = resultImg.width(), height = resultImg.height();
    cimg_forXY(resultImg, x, y) {
        double _x = x / width, _y = y / height;
        double denominator = M[2] * _x + M[5] * _y + M[8];
        double tx = (M[0] * _x + M[3] * _y + M[6]) / denominator;
        double ty = (M[1] * _x + M[4] * _y + M[7]) / denominator;
        cimg_forC(resultImg, c) {
            resultImg(x,y,c) = Img((int)tx, (int)ty, c);
        }
    }
}
```

4. 单个数字分割

(1). 自适应阈值分割

```
// 图像二值化处理
CImg<float> ImageSegmentation::AdaptiveThreshold(CImg<float> imgIn) {
    int win_length = 16;
    float threshold = 0.08;
    CImg<float> result(imgIn.width(), imgIn.height(), 1, 1, 255);
    //求积分
    CImg<int> integral(imgIn.width(), imgIn.height(), 1, 1, 0);
    cimg_forY(result, y){
        int sum = 0;
        cimg_forX(result, x){
            sum += imgIn(x, y);
            if(y == 0){
                integral(x, y) = sum;
            }
            else{
                integral(x, y) = integral(x, y - 1) + sum;
            }
        }
    }
    //自适应阈值
    cimg_forY(imgIn, y) {
        int y1 = (y - win_length > 0) ?y - win_length : 0;
        int y2 = (y + win_length < imgIn.height()) ? (y + win_length) : (imgIn.height() - 1);
        cimg_forX(imgIn, x) {
            int x1 = (x - win_length > 0) ? x - win_length : 0;
            int x2 = (x + win_length < imgIn.width()) ? (x + win_length) : (imgIn.width() - 1);
            int count = (x2 - x1) * (y2 - y1);
            int sum = integral(x2, y2) - integral(x1, y2) - integral(x2, y1) + integral(x1, y1);
            if (imgIn(x, y) * count < sum * (1.0 - threshold)) {
                result(x, y) = 0;
            }
            else {
                result(x, y) = 255;
            }
        }
    }
    //删除边缘部分的黑边
    cimg_for_borderXY(imgIn, x, y, border_DIFF) {
        result(x, y) = 255; // 白色
    }
    return result;
}
```

(2). 垂直方向边缘分割

```
void ImageSegmentation::findDividingLine() {
    histogramImg = CImg<float>(binaryImg._width, binaryImg._height, 1, 3, 255);
    dividingImg = binaryImg;
    vector<int> inflectionPoints; // 拐点
    cimg_forY(histogramImg, y) {
        int blackPixel = 0;
        cimg_forX(binaryImg, x) {
            if (binaryImg(x, y, 0) == 0)
                blackPixel++;
        }
        cimg_forX(histogramImg, x) {
            if (x < blackPixel) {
                histogramImg(x, y, 0) = 0;
                histogramImg(x, y, 1) = 0;
                histogramImg(x, y, 2) = 0;
            }
        }

        // 求Y方向直方图，谷的最少黑色像素个数为0
        // 判断是否为拐点
        if (y > 0) {
            // 下白上黑：取下
            if (blackPixel <= 0 && histogramImg(0, y - 1, 0) == 0)
                inflectionPoints.push_back(y);
            // 下黑上白：取上
            else if (blackPixel > 0 && histogramImg(0, y - 1, 0) != 0)
                inflectionPoints.push_back(y - 1);
        }
    }

    dividePoints.push_back(Point(0, -1));
    // 两拐点中间做分割
    if (inflectionPoints.size() > 2) {
        for (int i = 1; i < inflectionPoints.size() - 1; i = i + 2) {
            int dividePoint = (inflectionPoints[i] + inflectionPoints[i + 1]) / 2;
            dividePoints.push_back(Point(0, dividePoint));
        }
    }
    dividePoints.push_back(Point(0, binaryImg._height - 1));
}
// 根据行分割线划分图片
void ImageSegmentation::divideIntoBarItemImg() {
    vector<Point> tempDivideLinePointSet;
    for (int i = 1; i < dividePoints.size(); i++) {
        int barHeight = dividePoints[i].y - dividePoints[i - 1].y;
        int blackPixel = 0;
        CImg<float> barItemImg = CImg<float>(binaryImg._width, barHeight, 1, 1, 0);
        cimg_forXY(barItemImg, x, y) {
            barItemImg(x, y, 0) = binaryImg(x, dividePoints[i - 1].y + 1 + y, 0);
            if (barItemImg(x, y, 0) == 0)
                blackPixel++;
        }
        double blackPercent = (double)blackPixel / (double)(binaryImg._width * barHeight);
        // 只有当黑色像素个数超过图像大小一定比例0.001时，才可视作有数字
        if (blackPercent > 0.001) {
            vector<int> dividePosXset = DivideLineXofSubImage(barItemImg);
            vector< CImg<float> > rowItemImgSet = RowItemImg(barItemImg, dividePosXset);

            for (int j = 0; j < rowItemImgSet.size(); j++) {
                subImageSet.push_back(rowItemImgSet[j]);
                tempDivideLinePointSet.push_back(Point(dividePosXset[j], dividePoints[i -
1].y));
            }
        }
    }

    dividePoints.clear();
    for (int i = 0; i < tempDivideLinePointSet.size(); i++)
        dividePoints.push_back(tempDivideLinePointSet[i]);
}
```

(3). 图像膨胀

```
// 图像膨胀
int ImageSegmentation::Dilate(const CImg<float>& Img, int x, int y) {
    int intensity = Img(x, y, 0);
    if (intensity == 255) {
        for (int i = -1; i <= 1; i++) {
            for (int j = -1; j <= 1; j++) {
                if (0 <= x + i && x + i < Img._width && 0 <= y + j && y + j < Img._height) {
                    if (i != -1 && j != -1 || i != 1 && j != 1 || i != 1 && j != -1 || i != -1
&& j != 1)
                        if (Img(x + i, y + j, 0) == 0) {
                            intensity = 0;
                            break;
                        }
                }
            }
            if (intensity != 255)
                break;
        }
    }
    return intensity;
}
void ImageSegmentation::toDilate(int barItemIndex) {
    //扩张Dilation XY方向
    CImg<float> picture = CImg<float>(subImageSet[barItemIndex]._width,
subImageSet[barItemIndex]._height, 1, 1, 0);
    cimg_forXY(subImageSet[barItemIndex], x, y) {
        picture(x, y, 0) = Dilate(subImageSet[barItemIndex], x, y);
    }

    subImageSet[barItemIndex] = picture;
}
```

(4). 连通区域标记切割数字

```
// 连通区域标记算法
void ImageSegmentation::connectedRegionsTagging(int barItemIndex) {
    tagImg = CImg<float>(subImageSet[barItemIndex]._width, subImageSet[barItemIndex]._height, 1,
1, 0);
    tagAccumulate = -1;

    cimg_forX(subImageSet[barItemIndex], x)
        cimg_forY(subImageSet[barItemIndex], y) {
        // 第一行和第一列
        if (x == 0 || y == 0) {
            int intensity = subImageSet[barItemIndex](x, y, 0);
            if (intensity == 0) {
                addNewTag(x, y, barItemIndex);
            }
        }
        // 其余的行和列
        else {
            int intensity = subImageSet[barItemIndex](x, y, 0);
            if (intensity == 0) {
                // 检查正上、左上、左中、左下这四个邻点
                int minTag = INT_MAX; //最小的tag
                Point minTagPointPos(-1, -1);
                // 先找最小的标记
                findMinTag(x, y, minTag, minTagPointPos, barItemIndex);
                // 当正上、左上、左中、左下这四个邻点有黑色点时，合并；
                if (minTagPointPos.x != -1 && minTagPointPos.y != -1) {
                    mergeTagImageAndList(x, y - 1, minTag, minTagPointPos, barItemIndex);
                    for (int i = -1; i <= 1; i++) {
                        if (y + i < subImageSet[barItemIndex]._height)
                            mergeTagImageAndList(x - 1, y + i, minTag, minTagPointPos,
barItemIndex);
                    }
                    // 当前位置
                    tagImg(x, y, 0) = minTag;
                    Point cPoint(x + dividePoints[barItemIndex].x + 1, y +
dividePoints[barItemIndex].y + 1);
                    pointPosListSet[minTag].push_back(cPoint);

                }
                // 否则，作为新类
                else {
                    addNewTag(x, y, barItemIndex);
                }
            }
        }
    }
}
// 添加新的类tag
void ImageSegmentation::addNewTag(int x, int y, int barItemIndex) {
    tagAccumulate++;
    tagImg(x, y, 0) = tagAccumulate;
    classTagSet.push_back(tagAccumulate);
    list<Point> pList;
    Point cPoint(x + dividePoints[barItemIndex].x + 1, y + dividePoints[barItemIndex].y + 1);
    pList.push_back(cPoint);
    pointPosListSet.push_back(pList);
}
// 在正上、左上、正左、左下这四个邻点中找到最小的tag
void ImageSegmentation::findMinTag(int x, int y, int &minTag, Point &minTagPointPos, int
barItemIndex) {
    // 正上
    if (subImageSet[barItemIndex](x, y - 1, 0) == 0) {
        if (tagImg(x, y - 1, 0) < minTag) {
            minTag = tagImg(x, y - 1, 0);
            minTagPointPos.x = x;
            minTagPointPos.y = y - 1;
        }
    }
    // 左上、左中、左下
    for (int i = -1; i <= 1; i++) {
        if (y + i < subImageSet[barItemIndex]._height) {
            if (subImageSet[barItemIndex](x - 1, y + i, 0) == 0 && tagImg(x - 1, y + i, 0) <
minTag) {
                minTag = tagImg(x - 1, y + i, 0);
                minTagPointPos.x = x - 1;
                minTagPointPos.y = y + i;
            }
        }
    }
}
// 合并某个点(x,y)所属类别
void ImageSegmentation::mergeTagImageAndList(int x, int y, const int minTag, const Point
minTagPointPos, int barItemIndex) {
    // 赋予最小标记，合并列表
    if (subImageSet[barItemIndex](x, y, 0) == 0) {
        int tagBefore = tagImg(x, y, 0);
        if (tagBefore != minTag) {
            //把所有同一类的tag替换为最小tag、把list接到最小tag的list
            list<Point>::iterator it = pointPosListSet[tagBefore].begin();
            for (; it != pointPosListSet[tagBefore].end(); it++) {
                tagImg((*it).x - dividePoints[barItemIndex].x - 1, (*it).y -
dividePoints[barItemIndex].y - 1, 0) = minTag;
            }
            pointPosListSet[minTag].splice(pointPosListSet[minTag].end(),
pointPosListSet[tagBefore]);
        }
    }
}
```

5. mnist数字检测

(1). CNN实现MNIST手写识别模型

1、输入数据

- 直接使用tensorflow中的模块，导入输入数据：

```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

或者使用官方提供的input_data.py文件下载mnist数据

2、启动session

1. 交互方式启动session

```
sess = tf.InteractiveSession()
```

2. 一般方式启动session

```
sess = tf.Session()
```

ps: 使用交互方式不用提前构建计算图，而使用一般方式必须提前构建好计算图才能启动session

3、权重和偏置初始化

权重初始化的原则：应该加入少量的噪声来打破对称性并且要避免0梯度（初始化为0）

权重初始化一般选择均匀分布或是正态分布

定义权重初始化方法：

```
def weight_variable(shape):

        #截尾正态分布,stddev是正态分布的标准偏差

        initial = tf.truncated_normal(shape=shape, stddev=0.1)

        return tf.Variable(initial)
```

定义偏置初始化方法：

```
def bias_variable(shape):

        initial = tf.constant(0.1, shape=shape)

        return tf.Variable(initial)
```

4、定义卷积和池化方法

TensorFlow在卷积和Pooling上有很强的灵活性。在这个实例里，我们的卷积使用1步长（stride size），0填充模块（zero padded），保证输出和输入是同一个大小。我们的pooling用简单传统的2x2大小的模板做maxpooling。为了代码更简洁，我们把这部分抽象成一个函数。

```
def conv2d(x, W):

        return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1],  padding='SAME')

def max_pool_2x2(x):

        return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],padding='SAME')
```

(2). 训练model

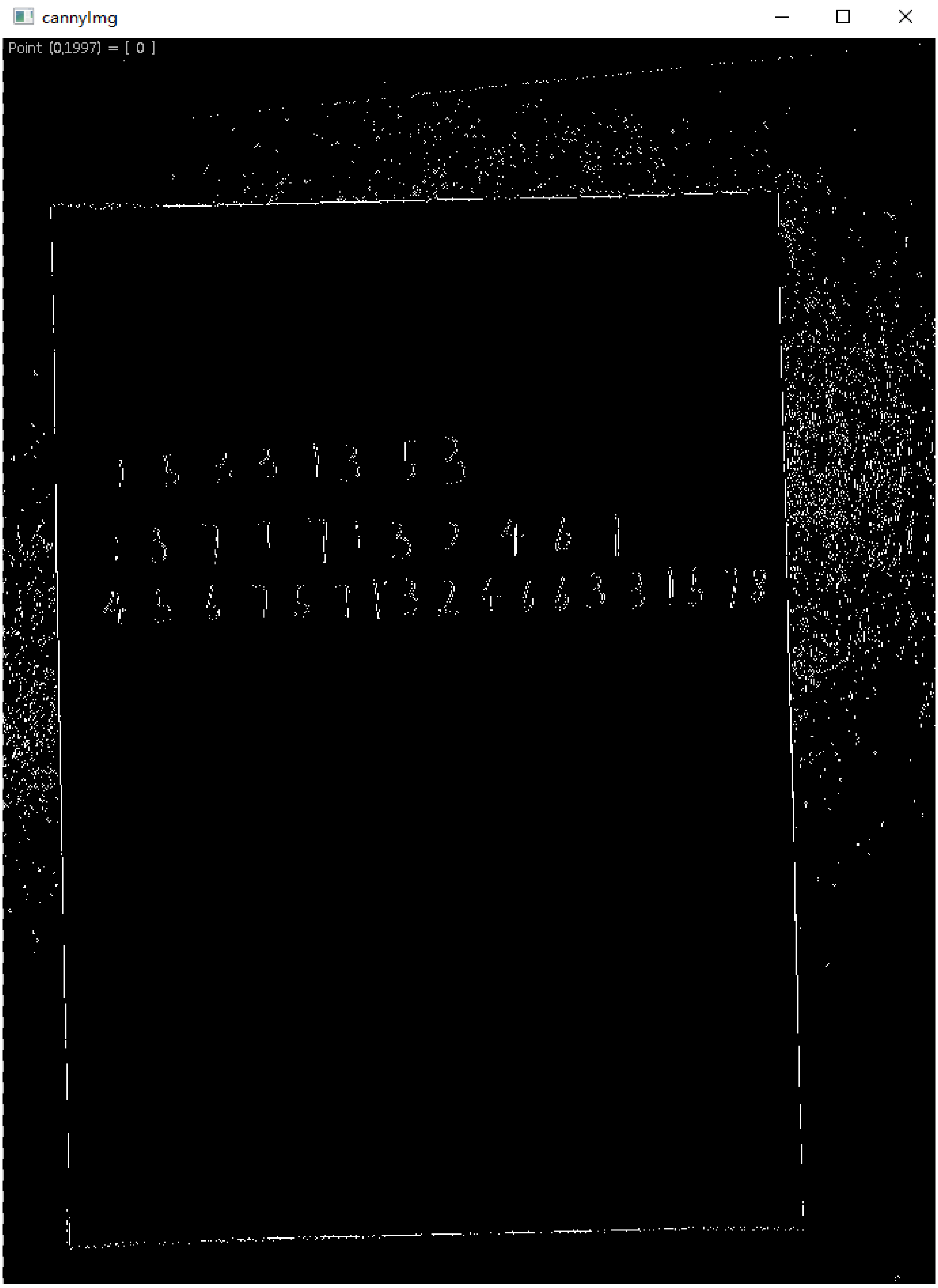
```
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g"%(i, train_accuracy))

train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
saver.save(sess, './model/model.ckpt')
```

(3). 测试验证

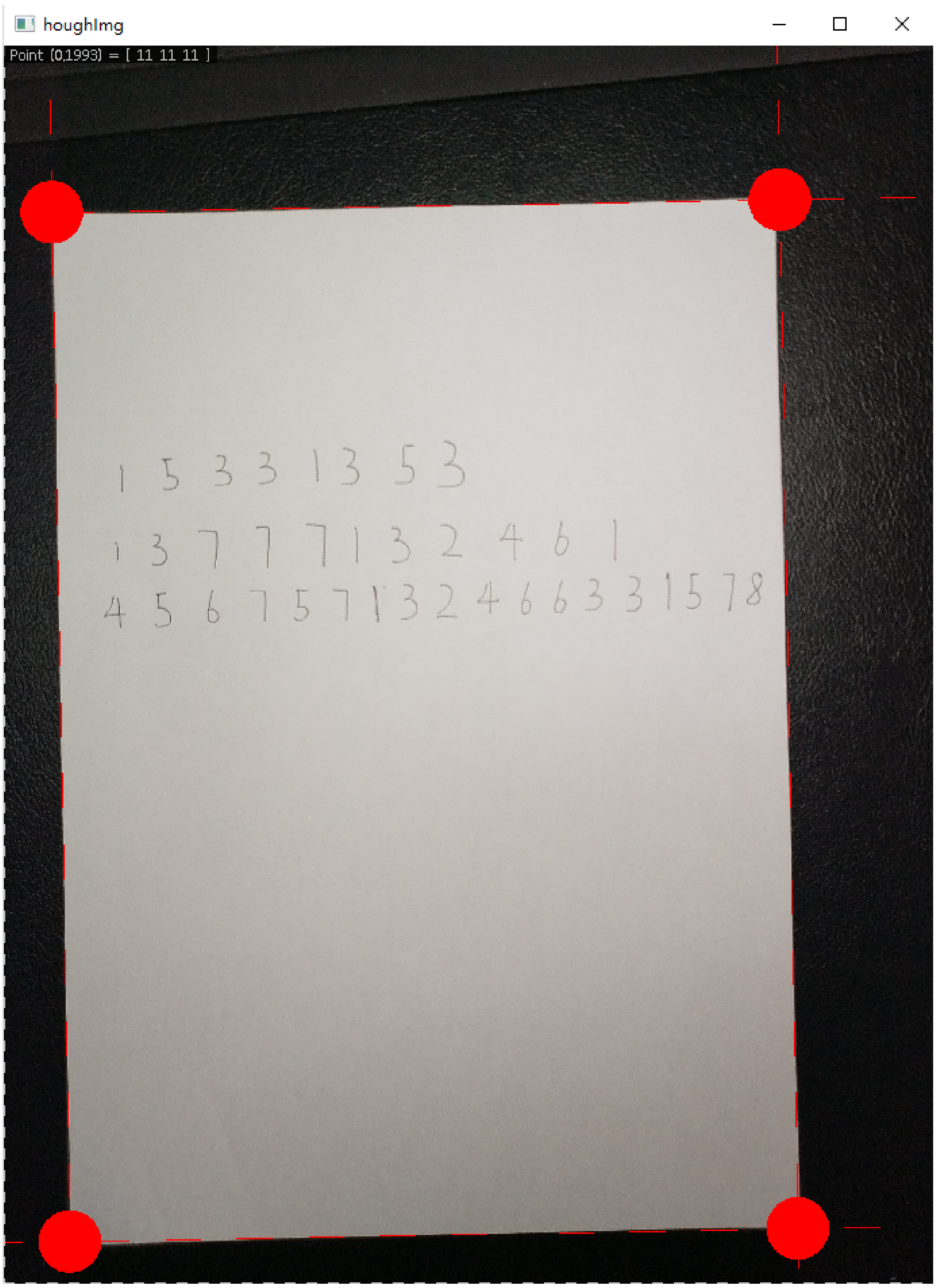
```
with tf.Session() as sess:
    sess.run(init_op)
    saver.restore(sess, "./model/model.ckpt")#这里使用了之前保存的模型参数

    prediction=tf.argmax(y_conv,1)
    predint=prediction.eval(feed_dict={x: result, keep_prob: 1.0}, session=sess)
    print('Predictive value is: ', predint)
```

四. 各阶段效果图

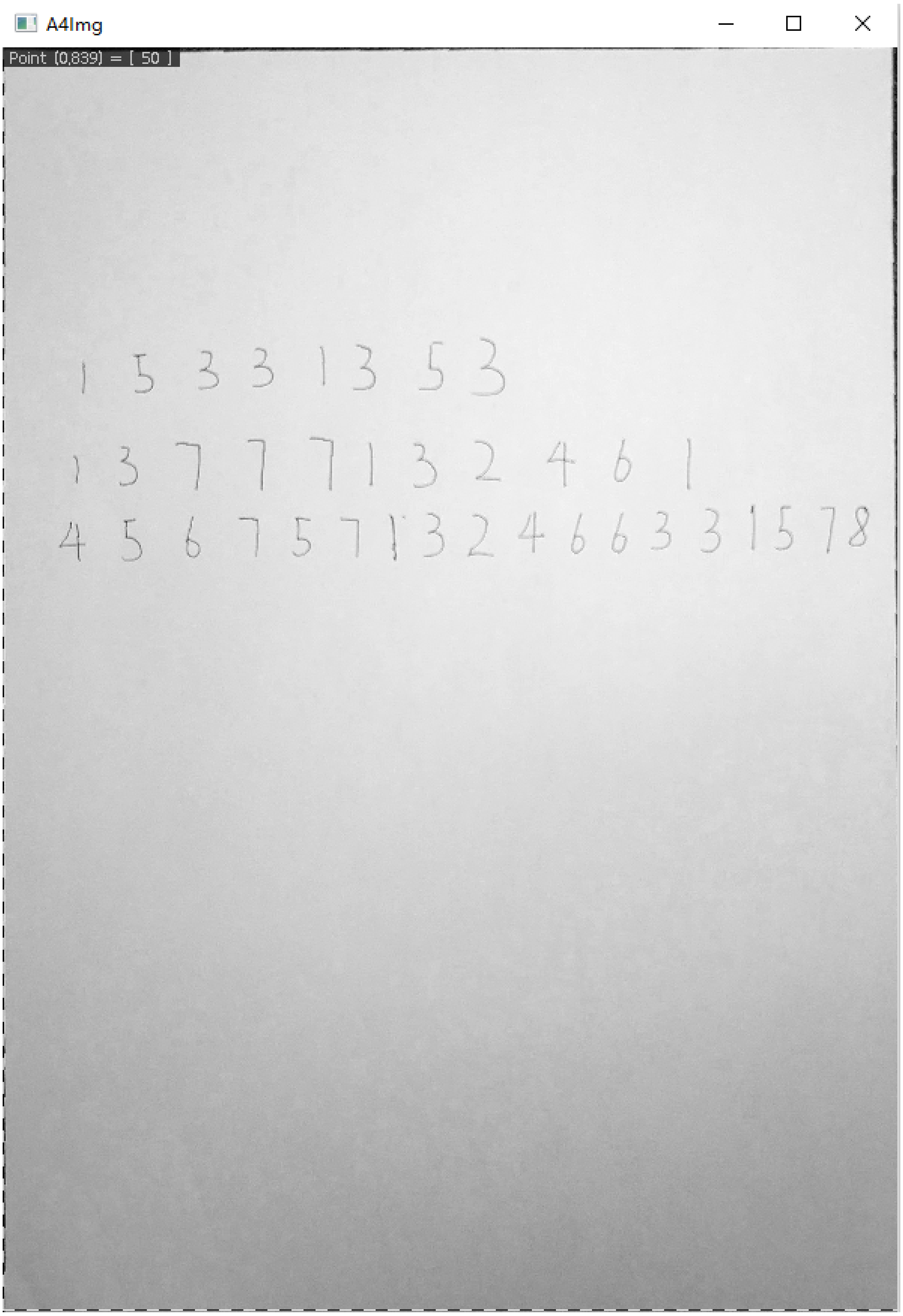
1. canny边缘检测图:



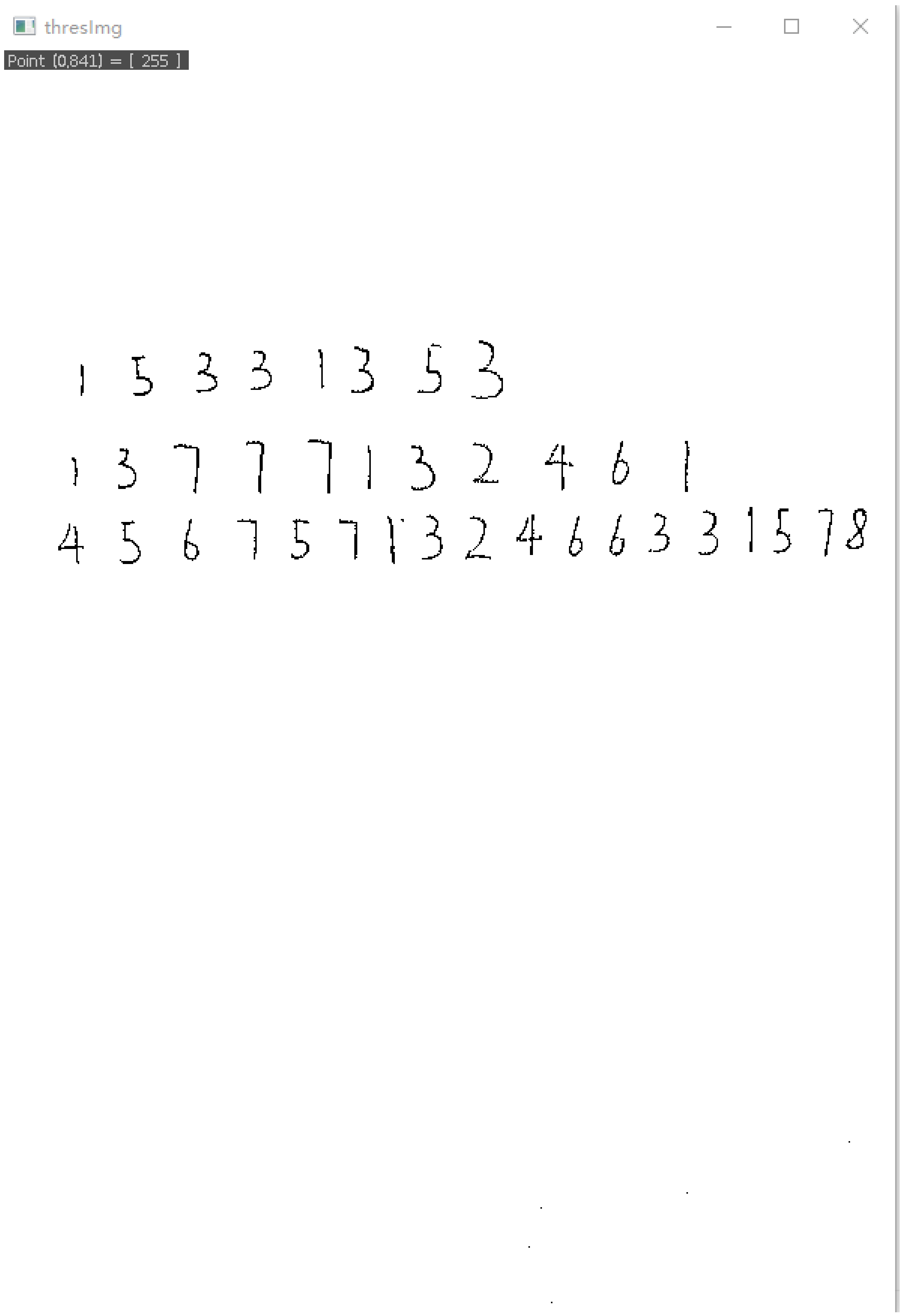
2. 霍夫变换图:


houghImg
Point (0,1993) = [11 11 11]
1 5 3 3 1 3 5 3
1 3 7 7 7 1 3 2 4 6 1
4 5 6 7 5 7 1 3 2 4 6 6 3 3 1 5 7 8

3. A4矫正图:

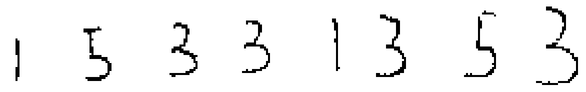

A4Img
Point (0,839) = [50]
1 5 3 3 1 3 5 3
1 3 7 7 7 1 3 2 4 6 1
4 5 6 7 5 7 1 3 2 4 6 6 3 3 1 5 7 8

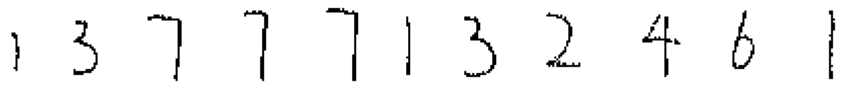
4. 阈值分割图:

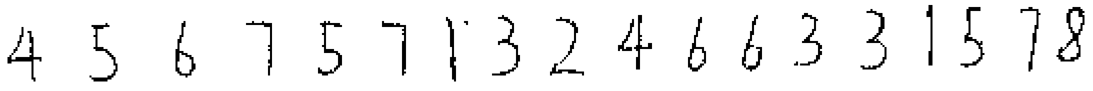

thresImg
Point (0,841) = [255]
1 5 3 3 1 3 5 3
1 3 7 7 7 1 3 2 4 6 1
4 5 6 7 5 7 1 3 2 4 6 6 3 3 1 5 7 8

5. 水平方向切割图:

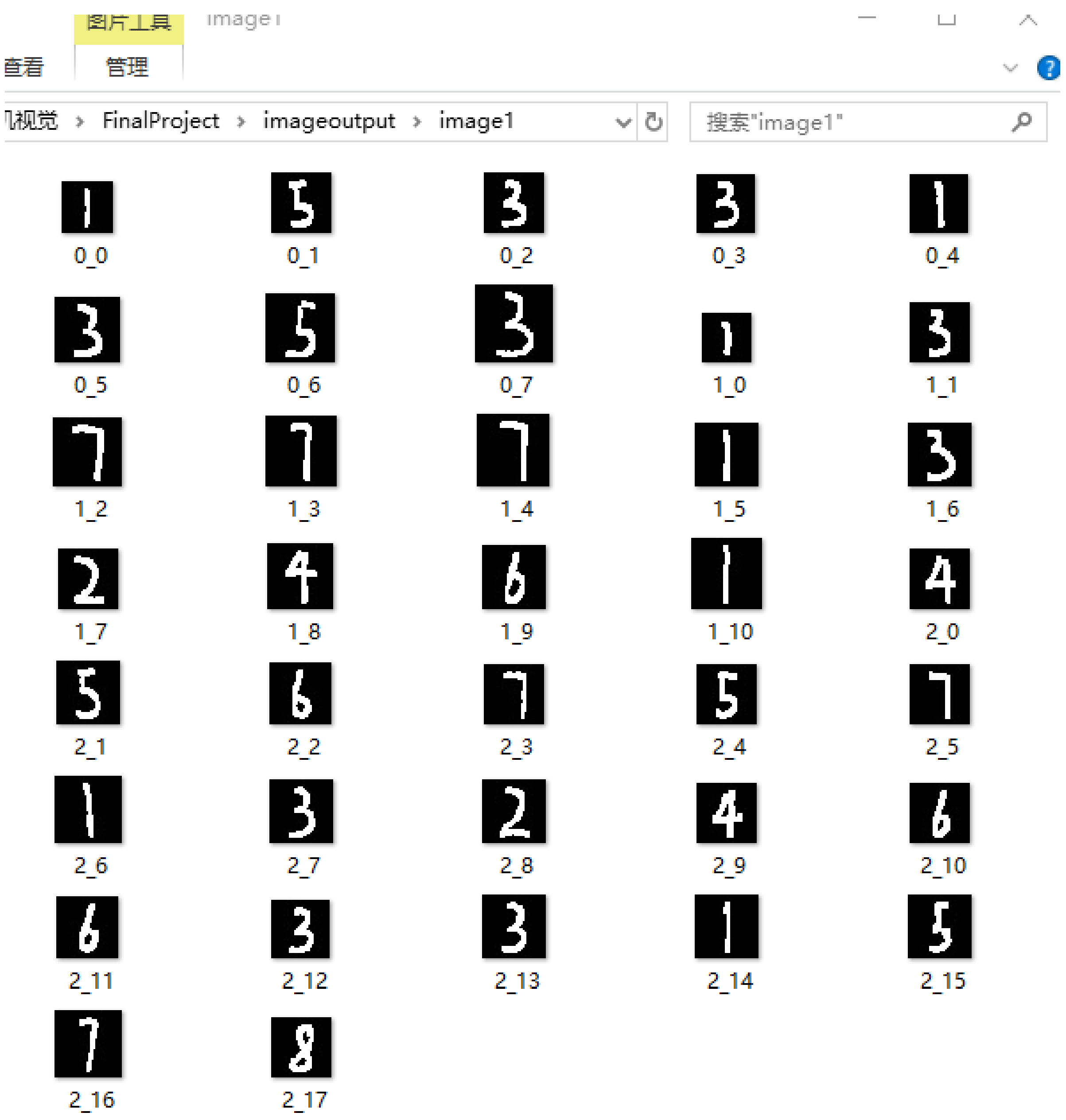
-



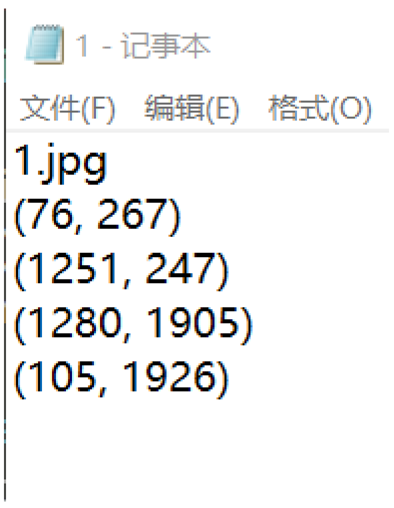
- 

- 

6. 单个数字的切割图:



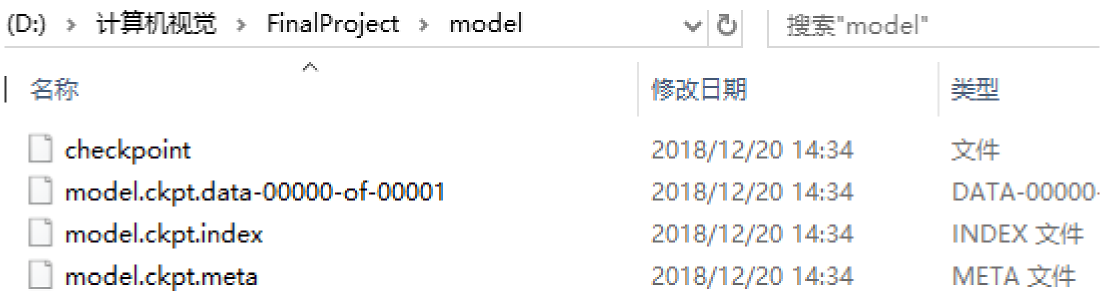
7. 记录角点的文本:



1.jpg
(76, 267)
(1251, 247)
(1280, 1905)
(105, 1926)

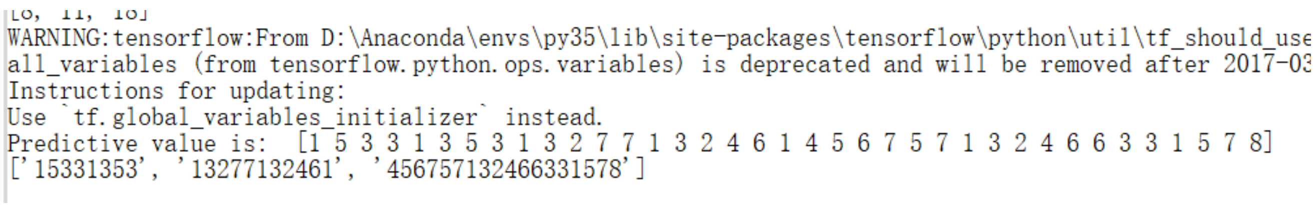
8. 训练出的model:

训练的准确度可以达到99.3%

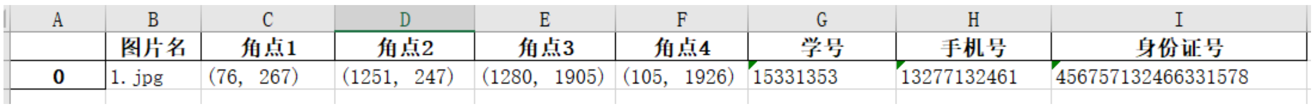

(D:) › 计算机视觉 › FinalProject › model

名称	修改日期	类型
checkpoint	2018/12/20 14:34	文件
model.ckpt.data-00000-of-00001	2018/12/20 14:34	DATA-00000
model.ckpt.index	2018/12/20 14:34	INDEX 文件
model.ckpt.meta	2018/12/20 14:34	META 文件

9. 识别结果:

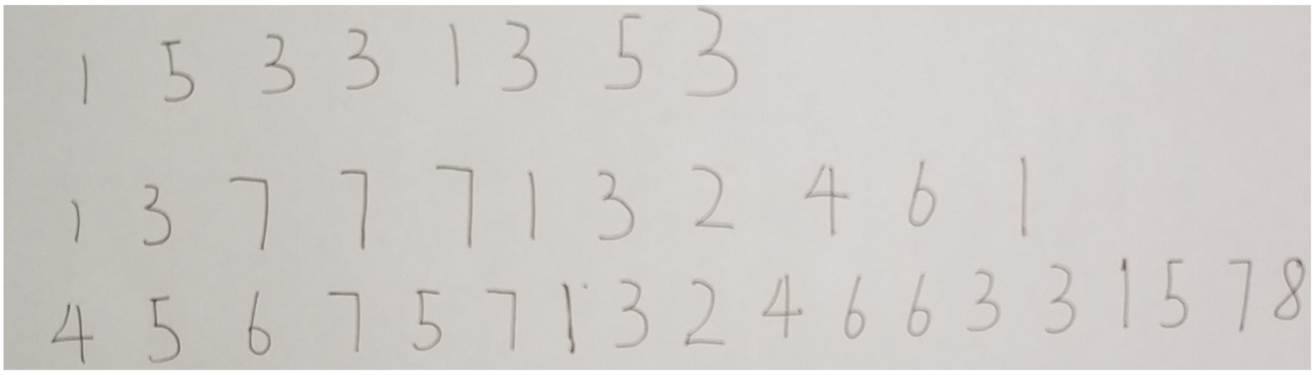

```
[8, 11, 18]
WARNING:tensorflow:From D:\Anaconda\envs\py35\lib\site-packages\tensorflow\python\util\tf_should_use
all_variables (from tensorflow.python.ops.variables) is deprecated and will be removed after 2017-03
Instructions for updating:
Use `tf.global_variables_initializer` instead.
Predictive value is:  [1 5 3 3 1 3 5 3 1 3 2 7 7 1 3 2 4 6 1 4 5 6 7 5 7 1 3 2 4 6 6 3 3 1 5 7 8]
['15331353', '13277132461', '4567571324663315 78']
```

10. 结果记入Excel:



A	B	C	D	E	F	G	H	I
	图片名	角点1	角点2	角点3	角点4	学号	手机号	身份证号
0	1.jpg	(76, 267)	(1251, 247)	(1280, 1905)	(105, 1926)	15331353	13277132461	456757132466331578

与原图相比可以看出只有手机号中的第三位识别出错，其他都正确，准确度还是很高的。



五. 实验分析

虽然说期末项目是把平时做的一些东西整合到一起，但实际做的时候就会发现其实整合到一起直接搞的话并不会有那么好的效果，其实canny，hough还有矫正部分倒还比较好办，主要是调到合适的参数同时排除掉一部分拍摄不好图片的特殊线条噪点等的影响。

主要改动的部分一部分是图像分割的部分，开始时考虑了很多办法，比如最初简单的固定值全局阈值分割，再到老师讲的OSTU大律法以及全局迭代分割或者双峰分割，效果都很不好。后来想到自己的思路太过于局限，一直都在考虑全局分割，而整幅图的光照各处不同，效果肯定不会很好，所以想到了如果局部分割效果会不会好一点。开始时想的方法也比较简单，只是类似将纸张的每一行裁成等比例的块进行局部求阈值，但这种方法缺点是如果块裁的太大那么会损失很多需要的细节，如果太小又会出现棋盘效应。之后又查了资料以及询问了大佬的意见，了解到

opencv有一个封装的自适应阈值分割算法，于是查了opencv源码，阅读之后自己重写了一下这个函数，倒不是很麻烦。然后发现效果很不错，关键点细节基本都在。

然后再就是对A4纸中的数字一个个分割出来，开始想的是将纸张先横向进行灰度统计，然后取高峰间的直线，这样可以将每一行的数字分割出来，然后再用相同的办法对于每一行竖向灰度统计然后再切割出单个数字。但实际发现这样效果对于10张图还好，但对于后面100张测试图就有点炸。于是考虑用K-means的方法进行聚类，但聚类的方法局限性比较强，要将每一行切割出并确定好每一行数字的个数并且需要合适的初始质点，不适用。于是参考了网上大佬的一个叫做区域连通标记的方法，采用二次扫描的办法，对连通区域内的数字切割出来，整体效果还不错。但缺点是有点上下有覆盖情况的图片比较难切割出来，如果结合投影法一起会好一些。

然后到了单个数字识别，开始用老师讲的分类器Adaboost以及SVM，但是效果一般。尤其是Adaboost，对于mnist的测试集，也只有百分之60多的正确率。于是参考tensorflow中文社区上CNN的讲解用CNN的方法对数字进行识别，mnist测试集的识别率最高达到了99.3%，对于自己的数字识别效果也是不错的。