手势识别: 传统机器视觉技术部分

一、项目背景

随着计算机视觉技术和智能设备的日益普及，人机交互已经成为了现代社会中不可或缺的一部分。手势识别作为一种自然、直观、无需接触的人机交互方式，逐渐成为了研究的热点。虽然目前人们对手势识别的研究热点大多集中于卷积神经网络，传统的机器视觉技术依然在工业界中有着非常广泛的应用。传统机器视觉技术主要包括图像边缘检测分析，二值图像图形分析，特征点检测等，相比于深度学习的方法，传统机器视觉技术的实现方式算法更加简洁，对算力要求更低，响应也更快。本项目从二值图像图形学的分析角度出发，首先将手部图像提取为二值图像，通过凸包的分析来实现对不同种类手势的图像进行识别。

图 1: 目标手势 A

图 2: 目标手势 C

图 3: 目标手势 $Five$

图 4: 目标手势 V

在传统机器视觉技术的分析中，凸包是指一个包含了给定点集中的所有点的最小凸多边形，凸包分析在图形学中是一种重要的算法和理论，具有广泛的应用。不同的手势有着不同的特征凸包，结合分析

凸包的缺陷，凸包的面积与周长特点等可以通过机器学习的方式实现不同手势的识别。

项目主要的开发工具是 *opencv*，开发平台为 $C++17$。*opencv* 是一个强大的计算机视觉处理库，利用 *opencv* 我们可以轻松完成对图像的读取以及处理，特征的提取等等，此外，*opencv* 自带有机器学习等科学计算库，方便了我们后续的分类任务。

二、项目基本原理

1、*Canny* 边缘检测

Canny 边缘检测是一种经典的边缘检测算法，其基本思路是通过对图像进行高斯滤波、求取梯度、非极大值抑制和双阈值处理等步骤，从而得到图像的边缘信息。具体来说，*Canny* 边缘检测算法的主要原理可以分为以下几个步骤：

首先，对图像进行高斯滤波，目的是去除图像中的高频噪声，同时保留图像中的边缘信息。高斯滤波可以在空间域或频率域中进行，对于二维图像而言，常用的高斯滤波器是二维高斯函数，可以写成如下形式：

$$g(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

其中，σ 代表高斯滤波器的标准差。

接着，求取图像的梯度，目的是检测图像中的边缘信息。常用的梯度算子包括 *Sobel* 算子、*Prewitt* 算子和 *Roberts* 算子等，其中 *Sobel* 算子是较为常用的一种，其计算方法可以表示为：

$$G_x=\begin{bmatrix}-1&0&1\\-2&0&2\\-1&0&1\end{bmatrix}*I$$

$$G_y=\begin{bmatrix}-1&-2&-1\\0&0&0\\1&2&1\end{bmatrix}*I$$

其中，$*$ 表示卷积操作，I 表示待处理的图像。将 G_x 和 G_y 计算得到的梯度图像可以进行合并，得到图像的总梯度和梯度方向，即：

$$G=\sqrt{G_x^2+G_y^2}$$

$$\Theta=\operatorname{atan2}(G_y,G_x)$$

然后，对于梯度图像中的每个像素点，进行非极大值抑制操作，目的是进一步压缩边缘信息，使得图像中的边缘线条更加锐利。具体来说，非极大值抑制可以定义为以下步骤：

- 对于梯度方向 Θ，将其量化为 4 个方向：水平、垂直和两个对角线方向；- 对于像素点 (i, j)，比较其梯度值 $G(i, j)$ 与在梯度方向上的相邻两个像素点的梯度值，如果 $G(i, j)$ 不是相邻两个像素点中最大的，则将其值设为 0。

最后，将边缘像素点进行连接，得到一条完整的边缘线，以及其对应像素的梯度强度。

图 5: 原图

图 6: $Canny$ 边缘检测结果，双阈值为 $T_{low} = 0$，$T_{High} = 100$

为了进一步提高算法的准确性，$Canny$ 边缘检测算法还引入了双阈值处理，即将梯度图像中的像素点分为强边缘和弱边缘两种类型，分别对应的梯度强度设定两个阈值 T_{high} 和 T_{low}。梯度值大于 T_{high} 的像素点被视为强边缘，梯度值介于 T_{low} 和 T_{high} 之间的像素点被视为弱边缘。

最终，$Canny$ 边缘检测算法将强边缘和与其相连的弱边缘合并成为一条完整的边缘线，同时将不属于边缘线的弱边缘剔除，从而得到了一张图像中的边缘信息。

$Canny$ 边缘检测算法具有较高的准确性和稳定性，能够有效地检测出图像中的边缘信息。但是，在应用中还需要进行一些参数的调整，以便得到更加准确的结果。例如，高斯滤波的标准差、梯度阈值等参数都需要根据具体图像的特征进行调整。此外，$Canny$ 边缘检测算法还存在一些缺陷，例如对于一些曲线边缘的检测效果较差，同时算法的计算复杂度也比较高。

为了进一步提高边缘检测的效果，一些改进的 $Canny$ 算法也得到了广泛的应用，例如基于自适应阈值的 $Canny$ 算法、基于多尺度的 $Canny$ 算法等。同时，$Canny$ 边缘检测算法也被广泛应用于图像处理中的各个领域，例如目标检测、图像分割、图像识别等，为实际应用带来了很大的价值和作用。

2、手部二值图像的提取

图像二值化是一种非常实用的图像处理操作，通过选择一个合适的阈值将图像的亮度二分化，突出不同对象的形状与轮廓。阈值的选择可以是人为选择，更多的是利用一些算法为不同的图片生成不同的阈值。目前主流的图像二值化算法包括 $OTSU$ 阈值化，迭代最优阈值化等。这些二值化方法采用的都是对图像的亮度值分布进行分析得到统计意义上的最优阈值。基于这个阈值来分割图像的前景和背景往往适用于比较简单，包含明显色差的前景和背景的图片，如果图像稍微复杂一些就会出现前后景分离错误

的结果，比如前景物体残缺等状况。

事实上，采用基于亮度值分布的图像阈值化算法所利用的图像信息非常少，这是上述这些二值化方法出现分割失误的一个重要原因。这里我们提出基于边缘检测的方法获得图像前景分离的方法。

表 1: 基于边缘检测的前后景二值化分离的方法

1、使用 $Canny$ 边缘检测提取图像边缘，边缘使用亮度值 255 的像素描述，背景用亮度值 0 的像素值描述。

2、可选：使用中值滤波除去由于噪声引起的细小错误边缘。

3、对边缘图像进行闭运算处理，尽可能使边界完整覆盖原图中的目标。

4、使用边界追踪算法，提取边缘图像中所有封闭图形的外边界轮廓，结果保存在一个点集列表中。

5、选择列表中点集最大的一项（也就是轮廓最长的一个），在 0 亮度值等尺寸背景图像画出轮廓。

图 7: 原图

图 8: 提取出来的二值图像

3、凸包分析

显然，不管是哪一种手势，其图形都是复杂的，为了提取出不同手势的特征，我们需要对不同手势的一些共有特征进行比较。最容易想到的首先是凸包缺陷的个数 N_{convex}。很明显，对于目标手势 $Five$ 来说，五个手指意味着指间的缝隙数更多，目标手势 C 和目标手势 V 要少一些，而目标手势 A 则没有指间缝隙。

凸包的概念在上文就已经提到，而凸包缺陷指的是在凸包边界内部的空白区域，也就是我们二值图像中的黑色区域，凸包缺陷的特点就是以凸包和图像的边界为自身的边界。在一般的凸包计算中，我们通常使用三个点来描述一个凸包缺陷即图像与凸包边界的两个交点和一个距离凸包边界最远的点。*opencv* 中使用的是 $Jarvis$ 算法来计算凸包，它的基本思想是将凸包上的每个点依次与当前点连线，找到与当前点连线形成的角度中最小的那个点，直到回到起始点形成闭合的凸包。

算法过程可以表示为如下的伪代码：

Input: 点集 S
Output: 点集 S 的凸包
p_{lowest} = leftmost point of S;
$p_{current} = p_{lowest}$;
$H = \{p_{lowest}\}$;
while $p_{current} \neq p_{lowest}$ **do**
    p_{next} = next point on the hull after $p_{current}$;
    foreach $p_i \in S$ **do**
        if p_i is left of the line from $p_{current}$ to p_{next}, set $p_{next} = p_i$;
    end
    $p_{current} = p_{next}$;
    add $p_{current}$ to H;
end
return H;

Algorithm 1: *Jarvis* 算法

尽管 Jarvis 算法的时间复杂度较高（$O(nh)$, 其中 h 表示凸包点数），但它的思想和实现相对简单。而凸包缺陷则可以通过使用凸包轮廓点集得到。

图 9: 凸包计算目标手势 *Five* 结果

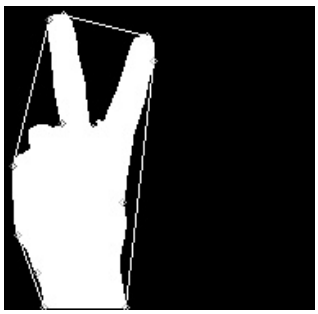

图 10: 凸包计算目标手势 V 结果

在上图中，我们可以看到手掌外围贴合的一圈细线正是我们的凸包结果，在每个凸包缺陷中，我们使用了不同颜色的小圈圈出了描述缺陷的三个点（两个交点，一个最远点）。显然在上面的两个不同手势中，凸包缺陷的数量有着非常大的区别，前者为 6，后者为 4。这是我们识别不同手势的一个重要特征。

当然如果仅以凸包缺陷个数 N_{convex} 作为唯一的判断标准是不现实的。可以看到上面两张图中，由于手部露出程度的不同，计算出来的凸包数量与理想的数量仍然可能有一定的区别，比如图 10 中手腕部分的弯曲可能会导致新的缺陷出现，图 9 中手指的张开程度比较大导致小拇指与手掌部分又形成了新的缺陷。为了改进这个问题，我们增加一个新的特征：假设两个交点为 A 和 B，最远点为 F，夹角 $\angle AFB$ 为锐角的缺陷个数 N_{acute}。这样做的原因是在自然状态下，指尖缝隙的夹角都是锐角，而手部其他地方的弯曲除非特别用力，一般得到的缺陷都是钝角。呈现锐角的缺陷夹角也是我们感兴趣的关键角。

图 11: 目标手势 $Five$ 的凸包缺陷角度状况，蓝线是关键角

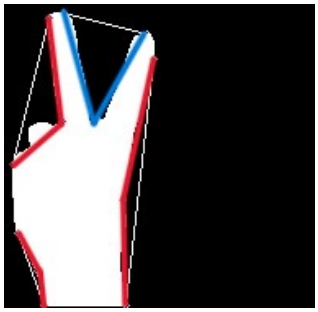

图 12: 目标手势 V 的凸包缺陷角度状况，蓝线是关键角

对于目标手势 V 和目标手势 C 来说，人类可以一眼看出两者的不同，但是对于我们到现在为止的模型来说却仍然有一定难度。这是因为在正常情况下，手势 C 和手势 V 可能都有一个特征的指间缝隙锐角和一个特征钝角（手势 V 是指尖到指关节的缺陷锐角，手势 C 是大拇指指尖到手腕的钝角）。

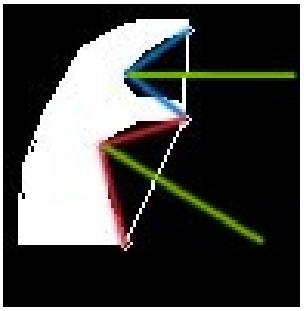

图 13: 目标手势 C 的凸包缺陷角度状况，绿线是关键角的角平分线

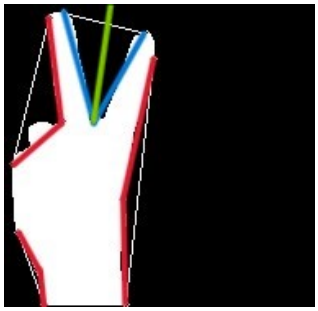

图 14: 目标手势 V 的凸包缺陷角度状况，绿线是关键角的角平分线

为了解决这个问题，我们再添加一个躺角数量 N_{lie}，这个特征表示的是角 $\angle AFB$ 对应角平分线与水平方向的夹角成锐角的数量。换句话说，人类之所以可以一眼看出图 14 和图 15 这两个二值图像分别是手势 C 和手势 V 的一个重要依据是前者的凸包缺陷是躺倒的，而后者是直立的。通过衡量绿线角平分线与水平方向的夹角就可以得到缺陷角的“倾斜程度”或者“躺平程度”。

对于最后一个目标手势 A，它并不像前三者一样可以通过计算缺陷来得到对应的特征，因为它二值图形的凸性非常好，计算的凸包几乎与图形是贴着的，存在凸包缺陷但是非常小，并且几乎没有规律可言。显然我们不能再从缺陷角的方面来考虑问题。这时我们引入三个比值：图形面积与凸包面积之比 V_{SS}、图形周长与图形面积之比 V_{CS} 以及图形周长与凸包周长之比 V_{CC}，这三者对于目标手势 A 来说，都是接近于 1 的，而对于其他的手势来说，它们的值可能各不相同。比如对于目标手势 $Five$ 来说，同等面积之下图形的周长会更长一些，对于 C 和 V 来说，这些指标可能相近。

图 15: 目标手势 A 的凸包分析

最后我们还考虑对凸包本身的特征进行分析，加入了凸包的边数 N_{edges} 作为一个分析特征。每一个样本将上文提到的所有特征抽取出来组成一个 7 维的特征向量，最后使用多分类 SVM 的方法，实现对不同种类的样本进行划分训练，得到手势识别的预测模型。

综上所述，我们最后得到的特征向量为

$$\overrightarrow{f} = [N_{acute}, N_{lie}, V_{SS}, \frac{N_{convex}}{20}, \frac{N_{edges}}{20}, V_{CS}, V_{CC}]$$

三、项目实现源码

依赖头文件以及工具函数

```
#include <iostream>
#include <fstream>
#include<vector>
#include<fstream>
#include<io.h>
#include<string>
#include<windows.h>
#include "opencv2/opencv.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <opencv2/core/hal/hal.hpp>
#include <stdio.h>
#include "opencv2/ml.hpp"

using namespace cv;
using namespace std;
double cos(Point center, Point a, Point b) {
    double ab = sqrt((a.x-b.x)*(a.x-b.x) + (a.y - b.y)*(a.y-b.y));
    double ac = sqrt((a.x - center.x)*(a.x-center.x) + (a.y -
    ↪  center.y)*(a.y-center.y));
```

```
    double bc = sqrt((b.x - center.x)*(b.x-center.x) + (b.y -
    ↪  center.y)*(b.y-center.y));
    return (ac * ac + bc * bc - ab * ab) / (2 *ac*bc);
}
vector<Point> find_rect_around(Mat canny_edge) {
    Point left(100000, 0), right(0, 0), top(0, 100000), bottom(0, 0);
    for (int i = 0; i < canny_edge.size().width; i++) {
        for (int j = 0; j < canny_edge.size().height; j++) {
        if (canny_edge.at<uchar>(j, i) == 255) {
                if (i < left.x)left = Point(i, j);
                if (i > right.x)right = Point(i, j);
                if (j > bottom.y)bottom = Point(i, j);
                if (j < top.y)top = Point(i, j);
            }
        }
    }
    Point left_top = Point(left.x, top.y);
    Point right_bottom = Point(right.x, bottom.y);
    vector<Point> Rect;
    Rect.push_back(left_top);
    Rect.push_back(right_bottom);
    return Rect;
}
```

这部分代码中，我们定义了一个利用`cv::Point`类来实现的计算夹角的 *cosine* 函数，以及一个寻找 *Canny* 检测边界最小矩形框的函数，后续需要用到这两个函数进行图像的处理以及特征的抽取。

手部图像处理函数

```
    Mat extract_main_hand(Mat origin) {
    vector<vector<Point>> contours;
    Mat canny_contours;

    Mat struct_kernel = getStructuringElement(MORPH_RECT, Size(3,3));
    Mat struct_kernel1 = getStructuringElement(MORPH_RECT, Size(2, 2));

    Canny(origin, canny_contours, 0, 100);

    dilate(canny_contours, origin, struct_kernel);
    erode(origin, origin, struct_kernel1);
    vector<Point> Rectangle = find_rect_around(origin);
```

```
    origin = origin(Rect(Rectangle[0], Rectangle[1]));
    copyMakeBorder(origin, origin, 4, 0, 4, 4, BORDER_CONSTANT, Scalar(0));
    copyMakeBorder(origin, origin, 1, 1, 1, 1, BORDER_CONSTANT, Scalar(255));
    findContours(origin, contours, RETR_TREE, CHAIN_APPROX_SIMPLE);

    Mat contours_image = Mat::zeros(origin.size(), CV_8U);
    int arg_max = 0;
    double max = 0;

    for (size_t i = 0; i < contours.size(); i++) {
        if (arcLength(contours[i],true) > max) {
            max = arcLength(contours[i],true);
            arg_max = i;
        }
    }

    drawContours(contours_image, contours, arg_max, 255);
    drawContours(contours_image, contours, arg_max, 255, cv::FILLED);
    contours_image = contours_image(Rect(Point(1, 1),
    ↪  Point(contours_image.size().width - 1, contours_image.size().height - 1)));
    for (int i = 0; i < contours_image.cols; i++)
        for (int k = 0; k < contours_image.rows; k++) {
            if (contours_image.at<uchar>(k, i) == 0)
                contours_image.at<uchar>(k, i) = 255;
            else
                contours_image.at<uchar>(k, i) = 0;
        }

    return contours_image;
}

vector<vector<Point>> prepare(Mat origin) //return the contours of hand
{
    double contrast = 3;

    if (origin.type() != CV_8U) {
        cvtColor(origin, origin, COLOR_RGB2GRAY);
        vector<vector<Point>> color_contours;

    }
```

```
    Mat g = getGaussianKernel(3, 0.75);
    Mat gaussian = g * g.t();
    filter2D(origin, origin, CV_32F, gaussian);
    origin.convertTo(origin, CV_8U);

    origin = extract_main_hand(origin);
    vector<vector<Point>> Contours;
    findContours(origin, Contours, RETR_TREE, CHAIN_APPROX_SIMPLE);
    int max = 0, argmax = 0;
    for (int k = 0; k < Contours.size(); k++) {
        if (Contours[k].size() > max) {
            max = Contours[k].size();
            argmax = k;
        }
    }

    vector<vector<Point>> r;
    vector<Point> size_point;
    size_point.push_back(Point(origin.size().width, origin.size().height));
    r.push_back(Contours[argmax]);
    r.push_back(size_point);
    return r;
}

```

函数prepare()将原始输入的手势图像进行预处理操作，包括高斯模糊，*Canny* 边缘提取，手势二值图像的抽取。抽取手势二值图像调用extract_main_hand()函数来完成。函数extract_main_hand()接受一个高斯模糊之后的原始图像，然后进行 *Canny* 边缘提取，将边缘提取到一个背景亮度为 0 的新图像中，然后进行闭运算，边界追踪，获得轮廓点集列表，找到最长轮廓之后以cv::FILLED方式将原来的轮廓全部填充，得到了手部二值图像contours_image并返回到prepare() 函数中，继续进行一次边界追踪之后将追踪的点集返回。

特征抽取函数

```
vector<double> extract_feature(vector<vector< Point >>Contours, Size origin_size) {
    vector<double> features;
    Size image_size(Contours[1][0].x, Contours[1][0].y);
    Mat Contours_image(origin_size, CV_8U);
    threshold(Contours_image, Contours_image, 255, 255, THRESH_BINARY);
    drawContours(Contours_image, Contours, 0, Scalar(255), cv::FILLED);
    vector<vector<Point>> hulls(1);
```

```
    vector<int> hullsi;
    vector<Vec4i> defects;
    convexHull(Contours[0], hulls[0]);
    convexHull(Contours[0], hullsi);
    try {
        convexityDefects(Contours[0], hullsi, defects);
    }
    catch(cv::Exception) {
        Mat struct_kernel1 = getStructuringElement(MORPH_RECT, Size(2, 2));
        erode(Contours_image, Contours_image, struct_kernel1);
        Contours_image = extract_main_hand(Contours_image);
        image_size = Contours_image.size();
        findContours(Contours_image, Contours, RETR_TREE, CHAIN_APPROX_SIMPLE);
        convexHull(Contours[0], hulls[0]);
        convexHull(Contours[0], hullsi);
        convexityDefects(Contours[0], hullsi, defects);
    }
    drawContours(Contours_image, hulls, 0, Scalar(255), 1);

    int acute_num=0;
    int lie_angle = 0;
    for (int k = 0; k < defects.size(); k++) {
        Point start = Contours[0][defects[k][0]];
        Point end = Contours[0][defects[k][1]];
        Point Far = Contours[0][defects[k][2]];

        vector<Point> defect_triangle{ start,end,Far };
        double defect_area = contourArea(defect_triangle);
        if (defect_area < 25)continue;

        circle(Contours_image, start, 2, Scalar(50+20*k));
        circle(Contours_image, end, 2, Scalar(51+20*k));
        circle(Contours_image, Far, 2, Scalar(52+20*k));

        double cosine = cos(Far, start, end);
        if (cosine > 0) {
            acute_num++;

            Point mid((start.x + end.x) / 2, (start.y + end.y) / 2);
            Vec<double, 2> angle(mid.x - Far.x, mid.y - Far.y);
```

```
            Vec<double, 2> horizon(1, 0);
            double lie = angle.dot(horizon) /
            ↪  sqrt((angle.val[0]*angle.val[0]+angle.val[1]*angle.val[1]) *
            ↪  (horizon.val[0]*horizon.val[0]+horizon.val[1]*horizon.val[1]));
            if (lie > 0.5 || lie < -0.5)lie_angle ++ ;
        }

    }
    features.push_back(acute_num);
    features.push_back(lie_angle);

    double contour_area = contourArea(Contours[0]);
    double hull_area = contourArea(hulls[0]);
    double contour_versus_hull = contour_area / hull_area;
    features.push_back(contour_versus_hull);

    features.push_back(double(defects.size()) /20);
    features.push_back(double(hullsi.size()) / 20);

    double circumference = arcLength(Contours[0], true);
    double hull_circumference = arcLength(hulls[0], true);
    double circumference_versus_area = circumference / contour_area;
    double circumference_versus_hull_circumference = circumference /
    ↪  hull_circumference;
    features.push_back(circumference_versus_area);
    features.push_back(circumference_versus_hull_circumference);
    return features;
}
```

函数extract_features()接受一个二值化手部图像，按照上文所述抽取相应的特征到features向量中，并将其返回。

主函数、文件读取函数与全局变量部分

```
vector<double> tag;
vector<vector<double>> train_data;

void read_data(string path,int t) {
    for (int k = 1; k < 64; k++) {
        string seq = to_string(k);
        Mat m;
```

```
        if (k <= 50)m = imread(path + seq + ".png");
        else if (k > 50) {
            continue;
            m = imread(path + seq + ".jpg");
        }
        m = m(Rect(Point(2, 2), Point(m.size().width - 2, m.size().height)));
        vector<vector<Point>> Contours = prepare(m);
        vector<double> features = extract_feature(Contours, Size(m.size().width,
        ↪  m.size().height));
        train_data.push_back(features);
        tag.push_back(t);
    }
}

std::vector<std::string> getFilesInFolder(const char* folderPath) {
    std::vector<std::string> fileList;
    WIN32_FIND_DATAA findData;
    HANDLE hFind = FindFirstFileA(folderPath, &findData);

    if (hFind == INVALID_HANDLE_VALUE) {
        std::cerr << "Error finding directory:" << folderPath << std::endl;
        return fileList;
    }
    do {
        if (findData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
            continue;
        }

        fileList.push_back(findData.cFileName);
    } while (FindNextFileA(hFind, &findData));

    FindClose(hFind);

    return fileList;
}

int main()
{
    string path = "../Hand_Posture_Easy_Stu/C/C";
    read_data(path, 0);
```

```
47      path = "../Hand_Posture_Easy_Stu/A/A";
48      read_data(path, 1);
49      path = "../Hand_Posture_Easy_Stu/V/V";
50      read_data(path, 2);
51      path = "../Hand_Posture_Easy_Stu/C/C";
52      read_data(path, 3);
53
54      Ptr<ml::SVM> svm_classifier = ml::SVM::create();
55      svm_classifier->setType(ml::SVM::C_SVC);
56      svm_classifier->setKernel(ml::SVM::RBF);
57      svm_classifier->setGamma(0.001);
58      svm_classifier->setC(20000);
59
60      Mat train_mat(train_data.size(), 7, CV_32F);
61      Mat train_tag(tag.size(), 1, CV_32S);
62      for (int i = 0; i < train_data.size(); i++) {
63          for (int j = 0; j < 7; j++) {
64              train_mat.at<float>(i, j) = train_data[i][j];
65          }
66          train_tag.at<int>(i, 0) = tag[i];
67      }
68
69      Ptr<ml::TrainData> train = ml::TrainData::create(train_mat, ml::ROW_SAMPLE,
        ↪  train_tag);
70      svm_classifier->train(train);
71
72      vector<string> filename;
73      filename=getFilesInFolder("..\\Hand_Posture_Easy_Stu\\validate\\*.*");
74      for (int k = 0; k < filename.size(); k++) {
75          Mat test = imread("../Hand_Posture_Easy_Stu/validate/"+filename[k]);
76          vector<vector<Point>> Contours = prepare(test);
77          vector<double> features = extract_feature(Contours, Size(test.size().width,
            ↪  test.size().height));
78          Mat features_mat(1, 7, CV_32F);
79          for (int k = 0; k < 7; k++) { features_mat.at<float>(0, k) = features[k]; }
80          int res = svm_classifier->predict(features_mat);
81          cout <<filename[k]<<' ' << res << endl;
82      }
83
84  }
```

在主函数中，我们利用 *opencv* 中的`ml`库，定义 SVM 训练器`svm_classifier`，并基于此将读取的文件数据进行训练。最终进行预测并输出。

四、验证结果

由于给定的数据集较少，而且由于数据的来源比较复杂，有些阴影比较重的图像可能会出现手部二值图像分离错误，导致最后的训练数据存在异常噪声，我们将全部的给定数据用于 SVM 的数据训练，验证图像来源于作者自己给自己手部拍摄的照片。这里我们同时给出验证的图像和模型的预测分类。

最终的报告准确率为 84%，实际调整参数的过程中，发现模型对 $A67$，$A70$ 和 $A73$ 的识别效果并不好，推测可能是由于光照原因，墙壁上产生非常明显的光斑被 $Canny$ 边缘检测器错误提取出来，导致最后凸包的分析结果与大多数目标手势 A 的样本有非常大的变异。

从最终的结果也可以看出来，模型对于目标手势 $Five$ 和目标手势 C 有着非常好的识别效果，对于目标手势 V 的识别效果稍差，这可能是因为验证图像在手势的旋转不变性也提出了一定的要求，这个时候除非能够使用纹路提取与辨识，否则很难将 V 的指间夹角和 C 的指间夹角区分开来。

A65.jpg
Pred:A

A66.jpg
Pred:A

A67.jpg
Pred:V

A68.jpg
Pred:A

A69.jpg
Pred:Five

A70.jpg
Pred:V

A71.jpg
Pred:A

A72.jpg
Pred:A

A73.jpg
Pred:V

A74.jpg
Pred:A

A75.jpg
Pred:A

C65.jpg
Pred:C

C66.jpg
Pred:C

C67.jpg
Pred:C

C68.jpg
Pred:C

C69.jpg
Pred:C

C70.jpg
Pred:C

C71.jpg
Pred:C

C72.jpg
Pred:C

C73.jpg
Pred:C

C74.jpg
Pred:C

C75.jpg
Pred:C

Five65.jpg
Pred:Five

Five66.jpg
Pred:Five

Five67.jpg
Pred:Five

Five68.jpg
Pred:Five

Five69.jpg
Pred:Five

Five70.jpg
Pred:Five

Five71.jpg
Pred:Five

Five72.jpg
Pred:V

Five73.jpg
Pred:Five

Five74.jpg
Pred:Five

Five75.jpg
Pred:Five

V65.jpg
Pred:C

V66.jpg
Pred:V

V67.jpg
Pred:V

V68.jpg
Pred:C

V69.jpg
Pred:V

V70.jpg
Pred:V

V71.jpg
Pred:V

V72.jpg
Pred:V

V73.jpg
Pred:V

V74.jpg
Pred:V

V75.jpg
Pred:V

Total number:44
Correct number:37
Accuracy:84%

五、项目总结

本项目从传统机器视觉技术出发，利用二值图像图形学分析的原理，提出了通过分析七维特征向量来分类的思路，有效实现了对四种目标手势的识别任务。准确率尚可。基于二值图像的检测算法最大的好处就是算法简单，并且可以随着后续的需要，向特征向量中添加新的维度，以实现对图像更多特征进行一个匹配与计算，具有极佳的可拓展性。另一方面，使用二值图像分析也有一些弱点比如检测结果可能受到物体光影的干扰，并且由于 $Canny$ 边缘提取器中的双阈值参数固定，很难保证检测器对一些复杂图片能否具有合适的边缘提取敏感度。