Visual Computing Lab 4

CHAKKA Karthik Subramanyam

CHATTERJEE Jit

TP4 - Image Segmentation

The objective of this practical work is to study image segmentation through the K-means method.

1) The method

K-means consists in finding regions in the image that minimize the following energy:

$$F(regions, pixels) = \sum_{i \in regions} \sum_{j \in region\ i} (x_j - c_i)^t (x_j - c_i),$$

where x_j is the value taken into account at pixel j and c_i is the value of the center of region i. Starting from an initial solution, the principle is to iterate 2 steps:

1) Assuming known region centers, associate each pixel to its closest region (i.e. with the minimum distance to the region centre).

2) Assuming known associations between pixel and regions, determine new region centers as the mean of pixel values in each region.

1) Introduction:

```
K-Means clustering algorithm is an unsupervised algorithm and it is
 used to segment the interest area from the background. It clusters,
 or partitions the given data into K-clusters or parts based on the K-
centroids.
```

Goal:

```
To apply K-Means to preform image segmentation. The goal is to find
 certain groups based on some kind of similarity in the data with the
 number of groups represented by K.
```

Procedure:

```
The objective of K-Means clustering is to minimize the sum of squared
 distances between all points and the cluster center:
```

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \left\| x_i^{(j)} - c_j \right\|^2$$

```
where, J is the objective function, k = number of clusters, n = number
 of cases, x~i~ = case i, c~j~ = centroid cluster.
```

Steps:

1) First we have to choose to choose the number of clusters K:

```
Suppose K = 3
```

2) Select at random K points, the centroids:

```
We have generated random K points with the use of rand function which
 generates a random list whose max value is not more than 255.
```

```
For a ppm image containing RGB, if K = 3, then the random list will
 contain 9 pixels as:
```

R	G	B	R	G	B	R	G	B

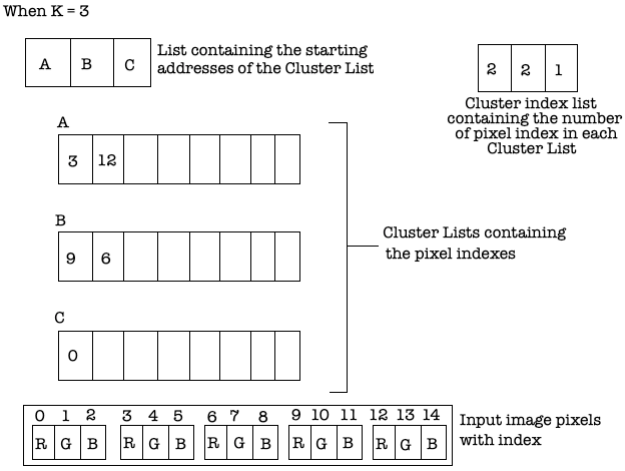
Code Snippet:

```
mat* genRandMat(size_t row, size_t col, size_t dim, mat *maxVal){
    struct timespec spec;
    clock_gettime(CLOCK_REALTIME, &spec);
    srand( (unsigned int) spec.tv_nsec );
    mat *rMat = allocMatMem(row, col, dim);
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            for (size_t k = 0; k < dim; ++k) {
                rMat[(i * col * dim) + (j * dim) + k] = rand() %
 (maxVal[k] + 1);
            }
        }
    }
    return rMat;
}

mat* genRandLst(size_t nbEle, size_t dim, mat *maxVal){
    return genRandMat(1, nbEle, dim, maxVal);
}
```

3) Assign each data point to the closest centroid, that forms K clusters:

```
First we will create a List containing the starting address of the
 Cluster Lists. We will also create a Cluster Index List containing
 the number of pixels in each Cluster List. Each time a new pixel index
 gets added in a Cluster List, the Cluster Index List will get updated.
 Initially, the Cluster Index List will be assigned to 0.
```



To get the closest centroid, we have to find the distance of the current image pixel with all the centroids and the image pixel will be added to the cluster list of the centroid which will have the minimum distance.

$$Distance = \sqrt{(R' - R)^2 + (G' - G)^2 + (B' - B)^2}$$

Code Snippet:

```
unsigned int calcDist(mat* a, mat* b, size_t dim){
    unsigned int sqSum = 0;
    for (size_t k = 0; k < dim; ++k) {
        int d = a[k] - b[k];
        sqSum = sqSum + ((unsigned)(d * d));
    }
    unsigned int dist = (unsigned int) +sqrt(sqSum);
    return dist;
}

size_t getArrMinEleIndex(mat* lst, size_t nbEle){
    size_t minIndex = 0;
    for (size_t i = 0; i < nbEle; ++i) {
        if(lst[minIndex] > lst[i]){
            minIndex = i;
        }
    }
    return minIndex;
}
```

```
void addToArray(mat* arr, size_t nbMaxEle, mat *idx, mat ele){
    assert(*idx < (signed)nbMaxEle);
    arr[*idx] = ele;
    *idx += 1;
}
```

Code Snippet:

```
size_t getNearestCentroidIndex(mat *centLst, mat* curPix, size_t K,
 size_t dim){
    chkLstValidity(centLst, K, dim);
    mat *distLst = allocLstMem(K, 1);
    mat initVal = 0;
    memsetLst(distLst, &initVal, K, 1);
    for (size_t i = 0; i < 1; ++i) {
        for (size_t j = 0; j < K; ++j) {
            distLst[j] = (signed) calcDist(&centLst[(i * K * dim) + (j *
 dim)], curPix, dim);
        }
    }
    size_t idx = getArrMinEleIndex((mat*)distLst, K);
    free(distLst);
    return idx;
}

void buildClusters(mat** clustLst, mat* clustLstCurIdx, mat* centLst,
 mat* imgMat, size_t row, size_t col, size_t dim, size_t K){
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            mat curPixIdx = (signed)(i * col * dim) + (signed)(j *
 dim); // index of start of current pix in imgMat array
            mat *curPix = &imgMat[curPixIdx];
            size_t nCentIdx = getNearestCentroidIndex(centLst, curPix,
 K, dim);
            addToArray( clustLst[nCentIdx], row * col,
 &clustLstCurIdx[nCentIdx], (signed)curPixIdx );
        }
    }
}
```

4) Compute and place the new centroid (region centers) of each cluster:

```
After step 3, we will get the cluster list and the cluster list index.
 So, now to get the new centroid, we need to add the pixels in each
 Cluster list and divide by the Cluster List index of that specific
 cluster which contains the number of pixel indexes in that cluster
 therefore, calculating the mean value in each cluster.
```

Code Snippet:

```
void getNewCentroids(mat** clustLst, mat* clustLstCurIdx, mat* imgMat,
 size_t dim, mat *newCentLst, size_t nbClust){
    for (size_t i = 0; i < nbClust; ++i) {
        mat *curClust = clustLst[i];
        if(clustLstCurIdx[i] != 0){
            mat *avgMat = allocLstMem((unsigned) clustLstCurIdx[i],
 dim);
            buildMatForAvg(avgMat, imgMat, curClust,
 (unsigned)clustLstCurIdx[i], dim);
            lstAvg(avgMat, (unsigned) clustLstCurIdx[i], dim,
 &newCentLst[i * dim]);
            free(avgMat);
        }
    }
}

void buildMatForAvg(mat *matr, mat* imgMat, mat *clust, size_t nbEle,
 size_t dim){
    for (size_t i = 0; i < nbEle; ++i) {
        cpyLst( &matr[i * dim], &imgMat[clust[i]], 1, dim);
    }
}

void matAvg(mat *matr, size_t row, size_t col, size_t dim, mat* res){
    for (size_t k = 0; k < dim; ++k) {
        res[k] = 0;
    }
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            for (size_t k = 0; k < dim; ++k) {
                res[k] = res[k] + matr[(i * col * dim) + (j * dim) + k];
            }
        }
    }
```

```
    for (size_t k = 0; k < dim; ++k) {
        res[k] = (mat) ( res[k] / (1.0 * row * col) );
    }
}

void lstAvg(mat *lst, size_t nbEle, size_t dim, mat *res){
    matAvg(lst, 1, nbEle, dim, res);
}
```

---

2) Implementation

1) Implement K-means with x values being the intensity values in the image.

2) What is the influence of the initial values for region centers?

3) What is the influence of the number of regions K ?

4) Consider now for x values both intensity and location in the image:

a. How does it change the results ?

b. How can we balance the influence of colors and locations in the image ?

2.1) Implementing K-means:

By successfully computing Steps 1,2,3 and 4 in the above part, we get the new centroids for each cluster.

Step 5: Now, we have to compare the new centroid list with the previous centroid list. If, both the lists are same we will build the Output segmented image. Otherwise, we will reassign each pixel to the new closest centroid and repeat step 4 till we get same centroid list.

Code Snippet:

```
void applyKMeans(mat* opMat, mat *imgMat, size_t row, size_t col, size_t
 dim, size_t K){
    mat *maxLst = allocLstMem(1, dim);
    mat initVal = MAX_PIX_VAL;
    memsetLst(maxLst, &initVal, dim, 1);
```

```
    mat *centLst = genRandLst(K, dim, maxLst);
    free(maxLst);
    mat *newCentLst = allocLstMem(K, dim);
    bool stopItr = false;
    while(stopItr == false){
        mat **clustLst = allocClusterLst(row * col, K);
        mat *clustLstCurIdx = allocLstMem(K, 1);
        mat initClustIdxVal = 0;
        memsetLst(clustLstCurIdx, &initClustIdxVal, K, 1);
        buildClusters(clustLst, clustLstCurIdx, centLst, imgMat, row,
 col, dim, K);
        getNewCentroids(clustLst, clustLstCurIdx, imgMat, dim,
 newCentLst, K);
        if( ( stopItr = cmpLst(centLst, newCentLst, K, dim) ) == true){
            buildOpMat(opMat, clustLst, clustLstCurIdx, centLst, dim,
 K);
        }
        free(clustLstCurIdx);
        deallocClusterLst(clustLst, K);
        cpyLst(centLst, newCentLst, K, dim);
    }
    free(newCentLst);
    free(centLst);
}

bool cmpLst(mat *a, mat *b, size_t nbEle, size_t dim){
    return matCmp(a, b, 1, nbEle, dim);
}

bool matCmp(mat *mat1, mat* mat2, size_t row, size_t col, size_t dim){
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            for (size_t k = 0; k < dim; ++k) {
                size_t index = (i * col * dim) + (j * dim) + k;
                if( mat1[index] != mat2[index]){
                    return false;
                }
            }
        }
    }
    return true;
}

void memsetLst(mat *lst, mat *val, size_t nbEle, size_t dim){
    memsetMatrix(lst, val, 1, nbEle, dim);
```

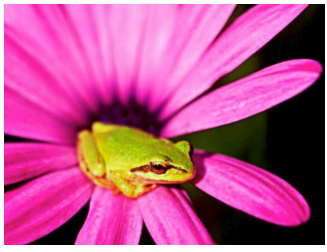
```
}

void memsetMatrix(mat *matr, mat *val, size_t row, size_t col, size_t
 dim){
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            for (size_t k = 0; k < dim; ++k) {
                matr[(i * col * dim) + (j * dim) + k] = val[k];
            }
        }
    }
}
```

Code Snippet:

```
void buildOpMat(mat *opMat, mat **clustLst, mat *clustLstCurIdx, mat
 *centLst, size_t dim, size_t nbClust){
    for (size_t i = 0; i < nbClust; ++i) {
        size_t curClustSiz = (unsigned) clustLstCurIdx[i];
        if(curClustSiz == 0){
            continue;
        }
        mat *curClust = clustLst[i];
        mat *curCent = &centLst[i * dim];
        for (size_t j = 0; j < curClustSiz; ++j) {
            mat curIdx = curClust[j];
            cpyLst(&opMat[(unsigned)curIdx], curCent, 1, dim);
        }
    }
}
```

Input Image:



Input Image (Fig 2.1)

Output Image:

```
Image Segmentation of Fig 2.1 in RGB domain by applying K means, where K
 = 3
                                  (Fig 2.2)
```

```
Image Segmentation of Fig 2.1 in RGB domain by applying K means, where K
 = 3
                                  (Fig 2.3)
```

Results:

```
As, we have randomly choosen the values of the centroids, thats why
 we see difference between Fig 2.2 and Fig 2.3. Both the images are
 generated by applying K means on Fig 2.1 but with different centroid
 values.
```

```
Image Segmentation of Fig 2.1 in RGB domain by applying K means, where
 K=10
                                    (Fig 2.4)
```

```
Image Segmentation of Fig 2.1 in RGB domain by applying K means, where
 K=10
                                    (Fig 2.5)
```

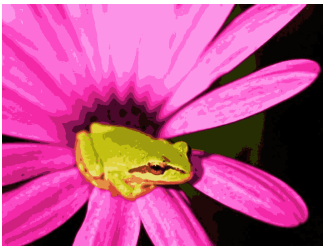
Results:

```
Similarly, for K = 10, we get Fig 2.4 and Fig 2.5
```

2.2) From the above results and Fig 2.4 and 2.5 clearly states that changing the initial centroid (region center) values, changes the image segmentation and we get two different images.

2.3) From the above results and Fig 2.2, 2.4 and Fig 2.6 clearly states increasing the value of K will always reduce the amount of error in the resulting clustering, to the extreme case of zero error if each data point is considered its own cluster

(i.e., when K equals the number of data points, n). In general, a large K probably decreases the error but increases the risk of overfitting.



```
Image Segmentation of Fig 2.1 in RGB domain by applying K means, where
 K=20
                                    (Fig 2.6)
```

2.4) K Means on RGB + Spatial (location) Domain(X,Y):

```
The steps will be similar to the above process, but due to the
 introduction of spatial domain, we get two new values of X and Y. Thus,
 during calculating the distance from the centroids we have to:
```

$$Distance = \sqrt{(R'-R)^2+(G'-G)^2+(B'-B)^2+(X'-X)^2+(Y'-Y)^2}$$

```
For a ppm image containing RGB, if K = 3, then the random list will
 contain 15 pixels as:
```

R	G	B	X	Y	R	G	B	X	Y	R	G	B	X	Y

```
For X and Y, max value will be the input image matrix dimensions.
```

```
The above 5 steps for Image Segmentation by using K means will be
 applied with just 2 new dimensions X and Y (spatial) which needs to be
 considered.
```

```
So, here we are converting the image matrix to an image spatial matrix
 containing the X and Y values for each RGB values.
```

Code Snippet:

```
mat *imgMatToImgSpatialMat(mat *imgMat, size_t row, size_t col, size_t
 *dim){
    size_t colDim = *dim;
    *dim += 2;
    mat *spatImgMat = allocMatMem(row, col, *dim);
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            for (size_t k = 0; k < *dim; ++k) {
                size_t spatImgMatIdx = (i * col * *dim) + (j * *dim) +
 k;
                if(k < colDim){
                    size_t imgMatIdx = spatImgMatIdx - (2 * i * col) -
 (2 * j);
                    spatImgMat[spatImgMatIdx] = imgMat[imgMatIdx];
                } else if (k == colDim){
                    spatImgMat[spatImgMatIdx] = (signed)i;
                } else {
                    spatImgMat[spatImgMatIdx] = (signed)j;
                }
            }
        }
    }
    return spatImgMat;
}
```

Further, applying all the steps till step 5, when we get the same centroid list, we will generate the segmented image. Here, before generating the final segmented image we will convert the image spatial matrix to the image matrix by eliminating the X and Y values.

Code Snippet:

```
void imgSpatialMatToImgMat(mat *imgMat, mat *spatImgMat, size_t row,
 size_t col, size_t *dim){
    *dim -= 2;
    for (size_t i = 0; i < row; ++i) {
        for (size_t j = 0; j < col; ++j) {
            for (size_t k = 0; k < *dim; ++k) {
                size_t imgMatIdx = (i * col * *dim) + (j * *dim) + k;
                size_t spatImgMatIdx = imgMatIdx + (2 * i * col) + (2 *
 j);
```

```
                imgMat[imgMatIdx] = spatImgMat[spatImgMatIdx];
            }
        }
    }
}

void applySpatialKMeans(mat* opMat, mat *imgMat, size_t row, size_t col,
 size_t dim, size_t K){
    size_t colDim = dim;
    mat *spatImgMat = imgMatToImgSpatialMat(imgMat, row, col, &dim);
    mat *spatImgOpMat = allocMatMem(row, col, dim);
    mat *maxLst = allocLstMem(1, dim);
    mat initVal = MAX_PIX_VAL;
    memsetLst(maxLst, &initVal, colDim, 1);
    maxLst[colDim] = (signed)row;
    maxLst[colDim + 1] = (signed)col;
    mat *centLst = genRandLst(K, dim, maxLst);
    free(maxLst);
    mat *newCentLst = allocLstMem(K, dim);
    bool stopItr = false;
    while(stopItr == false){
        mat **clustLst = allocClusterLst(row * col, K);
        mat *clustLstCurIdx = allocLstMem(K, 1);
        mat initClustIdxVal = 0;
        memsetLst(clustLstCurIdx, &initClustIdxVal, K, 1);
        buildClusters(clustLst, clustLstCurIdx, centLst, spatImgMat,
 row, col, dim, K);
        getNewCentroids(clustLst, clustLstCurIdx, spatImgMat, dim,
 newCentLst, K);
        if( ( stopItr = cmpLst(centLst, newCentLst, K, dim) ) == true){
            buildOpMat(spatImgOpMat, clustLst, clustLstCurIdx, centLst,
 dim, K);
            imgSpatialMatToImgMat(opMat, spatImgOpMat, row, col, &dim);
            free(spatImgMat);
            free(spatImgOpMat);
        }
        free(clustLstCurIdx);
        deallocClusterLst(clustLst, K);
        cpyLst(centLst, newCentLst, K, dim);
    }
    free(newCentLst);
    free(centLst);
```

Output Image:

```
Image Segmentation of Fig 2.1 in RGB + Spatial (location) domain by
 applying K means, where K = 3
                                  (Fig 2.7)
```

```
Image Segmentation of Fig 2.1 in RGB + Spatial (location) domain by
 applying K means, where K = 3
                                  (Fig 2.8)
```

```
Image Segmentation of Fig 2.1 in RGB + Spatial (location) domain by
 applying K means, where K=10
                                  (Fig 2.9)
```

```
Image Segmentation of Fig 2.1 in RGB + Spatial (location) domain by
 applying K means, where K=10
                                (Fig 2.10)
```

```
Image Segmentation of Fig 2.1 in RGB + Spatial (location) domain by
 applying K means, where K=20
                                (Fig 2.11)
```

Results:

```
We can clearly see from Fig 2.7, Fig 2.9 and Fig 2.11, that increasing
 the value of K, increases the image segmenation parts and we are
 getting more colors in the segmented image.
```

2.4.a) Due to the inclusion of the spatial (location) domain X and Y, localization of the colors occur in the segmented images.

2.4.b) If we know the goal that what part of the image needs to be segmented, we can set a balance between the influence of colors and location of the image.

Conclusion:

```
   We can conclude that, K-Means is relatively an efficient method for
image segmentation. However, we need to specify the number of clusters,
in advance and the final results are sensitive to initialization and
often terminates at a local optimum.
```
