CSE578 Computer Vision

Assignment 4 : GrabCut

Karnik Ram
2018701007

The procedure for implementing the GrabCut algorithm for foreground extraction, along with the code is explained in this report. The influence of the hyperparameters on the obtained results is then discussed. The generated results are provided at the end.

The code files are provided in the `code` directory along with this submission. The code is written in C++ and is written as a class (`GrabCut`) for modularity. To run the program,

```
cd code
mkdir build && cd build
cmake ..
make
./run <image-path> <ouput-path>
# eg: ./run ../data/images/person2.png ../data/results/person2.png
```

1 Procedure with Code

Each step in the procedure is explained briefly in the following subsections, along with the corresponding code.

1.1 User interaction

The user is required to draw a bounding box around the object of interest, thereby labelling some background pixels. This is implemented using OpenCV's `highgui` module. Once the bounding box is set, the grabcut algorithm begins. The code is as follows:

```
void GrabCut::captureMouseClick(int event, int x, int y, int flags, void *userdata)
{
    switch(event)
    {
        case CV_EVENT_LBUTTONDOWN:
            {
                if(roi_state == NOT_SET)
                {
                    roi_state = IN_PROCESS;
                    roi = cv::Rect(x,y,1,1);
                }
            }
            break;

        case CV_EVENT_LBUTTONUP:
            {
                if(roi_state == IN_PROCESS)
                {
                    roi = cv::Rect(cv::Point(roi.x,roi.y),cv::Point(x,y));
                    roi_state = SET;
                    showInputImage();
                    std::cout << "Region set!\nPress 's' to start GrabCut!\n";
                }
            }
            break;

        case CV_EVENT_MOUSEMOVE:
            {
                if(roi_state == IN_PROCESS)
                {
                    roi = cv::Rect(cv::Point(roi.x,roi.y),cv::Point(x,y));
                    showInputImage();
                }
            }
            break;
    }
}

```

1.2 Initializing Gaussian mixture models

First an alpha matte matrix is initialized with the background pixels hard labelled as 0, and the unknown pixels (those within the bounding box) soft labelled as foreground or 3. 5-component GMMs are then

fitted for the background and foreground regions each using Kmeans. The following is the corresponding code:

```
void GrabCut::initializeGmm()
{
    // Assign initial opacity values based on selected background region
    alpha_matte.create(input_img.size(), CV_8UC1);
    alpha_matte.setTo(BG);
    alpha_matte(roi).setTo(PFG);

    // Initialize fg and bg GMM from initial selection using kmeans
    fg_gmm.models.resize(num_components);
    bg_gmm.models.resize(num_components);

    std::vector<cv::Vec3f> fg_samples, bg_samples;

    for(int x = 0; x < input_img.rows; x++)
        for(int y = 0; y < input_img.cols; y++)
        {
            if(alpha_matte.at<uchar>(x, y) == FG || alpha_matte.at<uchar>(x, y) == PFG)
                fg_samples.push_back((cv::Vec3f)input_img.at<cv::Vec3b>(x, y));

            else
                bg_samples.push_back((cv::Vec3f)input_img.at<cv::Vec3b>(x, y));
        }

    std::vector<int> fg_cluster_indices, bg_cluster_indices;
    cv::kmeans(fg_samples, num_components, fg_cluster_indices, cv::TermCriteria(
    CV_TERMCRIT_ITER, 10, 0.0), 0, cv::KMEANS_PP_CENTERS);
    cv::kmeans(bg_samples, num_components, bg_cluster_indices, cv::TermCriteria(
    CV_TERMCRIT_ITER, 10, 0.0), 0, cv::KMEANS_PP_CENTERS);

    int i = 0, j = 0;
    pixel_to_component.create(input_img.size(), CV_8UC1);
    for(int x = 0; x < pixel_to_component.rows; x++)
        for(int y = 0; y < pixel_to_component.cols; y++)
        {
            if(alpha_matte.at<uchar>(x, y) == FG || alpha_matte.at<uchar>(x, y) == PFG)
                pixel_to_component.at<uchar>(x, y) = fg_cluster_indices[i++];
            else
                pixel_to_component.at<uchar>(x, y) = bg_cluster_indices[j++];
        }

    learnGmmParameters();
}

```

The model parameters - mean, covariance, component weights - are then estimated using standard MLE expressions. The inverse covariance and determinant of covariance matrix are also stored for convenience. The code is as follow:

```
 void GrabCut::learnGmmParameters()
{
    // Estimate mean, convariance, component weights
    int component;
    cv::Vec3b pixel;
    Eigen::Vector3f eg_pixel;

    for(int x = 0; x < pixel_to_component.rows; x++)
        for(int y = 0; y < pixel_to_component.cols; y++)
        {
            component = pixel_to_component.at<uchar>(x, y);
            pixel = input_img.at<cv::Vec3b>(x, y);
            eg_pixel << pixel[0], pixel[1], pixel[2];

            if(alpha_matte.at<uchar>(x, y) == FG || alpha_matte.at<uchar>(x, y) == PFG)
            {
                fg_gmm.models[component].mean += eg_pixel;
                fg_gmm.models[component].covariance += eg_pixel * eg_pixel.transpose();
                fg_gmm.models[component].sample_count++;
                fg_gmm.sample_count++;
            }

            else
            {
                bg_gmm.models[component].mean += eg_pixel;
                bg_gmm.models[component].covariance += eg_pixel * eg_pixel.transpose();
                bg_gmm.models[component].sample_count++;
                bg_gmm.sample_count++;
            }
        }
```

```
31
32      for(int i = 0; i < num_components; i++)
33      {
34          if(fg_gmm.models[i].sample_count == 0)
35              fg_gmm.models[i].weight = 0;
36
37          else
38          {
39              fg_gmm.models[i].mean /= fg_gmm.models[i].sample_count;
40              fg_gmm.models[i].covariance /= fg_gmm.models[i].sample_count;
41              fg_gmm.models[i].covariance -= fg_gmm.models[i].mean * fg_gmm.models[i].mean
    .transpose();
42              fg_gmm.models[i].inverse_covariance = fg_gmm.models[i].covariance.inverse();
43              fg_gmm.models[i].detm_covariance = fg_gmm.models[i].covariance.determinant()
    ;
44              fg_gmm.models[i].weight = (double)fg_gmm.models[i].sample_count / fg_gmm.
    sample_count;
45          }
46
47          if(bg_gmm.models[i].sample_count == 0)
48              bg_gmm.models[i].weight = 0;
49
50          else
51          {
52              bg_gmm.models[i].mean /= bg_gmm.models[i].sample_count;
53              bg_gmm.models[i].covariance /= bg_gmm.models[i].sample_count;
54              bg_gmm.models[i].covariance -= bg_gmm.models[i].mean * bg_gmm.models[i].mean
    .transpose();
55              bg_gmm.models[i].inverse_covariance = bg_gmm.models[i].covariance.inverse();
56              bg_gmm.models[i].detm_covariance = bg_gmm.models[i].covariance.determinant()
    ;
57              bg_gmm.models[i].weight = (double)bg_gmm.models[i].sample_count / bg_gmm.
    sample_count;
58          }
59      }
60  }
61
```

1.3 Constructing a graph and solving

The unary and binary cost terms in the Gibb's segmentation energy function segmentation are then computed. The function is then minimized using min-cut. GrabCut co-author Vladimir's C-library implementation for max-flow/min-cut is made use of for this purpose. The unary cost terms correspond to the terminal edges, and the binary cost terms correspond to the pixel-pixel edges. The code is as follows:

```
1
2   void GrabCut::estimateNeighborWeights(const double &beta, const double &gamma, cv::Mat &
    left_weights,
3           cv::Mat &up_weights, cv::Mat &up_left_weights, cv::Mat &up_right_weights)
4   {
5       cv::Vec3b pixel, diff;
6
7       for(int x = 0; x < input_img.rows; x++)
8           for(int y = 0; y < input_img.cols; y++)
9           {
10              pixel = input_img.at<cv::Vec3b>(x, y);
11              if(x > 0)
12              {
13                  diff = pixel - input_img.at<cv::Vec3b>(x - 1, y);
14                  left_weights.at<double>(x, y) = gamma * exp(-beta*diff.dot(diff));
15              }
16
17              if(y > 0)
18              {
19                  diff = pixel - input_img.at<cv::Vec3b>(x, y - 1);
20                  up_weights.at<double>(x, y) = gamma * exp(-beta*diff.dot(diff));
21              }
22
23              if(x>0 && y>0)
24              {
25                  diff = pixel - input_img.at<cv::Vec3b>(x - 1, y - 1);
26                  up_left_weights.at<double>(x, y) = (gamma/sqrt(2)) * exp(-beta*diff.dot(
    diff));
27              }
28
29              if(x>0 && y<input_img.cols-1)
30              {
```

```
                diff = pixel - input_img.at<cv::Vec3b>(x + 1, y - 1);
                up_right_weights.at<double>(x, y) = (gamma/sqrt(2)) * exp(-beta*diff.dot
    (diff));
            }
        }
}

void GrabCut::constructGraph(const double &lambda, const cv::Mat &left_weights,
        const cv::Mat &up_weights, const cv::Mat &up_left_weights,
        const cv::Mat &up_right_weights, GraphType *graph)
{
    cv::Vec3b pixel;
    Eigen::Vector3f eg_pixel;
    int node_id;
    Gaussian model;
    double source_weight, sink_weight, n_weight;

    for(int x = 0; x < input_img.rows; x++)
        for(int y = 0; y < input_img.cols; y++)
        {
            pixel = input_img.at<cv::Vec3b>(x,y);
            eg_pixel << pixel[0], pixel[1], pixel[2];
            node_id = graph->add_node();
            source_weight = sink_weight = n_weight = 0;

            if(alpha_matte.at<uchar>(x, y) == FG)
            {
                source_weight = lambda;
                sink_weight = 0;
            }

            else if(alpha_matte.at<uchar>(x, y) == BG)
            {
                source_weight = 0;
                sink_weight = lambda;
            }

            else if(alpha_matte.at<uchar>(x, y) == PFG || alpha_matte.at<uchar>(x,y) ==
    PBG)
            {
                for(int i = 0; i < num_components; i++)
                {
                    model = fg_gmm.models[i];
                    if(model.weight > 0)
                    {
                        sink_weight += model.weight * 1.0f/sqrt(model.detm_covariance)
                            * exp(-0.5f*(eg_pixel - model.mean).transpose() * model.
    inverse_covariance * (eg_pixel - model.mean));
                    }

                    model = bg_gmm.models[i];
                    if(model.weight > 0)
                    {
                        source_weight += model.weight * 1.0f/sqrt(model.detm_covariance)
                            * exp(-0.5f*(eg_pixel - model.mean).transpose() * model.
    inverse_covariance * (eg_pixel - model.mean));
                    }
                }

                source_weight = -log(source_weight);
                sink_weight = -log(sink_weight);
            }

            graph->add_tweights(node_id, source_weight, sink_weight);

            if(x > 0)
            {
                n_weight = left_weights.at<double>(x, y);
                graph->add_edge(node_id, node_id - input_img.cols, n_weight, n_weight);
            }

            if(y > 0)
            {
                n_weight = up_weights.at<double>(x, y);
                graph->add_edge(node_id, node_id - 1, n_weight, n_weight);
            }

            if(x>0 && y>0)
            {
                n_weight = up_left_weights.at<double>(x, y);
```

```
                graph->add_edge(node_id, node_id - input_img.cols - 1, n_weight,
    n_weight);
            }

            if(x>0 && y<input_img.cols-1)
            {
                n_weight = up_right_weights.at<double>(x, y);
                graph->add_edge(node_id, node_id -input_img.cols + 1, n_weight, n_weight
    );
            }
        }
}

void GrabCut::estimateSegmentation(GraphType *graph)
{
    int flow = graph->maxflow();

    for(int x = 0; x < alpha_matte.rows; x++)
        for(int y = 0; y < alpha_matte.cols; y++)
        {
            if(alpha_matte.at<uchar>(x, y) == PBG || alpha_matte.at<uchar>(x, y) == PFG)
            {
                if(graph->what_segment(x*alpha_matte.cols + y) == GraphType::SOURCE)
                    alpha_matte.at<uchar>(x, y) = PFG;

                else
                    alpha_matte.at<uchar>(x, y) = PBG;
            }
        }

    //extract foreground image
    foreground = cv::Mat::zeros(input_img.size(), CV_8UC3);

    for(int i = 0; i < input_img.rows; i++)
        for(int j = 0; j < input_img.cols; j++)
            if(alpha_matte.at<uchar>(i, j) == PFG || alpha_matte.at<uchar>(i, j) == FG)
                foreground.at<cv::Vec3b>(i, j) = input_img.at<cv::Vec3b>(i, j);
}

```

1.4 Re-estimation and learning

Using the estimated segmentation, the alpha matte values are re-assigned according to the estimated foregound and background regions and then the GMMs are remodelled. The most likely component is assigned to each fg/bg pixel and the model parameters are re-estimated. The code is as follows:

```
void GrabCut::assignGmmComponents()
{
    cv::Vec3b pixel;
    Eigen::Vector3f eg_pixel;
    Gaussian model;

    double p, p_max, best_component;

    for(int x = 0; x < input_img.rows; x++)
        for(int y = 0; y < input_img.cols; y++)
        {
            pixel = input_img.at<cv::Vec3b>(x, y);
            eg_pixel << pixel[0], pixel[1], pixel[2];

            if(alpha_matte.at<uchar>(x, y) == FG || alpha_matte.at<uchar>(x, y) == PFG)
            {
                p_max = 0;

                for(int i = 0; i < num_components; i++)
                {
                    model = fg_gmm.models[i];
                    if(model.weight > 0)
                    {
                        p = 1.0f/sqrt(model.detm_covariance)
                            * exp(-0.5f*(eg_pixel - model.mean).transpose() * model.
    inverse_covariance
                                    * (eg_pixel - model.mean));
                    }

                    if(p > p_max)
                    {
                        p_max = p;
```

```
                        best_component = i;
                    }
                }

                pixel_to_component.at<uchar>(x, y) = best_component;
            }

            else
            {
                p_max = 0;

                for(int i = 0; i < num_components; i++)
                {
                    model = bg_gmm.models[i];
                    if(model.weight > 0)
                    {
                        p = 1.0f/sqrt(model.detm_covariance)
                            * exp(-0.5f*(eg_pixel - model.mean).transpose() * model.
    inverse_covariance
                                    * (eg_pixel - model.mean));
                    }

                    if(p > p_max)
                    {
                        p_max = p;
                        best_component = i;
                    }
                }

                pixel_to_component.at<uchar>(x, y) = best_component;
            }
        }
}


```

1.5 Iterative process

This process is repeated iteratively within a loop, with the user choosing whether to continue for another iteration or to stop, after each iteration. The code is as follows:

```
void GrabCut::runGrabCut()
{
    int num_nodes = input_img.cols * input_img.rows;
    int num_edges = 4*input_img.cols*input_img.rows - 3*input_img.cols -3*input_img.rows
     + 2;
    //int num_edges = 2*input_img.cols*input_img.rows - input_img.cols - input_img.rows;
    double beta = estimateBeta();
    double gamma = 20;
    double lambda = 200;

    cv::Mat left_weights = cv::Mat::zeros(input_img.size(), CV_64FC1);
    cv::Mat up_weights = cv::Mat::zeros(input_img.size(), CV_64FC1);
    cv::Mat up_left_weights = cv::Mat::zeros(input_img.size(), CV_64FC1);
    cv::Mat up_right_weights = cv::Mat::zeros(input_img.size(), CV_64FC1);

    int iter = 1;
    std::cout << "Running iteration #" << iter++ << std::endl;

    estimateNeighborWeights(beta, gamma, left_weights, up_weights, up_left_weights,
    up_right_weights);

    initializeGmm();
    std::cout << "Initialization complete!\n";

    cv::namedWindow("Foreground", CV_WINDOW_NORMAL);

    while(1)
    {

        GraphType *graph = new GraphType(num_nodes, num_edges);
        constructGraph(lambda, left_weights, up_weights, up_left_weights,
    up_right_weights, graph);
        estimateSegmentation(graph);
        std::cout << "Segmentation complete!\n";

        cv::imshow("Foreground", foreground);
        std::cout << "\nPress 'c' for another iteration\nPress 'Esc' to save output and
    exit\n\n";
```

```
35
36          while(1)
37          {
38              char c = cv::waitKey(0);
39
40              if(c == 'c')
41              {
42                  std::cout << "Running iteration #" << iter++ << std::endl;
43                  break;
44              }
45
46              else if(c == '\x1b')
47              {
48                  std::cout << "Saving output..\n";
49                  cv::imwrite(output_path, foreground);
50                  boost::filesystem::path p(output_path);
51                  std::string file_name = p.filename().string();
52                  cv::Mat temp_img;
53                  input_img.copyTo(temp_img);
54                  cv::rectangle(temp_img,cv::Point(roi.x,roi.y),cv::Point(roi.x + roi.
    width,
55                      roi.y + roi.height),cv::Scalar(0,0,255),2);
56                  cv::imwrite("../data/results/input/" + file_name, temp_img);
57
58                  return;
59              }
60          }
61
62          assignGmmComponents();
63          learnGmmParameters();
64      }
65  }
66
```

2 Hyperparameters

2.1 Number of iterations

Images in which the foregound object is only a small fraction of the bounding box region required more number of iterations. Images with a large foreground, like the `flower` example gave a good result in one step. However, the segmentation doesn't converge to a fixed segmentation - more likely to be a bug in my implementation.

2.2 Number of components

Performance decreased with more number of components, especially for images with fewer colors. In a sense, more components causes multiple components to be assigned to the same object which leads to harder segmentation. Number of components was set to 5 for generating results.

2.3 Number of neighbors

8-neighborhood gave better results than 4-neighborhood and also required fewer iterations. 8-neighborhood was used for generating results.

2.4 Color space

Though in theory, color spaces like HSV and LAB can lead to better segmentation results, I did not get any better results in my experiments. YCrCb space gave worse results.

3 Results

All the input and result images show here can also be found here - link.

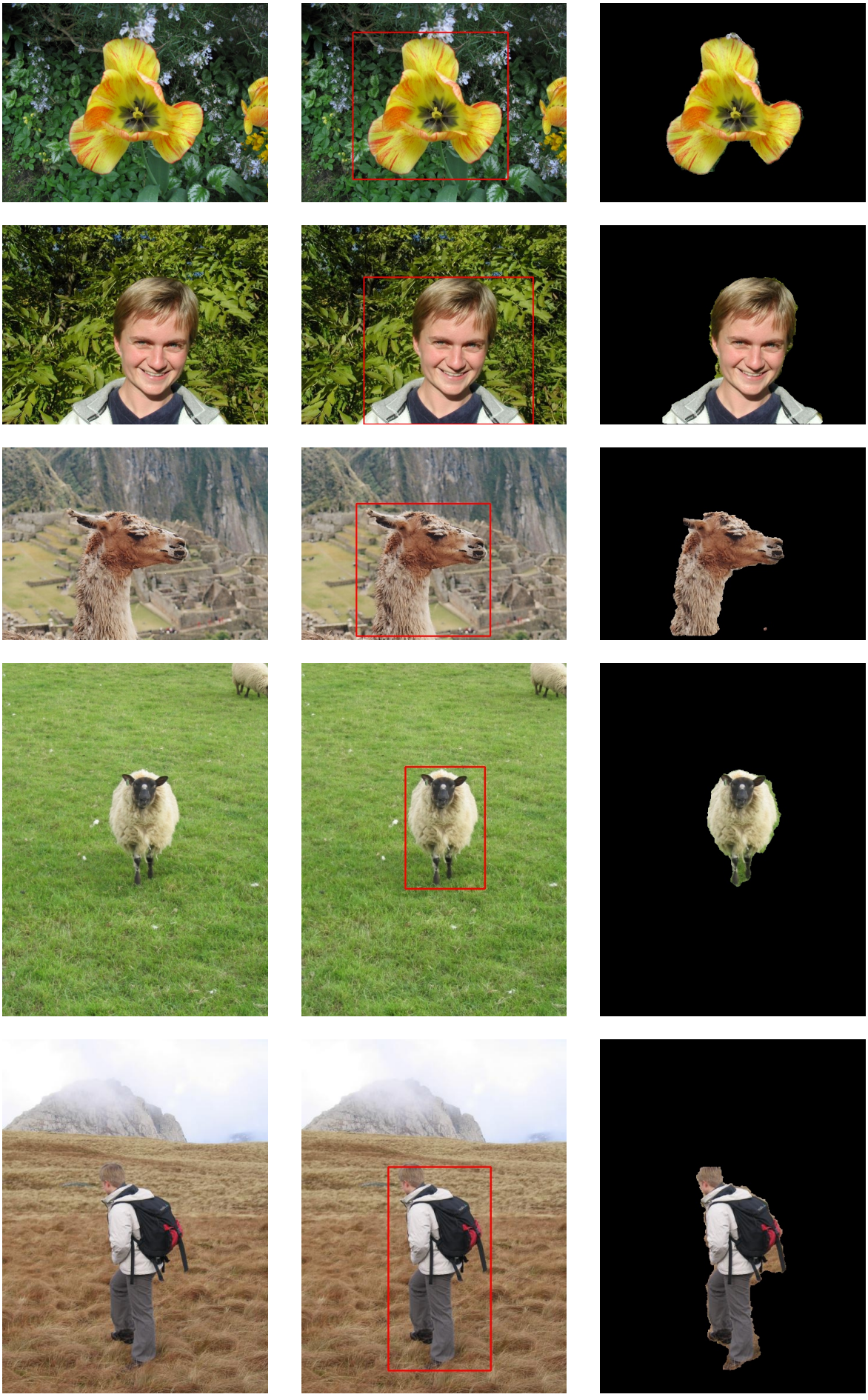

Figure 1: Results

Figure 2: Some more results

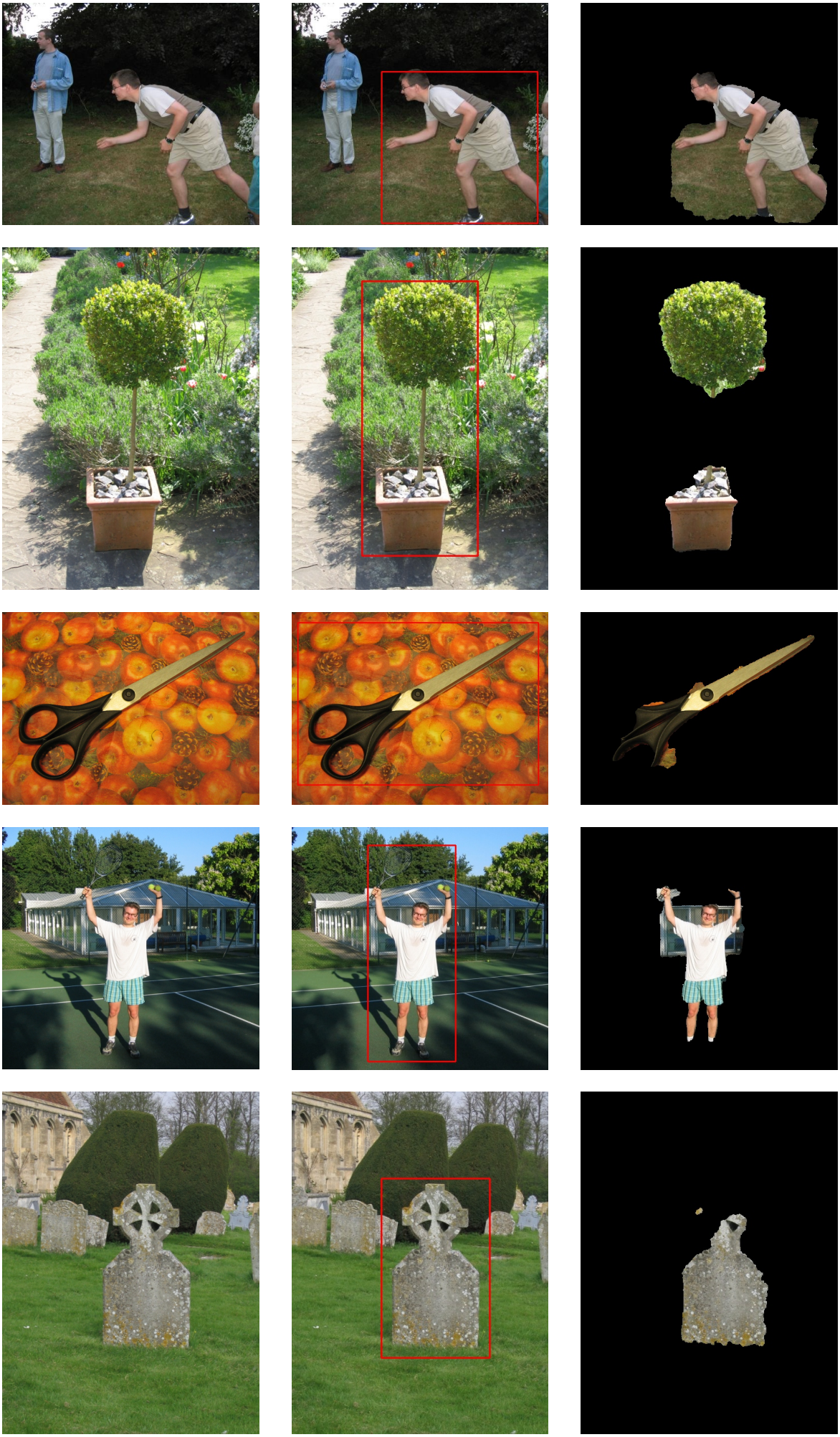

Figure 3: Some bad results