Applicazioni pratiche della visione artificiale

Parte 1.8– OpenCV

Curato da: Ing. Francesco La Rosa
Università di Messina – Facoltà di Ingegneria

Corso di Calcolatori II – A.A. 2003/2004
Ing. Giancarlo Iannizzotto

K-Means

Questo Algoritmo opera una *classificazione* senza dover preventivamente *imparare* le classi.

- Si suppone che il numero degli oggetti dell'immagine (sfondo compreso) sia noto.
- I pixel vengono iterativamente assegnati a *k* classi.
- Non è sempre disponibile una tecnica ottimale per determinare numero *k* e posizione approssimativa degli oggetti.
- Se gli oggetti non sono *k* o la loro posizione approssimativa non è nota, è possibile giungere a risultati errati.

K-Means

Algoritmo:

1. Definisci il numero *k*;
2. Scegli i *k* punti che faranno da semi iniziali per i cluster da formare;
3. Prima passata: assegna ciascun punto dell'immagine (tranne i *k* semi) al cluster più *vicino*;
4. Per ogni cluster, calcola il class exemplar corrispondente ogni volta che al cluster viene aggiunto un punto;
5. Seconda passata: riclassifica tutti i punti, *compresi* i *k* semi iniziali, in base al class exemplar più vicino.

K-Means

Note:

- Non è detto che alla fine della seconda passata i semi appartengano ancora alle classi originarie!
  - In effetti questo ci permette di scegliere a caso i *k* semi.
- Se il numero *k* non è noto, è necessario ricorrere ad altre tecniche o a raffinamenti successivi.
- Non sempre si considerano le coordinate spaziali fra le features, quindi una classe potrebbe essere composta da oggetti non spazialmente contigui.
- Non abbiamo dato alcuna indicazione su come scegliere la metrica più adatta (dipende dall'applicazione).

K-Means

Cluster = 2

K-Means

Cluster = 3

K-Means

Splits set of vectors into given number of clusters.

void cvKMeans(int *numClusters*, CvVect32f* *samples*, int *numSamples*, int *vecSize*, CvTermCriteria *termcrit*, int* *cluster*);

- *numClusters* Number of required clusters.
- *samples* Pointer to the array of input vectors.
- *numSamples* Number of input vectors.
- *vecSize* Size of every input vector.
- *termcrit* Criteria of iterative algorithm termination.
- *cluster* Characteristic array of cluster numbers, corresponding to each input vector.

K-Means

Discussion

The function KMeans iteratively adjusts mean vectors of every cluster. Termination criteria must be used to stop the execution of the algorithm. At every iteration the convergence value is computed as follows:

$$E = \sum_{i=1}^{K} \left\| old\_mean_i - new\_mean_i \right\|^2$$

The function terminates if $E < Termcrit.epsilon$.

K-Means

```
…
src = cvLoadImage(path);
CvSize size_src=cvSize(src->width,src->height);
IplImage *PlaneB=cvCreateImage(size_src,src->depth,1);
IplImage *PlaneG=cvCreateImage(size_src,src->depth,1);
IplImage *PlaneR=cvCreateImage(size_src,src->depth,1);
cvCvtPixToPlane(src, PlaneB, PlaneG, PlaneR,0);
dst=cvCreateImage(size_src,IPL_DEPTH_8U,1);
 //inserisco da una dialog box il numero di cluster da creare
int* cluster, nvector=0, i, ncluster=10;
CvVect32f *samples; //CvVect32f = = float*
cluster=(int *)malloc((src->width)*(src->height)*sizeof(int));
…
```


K-Means

```
…
samples=(CvVect32f *)malloc((src->width)*
            (src->height)*sizeof(CvVect32f));
CvTermCriteria termcrit;
termcrit.type=CV_TERMCRIT_ITER +
                CV_TERMCRIT_EPS;

termcrit.maxIter=10; termcrit.epsilon=1.0;
unsigned char pixel_mask[3];
…
```

K-Means

```
… for(i=0; i<PlaneB->height; i++)
  {
     for(int j=0; j<PlaneB->width; j++)
     {
       iplGetPixel(PlaneB,j,i,&pixel_mask[0]);
       iplGetPixel(PlaneG,j,i,&pixel_mask[1]);
       iplGetPixel(PlaneR,j,i,&pixel_mask[2]);
       samples[nvector]=(CvVect32f)malloc(3*sizeof(float));
       samples[nvector][0]=(float)pixel_mask[0];
       samples[nvector][1]=(float)pixel_mask[1];
       samples[nvector][2]=(float)pixel_mask[2];
       nvector++;}
  } …
```


K-Means

```
…
  cvKMeans (ncluster, samples, (src->width)*
                 (src->height), 3, termcrit, cluster);
  float step=255/(float) ncluster;
  unsigned char pixel[1]; nvector=0;
  for(i=0; i<dst->height; i++)
  {
      for(int j=0; j<dst->width; j++)
      {
        pixel[0]=(unsigned char)(step*cluster[nvector]);
        iplPutPixel(dst,j,i,pixel);
        nvector++;
      }
  } …
```

K-Means

```
…
    cvNamedWindow("window src",1);
    // apro una finestra denominata window src
    cvShowImage("window src",src);
    cvNamedWindow("window dst",1);
    // apro una finestra denominata window dst
    cvShowImage("window dst",dst);
    cvWaitKey(0);
    nvector=0;
…
```


K-Means

```
…
for(i=0; i<PlaneB->height; i++)
{
    for(int j=0; j<PlaneB->width; j++)
    {
        free(samples[nvector]);
        nvector++;
    }
}
…
```

K-Means

```
…
    free(samples);
    free(cluster);
    cvDestroyWindow("window src");
    cvDestroyWindow("window dst");

    cvReleaseImage (&src);
    cvReleaseImage (&dst);
}
```


K-Means

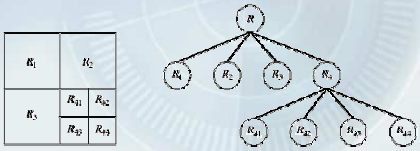
Cluster = 5

Split and Merge

- Consiste nel suddividere inizialmente l'immagine in un insieme di regioni arbitrarie e disgiunte, quindi fondere e/o dividere le regioni con il fine di soddisfare un partizionamento dell'immagine in regioni disgiunte ed internamente omogenee e connesse.
- Esistono *diversi algoritmi* che usano questa idea di base. La maggior parte di essi sono sostanzialmente variazioni dell'algoritmo *Quadtree Decomposition*. Ciascuna variante mira ad ottimizzare le prestazioni ottenibili.

Quadtree decomposition

- Realizzazione recursiva del principio dello split and merge, in cui le regioni sono quadrate.



Quadtree decomposition

- Indicata con R la regione costituente l'intera immagine e con R_i la generica subimage e scelto un predicato $P(R_i)$, viene applicata la seguente procedura:
  - Split in quattro quadranti di tutte le regioni per cui

$$P(R_i) = FALSE;$$

- Merge di ogni coppia di regioni adiacenti (R_i, R_j) per cui

$$P(R_j \bigcup R_k) = TRUE;$$

- Stop quando ulteriori split & merge sono impossibili.

Il predicato $P(R_i)$ può essere del tipo: $\sigma^2(R_i) > T$

Quadtree decomposition

DEMO MATLAB

Snake

ø This section describes a function for working with active contours, also called snakes.The snake was presented in [*Kass88*] as an energy-minimizing parametric closed curve guided by external forces. Energy function associated with the snake is:

$$E = E_{int} + E_{ext}$$

where E_{int} is the internal energy formed by the snake configuration, E_{ext} is the external energy formed by external forces affecting the snake. The aim of the snake is to find a location that minimizes energy.

parte 1.8 – OpenCV

Snake

ø Let $p_1, \ldots, p_n$ be a discrete representation of a snake, that is, a sequence of points on an image plane. In OpenCV the internal energy function is the sum of the contour continuity energy and the contour curvature energy, as follows:

$$E_{int} = E_{cont} + E_{curv}$$

VisiLAB Computer Vision and Image Processing Lab - University of Messina

Snake

- E_{cont} is the contour continuity energy. This energy is $E_{cont} = |\delta - \|p_i - p_{i-1}\|\,|$, where δ is the average distance between all pairs (p_i, p_{i-1}). Minimizing over all the snake points, causes the snake points $p_1, \ldots, p_n$ become more equidistant.
- E_{curv} is the contour curvature energy. The smoother the contour is, the less is the curvature energy. $Ecurv = |\, p_{i-1} - 2p_i - p_{i+1}|^2$

parte 1.8 – OpenCV

Snake

- In [*Kass88*] external energy was represented as $E_{ext} = E_{img} + E_{con}$ where E_{img} – image energy and E_{con} – energy of additional constraints.
- Two variants of image energy are proposed:
  1. $E_{img} = -I$ where I is the image intensity. In this case the snake is attracted to the bright lines of the image.
  2. $E_{img} = -|\, grad(I)\,|$. The snake is attracted to the image edges.

VisiLAB Computer Vision and Image Processing Lab - University of Messina

Snake

- Summary energy at every point can be written as

  $E_i = \alpha_i E_{cont,\, i} + \beta_i E_{curv,\, i} + \gamma_i E_{img,\, i}$ (1)

  where α, β, γ are the weights of every kind of energy.

  The full snake energy is the sum of E_i over all the points.

- The *meanings* of α, β, γ are as follows:

parte 1.8 – OpenCV

Snake

- *α* is responsible for contour continuity, that is, a big α makes snake points more evenly spaced.
- *β* is responsible for snake corners, that is, a big β for a certain point makes the angle between snake edges more obtuse.
- *γ* is responsible for making the snake point more sensitive to the image energy, rather than to continuity or curvature.

VisiLAB Computer Vision and Image Processing Lab - University of Messina

Snake

- There are three well-known algorithms for minimizing snake energy. In [*Kass88*] the minimization is based on variational calculus. In [*Yuille89*] dynamic programming is used.The greedy algorithm is proposed in [*Williams92*].
- The latter algorithm is the most efficient and yields quite good results. The scheme of this algorithm for each snake point is as follows:

parte 1.8 – OpenCV

Snake

1. Use Equation (1) to compute E for every location from point neighborhood. Before computing E, each energy term E_{cont}, E_{curv}, E_{img} must be normalized using formula:

   $E_{normalized} = (E_{img} - min)/ (max - min)$,

   where *max* and *min* are maximal and minimal energy in scanned neighborhood.
2. Choose location with minimum energy.
3. Move snakes point to this location.
4. Repeat all the steps until convergence is reached.

VisiLAB Computer Vision and Image Processing Lab - University of Messina

Snake

- Criteria of convergence are as follows:
  - maximum number of iterations is achieved;
  - number of points, moved at last iteration, is less than given threshold.

Snake

```
…
char path[256]; IplImage *image1, *image;
CvPoint  points[100];
int coeffUsage=CV_VALUE,  i;
imopen(path); //scelgo il path dell'immagine da aprire
float alpha=(float)1, beta=(float)1, gamma=(float)1.2;
CvSize win=cvSize(101,101);
CvTermCriteria criteria=cvTermCriteria( CV_TERMCRIT_EPS +
    CV_TERMCRIT_ITER, 2, 100000); …
```

Snake

```
…
image=cvLoadImage(path,-1);
image1=cvCloneImage(image);
cvNamedWindow("results", CV_WINDOW_AUTOSIZE);
cvSetMouseCallback( "results", on_mouse );
cvShowImage("results", image);
cvWaitKey(0);
…
```


Snake

```
…
cvSnakeImage( image, points, length, &alpha, &beta, &gamma,
                    coeffUsage, win, criteria, 1);

for(i=0; i<length;i++)  cvCircle(image1,points[i], 3,
                                        CV_RGB(0,0,255),1);

cvShowImage("results", image1);
…
```

Snake

```
…
cvWaitKey(0);
cvDestroyAllWindows();
cvReleaseImage(&image);
cvReleaseImage(&image1);
…
```


Snake

```
void on_mouse( int event, int x, int y, int flags )
{    switch( event )
        {
        case CV_EVENT_LBUTTONDOWN:
                {
                points[length]=cvPoint(x,y);
                cvCircle(image,points[length], 3,
                        CV_RGB(0,0,255),1);
…
```

Snake

```
…
                length++;
                cvShowImage("results",image);
               }
               break;
          }//fine switch

}//fine on_mouse
```


Hough transform

- Per effettuare un'operazione di edge-linking vengono usate relazioni globali tra pixel (Global processing).
- I punti sono collegati associandoli a curve di forma prestabilita (ad es.: rette, circonferenze, ecc.).
- Presuppone la conoscenza della forma cercata.

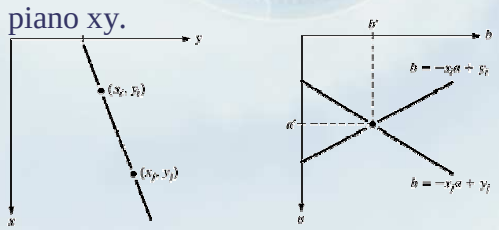
Hough transform

- Si consideri un punto di edge avente coordinate (x, y) e le infinite rette passanti per tale punto.
- Le rette in questione presentano una relazione tra i coefficienti a (pendenza) e b (intercetta) del tipo:

$$y_i = ax_i + b$$
$$b = -x_i a + y_i$$

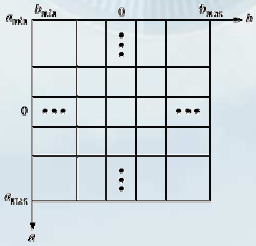
Hough transform

- Rappresentando la relazione matematica che intercorre tra i parametri a e b in un piano cartesiano (chiamato *spazio dei parametri*) si ottiene una retta.
- Dato un secondo punto (x_j,y_j), anch'esso ha una retta associata nello spazio dei parametri. Tale retta interseca quella associata ad (x_i,y_i) in (a',b').
- a' e b' sono i parametri della retta passante per entrambe i punti nel piano xy.



Hough transform

- Tutti i punti che si trovano sulla stessa retta, nel piano xy, hanno ad essi associati, nello spazio dei parametri, delle rette che si intersecano in un punto (a',b').
- La hough transform nasce dalla suddivisione dello spazio dei parametri in celle (rettangolari) di accumulazione, limitando il range dei valori ad (a_{min} , a_{max}) e (b_{min} , b_{max}).

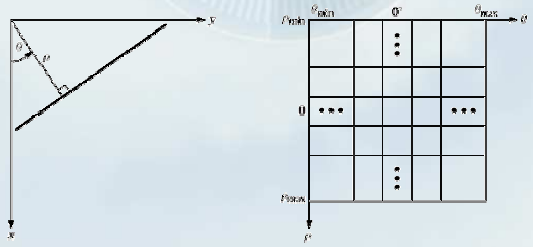


Hough transform

- Per ogni punto (x_k ,y_k) del piano dell'immagine si pone il parametro *a* uguale ad ognuno dei valori consentiti (a_i) e si calcola il corrispondente *b* usando l'equazione $b = -x_k a + y_k$
- I valori di *b* trovati vengono arrotondati al più vicino valore di *b* consentito (b_j) .
- Se il valore a_p restituisce b_q si tiene traccia dell'evento incrementando di uno l'elemento (p,q) della matrice A (matrice di accumulazione).
- Alla fine di questa procedura, un valore intero *M* in A(i,j) corrisponde ad *M* punti che nel piano *xy* stanno sulla retta $y = a_i x + b_j$. L'accuratezza con cui i punti sono allineati è determinata dal numero di divisioni nel piano *ab*.

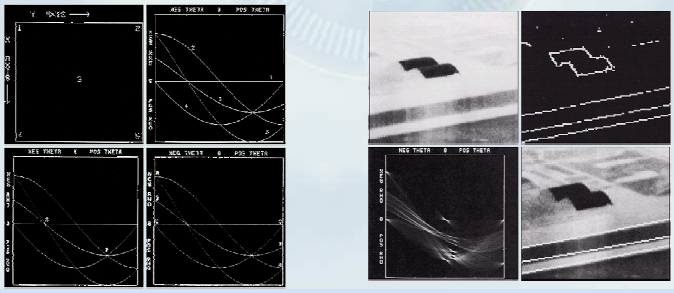
Hough transform

- Un problema nella rappresentazione delle rette tramite l'equazione y = ax + b è che pendenza (a) ed intercetta (b) per linee verticali tendono ad infinito.
- Un modo per superare la difficoltà è usare una rappresentazione polare della retta: $x \cos\theta + y \sin\theta = \rho$.



Hough transform

- L'uso della rappresentazione polare implica l'uso dei parametri ρ e θ.
- *M* punti allineati nel piano xy ($x \cos\theta_j + y \sin\theta_j = \rho_i$) corrispondono ad *M* sinusoidi che s'intersecano nel punto (ρ_i , θ_j) del piano $\rho\theta$.



Hough transform

```
…
IplImage* src;
if( src=cvLoadImage(".\\palazzi.bmp", 0))
{
  IplImage* dst = cvCreateImage( cvGetSize(src), 8, 1 );
  IplImage* color_dst = cvCreateImage( cvGetSize(src), 8, 3 );
  CvMemStorage* storage = cvCreateMemStorage(0);
  CvSeq* lines = 0; int i;
…
```


Hough transform

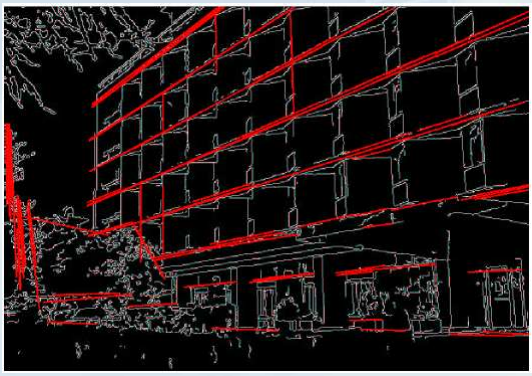
```
…
cvCanny( src, dst, 50, 200, 3 );
cvCvtColor( dst, color_dst, CV_GRAY2BGR );
lines = cvHoughLines2( dst, storage,
    CV_HOUGH_PROBABILISTIC, 1, CV_PI/180, 80, 30, 10 );
for( i = 0; i < lines->total; i++ )
{
 CvPoint* line = (CvPoint*)cvGetSeqElem(lines,i);
 cvLine( color_dst, line[0], line[1], CV_RGB(255,0,0), 3, 8 );} …
```

Hough transform

```
…
    cvNamedWindow( "Source", 1 );
    cvShowImage( "Source", src );
    cvNamedWindow( "Hough", 1 );
    cvShowImage( "Hough", color_dst );
    cvWaitKey(0);
    }
cvDestroyAllWindows();
}
```


Hough transform

Hough transform



Hough transform

The Hough transform variant:

- CV_HOUGH_STANDARD - classical or standard Hough transform. Every line is represented by two floating-point numbers (r, q), where r is a distance between (0,0) point and the line, and q is the angle between x-axis and the normal to the line. Thus, the matrix must be (the created sequence will be) of CV_32FC2 type.

Hough transform

The Hough transform variant:

- CV_HOUGH_PROBABILISTIC - probabilistic Hough transform (more efficient in case if picture contains a few long linear segments). It returns line segments rather than the whole lines. Every segment is represented by starting and ending points, and the matrix must be (the created sequence will be) of CV_32SC4 type.
- CV_HOUGH_MULTI_SCALE - multi-scale variant of classical Hough transform. The lines are encoded the same way as in CV_HOUGH_CLASSICAL.