Artificial Vision Report - Lab 5

Olivier Jais-Nielsen

April 29, 2009

Chapter 1

C++ Code

1.1 K-means algorithm

1.1.1 Initialisation

We first assign arbitrarily each point to a center. The criteria adopted here is the one indicated : the point n is associated to the center c where c is the modulus in the division of n by the number of centers. Cf. algorithm 1.1.

Algorithm 1.1 Initialisation

```
1 for( int p = 0; p < pointN; p++ )
2 {
3         pointsAssignment(p) = p % centerN;
4 }
```

1.1.2 Main loop

First, we create a boolean variable `converged` initialized to `false`. It is supposed to represent wether or not the algorithm has converged. The main loop runs until it is equal to `true`.

At each iteration of main the loop, we first recompute the centers' positions.

During the next loop, the list `groupSize` contains, for each center, the number of points associated to the center that have been used to recompute the center's position yet. For instance, before the loop starts, the list contains only 0 and after the loop, it contains for each center the number of points associated to it. At each iteration, a center is equal to the average of all the points associated to it that have been met in the previous iterations. Cf. algorithm 1.2 on the next page.

Then, we recompute each point's assignment. Before that, we initialize `converged` to`true`.

Algorithm 1.2 Recomputing the centers' positions

```
1  CImg<int> groupsSize(centerN, 1, 1, 1, 0);
2  for( int p = 0; p < pointN; p++ )
3  {
4          int c = pointsAssignment(p);
5          centers[c] *= groupsSize(c);
6          centers[c] += points[p];
7          centers[c] /= groupsSize(c) + 1;
8          groupsSize(c) += 1;
9  }
```

For each point `p` we compute `argMin`, the number of the center whose distance (norm 2 of the difference) from the point is minimal. If it is different from the center currently associated with the point, we associate the point with this center. After this loop, the variable `converged` is equal to `true` unless a point has been changed its center. Cf. algorithm 1.3 on the following page.

1.2 Texture segmentation

1.2.1 Gabor filtering

For each frequency `fc`, we filter the image with gabor filters corresponding to this frequency and to n different directions, from 0 to $(n-1)\pi/n$ where n is the number of directions, so that all the directions are different modulus π. Cf. algorithm 1.4 on the next page.

1.2.2 Building the features vectors

The list `features` contains, for each point, the values returned by the corresponding gabor filter for all the frequencies and all the directions. Cf. algorithm 1.5 on page 4.

Algorithm 1.3 Recomputing the points' assignments

```
1  float minDist;
2  float dist;
3  float argMin = 0;
4  converged = true;
5
6  for( int p = 0; p < pointN; p++ )
7  {
8          for( int c = 0; c < centerN; c++ )
9          {
10                 dist = (centers[c] - points[p]).norm(2);
11                 if( c == 0 )
12                 {
13                         minDist = dist;
14                 }
15                 else if( minDist > dist )
16                 {
17                         minDist = dist;
18                         argMin = c;
19                 }
20         }
21         converged = (pointsAssignment(p) == argMin) &&
               converged;
22         pointsAssignment(p) = argMin;
23 }
```

Algorithm 1.4 Gabor filtering

```
1  filtered[i][j] = GaborFilter(sigma, fc, dir, img_raw);
2  dir += M_PI / num_directions;
```

Algorithm 1.5 Building the features vectors

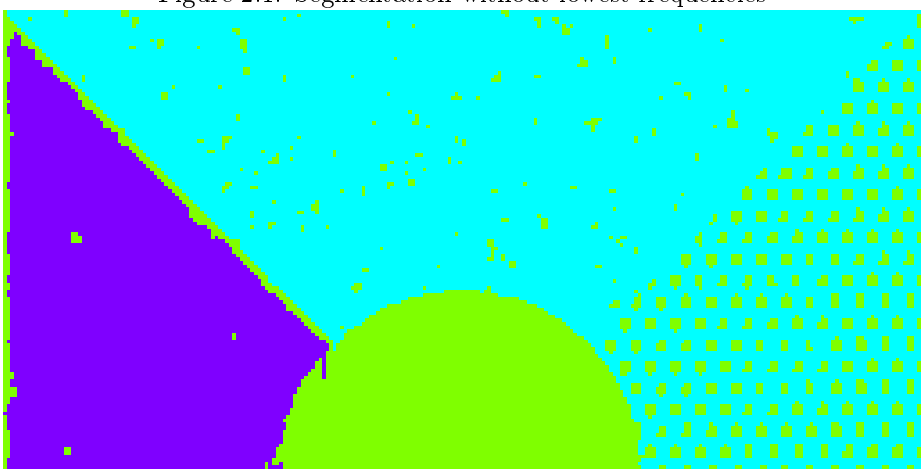
```
1  int p = 0;
2  CImgList<float> features(dimX * dimY, num_freqs,
       num_directions);
3  for( int x = 0; x < dimX; x++ )
4  {
5          for( int y = 0; y < dimY; y++ )
6          {
7                  for( int i = 0; i < num_freqs; i++ )
8                  {
9                          for(int j = 0; j < num_directions
                               ; j++ )
10                         {
11                                 features[p](i,j) =
                                       filtered[i][j](x,y);
12                         }
13                 }
14                 p++;
15         }
16 }
```

Chapter 2

Experiments

The algorithm gets extremely slow, as soon as the spatial bandwidth sigma gets wide. Therefore, I couldn't get the algorithm to segment the lowest frequency pattern zone: the one on the bottom right hand corner. Cf. figure 2.1.

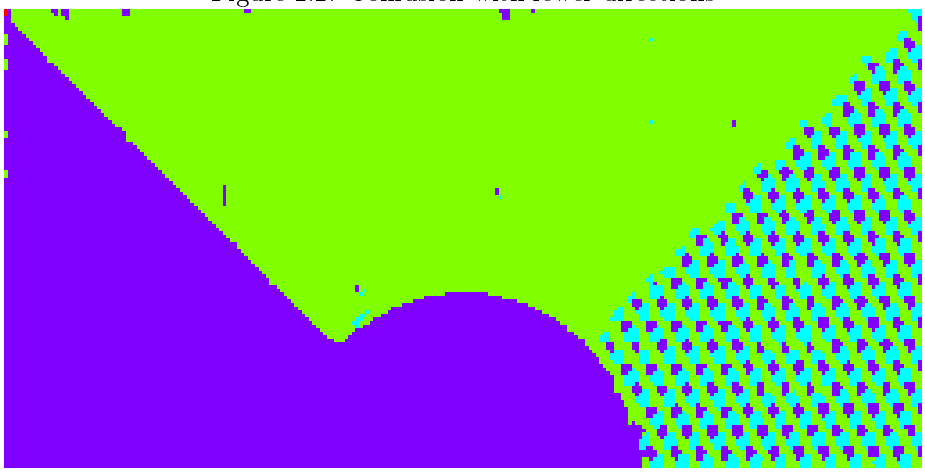
Figure 2.1: Segmentation without lowest frequencies



The parameters used here are: 4 clusers, 4 directions, 7 frequencies and a starting spatial bandwidth sigma equal to 1.

If we limit the number of directions, areas like the bottom centerd one which has a rather horizontal pattern and areas with near frequencies pattern but with different orientations, like the left one (which is indeed not particularly oriented) get confused. Cf. figure 2.2 on the following page.

Figure 2.2: Confusion with fewer directions



The parameters used here are: 4 clusers, 3 directions, 5 frequencies and a starting spatial bandwidth sigma equal to 1.