Computer Vision Assignment 3

Grab Cut and MRF

By- Apoorva Srivastava (2019702014)

Grab Cut:

Introduction:

The Grab Cut paper addresses the problem of efficient, interactive extraction of a foreground object in a complex environment whose background cannot be trivially subtracted. The resulting foreground object is an alpha-matte which reflects the proportion of foreground and background. The aim is to achieve high performance at the cost of only modest interactive effort on the part of the user. High performance in this task includes: accurate segmentation of object from background; subjectively convincing alpha values, in response to blur, mixed pixels and transparency; clean foreground color, free of color bleeding from the source background. In general, degrees of interactive effort range from editing individual pixels, at the labor-intensive extreme, to merely touching foreground and/or background in a few locations.

Classical image segmentation tools use either texture (color) information, e.g. Magic Wand, or edge (contrast) information, e.g. Intelligent Scissors. Recently, an approach based on optimization by graph-cut has been developed which successfully combines both types of information. In this paper we extend the graph-cut approach in three respects. First, we have developed a more powerful, iterative version of the optimization. Secondly, the power of the iterative algorithm is used to simplify substantially the user interaction needed for a given quality of result. Thirdly, a robust algorithm for “border matting” has been developed to estimate simultaneously the alpha-matte around an object boundary and the colors of foreground pixels.

Algorithm with Code:

Step1: Load the image and mark the region outside the bounding box as fixed background considering user input to be absolutely correct and make the corresponding Matte Variable=0.Mark all the pixels within the bounding box to be tentatively as foreground.

```
#Loading required libraries
import cv2
import numpy as np
from sklearn import mixture
from sklearn.cluster import KMeans
import math
from igraph import *
```

```
#Main function
if __name__ == "__main__":
    im_name='teddy'
    img=cv2.imread('./images/'+im_name+'.jpg') #Function to load image
    size=img.shape
    h=size[0]   #number of rows in img
    w=size[1]   #number of col in img
    Fore=(-1,-1) #Foreground Terminal
    Back=(h,w)   #Backgground Terminal
    gamma=5    #Parameter to be multiplied with N link weeight
    num_cluster=5 #Number of GMM Cluster
    neighbourhood=8 #Neighbourhood of Pixels;Either 4 or 8
    bb=open('./bb/'+im_name+'.txt') #(x,y,x+w,y+h) Bounding Box
    f=bb.readlines()
    f='.'.join(str(x) for x in f)
    pix = [int(i) for i in f.split() if i.isdigit()] #Bounding Box coordinates
    bb_h=pix[3]-pix[1]   #Bounding Box Height
    bb_w=pix[2]-pix[0]   #Bounding Box Width
    X_unknown=np.empty((bb_h,bb_w,3))  #Collection of unknown pixels i.e. pixels inside the bounding b
    Matte=np.empty((h,w))              #Binary Matrix to store Segmentation '0'-->Background '1'-->For

    fix_back=[]
```

```
    fix_back_idx=[]
    X_un=[]
    idx_un=[]
    for i in range(pix[1],pix[3],1):  #Tentatively initializing all unknown pixels as foreground
        for j in range(pix[0],pix[2],1):  #pix[0]-->pix[2] along w pix[1]-->pix[3] along h
            Matte[i,j]=1
            X_un.append(img[i,j])
            idx_un.append((i,j))
    for i in range(h):                  #Storing the sure background area
        for j in range(w):
            if Matte[i,j]!=1:
                fix_back.append(img[i,j])
                fix_back_idx.append((i,j))
                Matte[i,j]=0
    #print(len(fix_back),len(X_un))
    img3=np.empty((h,w,3))
    for i in range(h):
        for j in range(w):
            if Matte[i,j]==0:
                img3[i,j]=[0,0,0]
            elif Matte[i,j]==1:
                img3[i,j]=img[i,j]
```

```
cv2.imwrite('./V7/bb_'+im_name+'.jpg',img3)
cnt=0
for i in range(bb_h):
    for j in range(bb_w):
        X_unknown[i,j]=X_un[cnt]
        cnt+=1

N_links=N_Calc(X_unknown,pix)
foreground_model,background_model,Cluster_index=Model_fit(X_un,fix_back,X_un,fix_back_idx,idx_un)
T_links=T_Calc(foreground_model,background_model,Cluster_index,X_un,idx_un)
Edges=[]
Edges.extend(T_links)
Edges.extend(N_links)
min_cut,correspondences=Graph_Cut__(Edges,fix_back_idx)
Matte=Update_Matte(min_cut,correspondences,Matte)
img2=np.empty((h,w,3))
for i in range(h):
    for j in range(w):
        if Matte[i,j]==0:
            img2[i,j]=[0,0,0]
        else:
            img2[i,j]=img[i,j]
```

```
cv2.imwrite('./V7/'+im_name + '_out.jpg',img2)
E=[]
E.append(min_cut.value)
for iteration in range(3):
    Energy,Matte=Grab_Cut_ITER(Matte,X_un,fix_back_idx)
    img2=np.empty((h,w,3))
    for i in range(h):
        for j in range(w):
            if Matte[i,j]==0:
                img2[i,j]=[0,0,0]
            else:
                img2[i,j]=img[i,j]
    cv2.imwrite('./V7/'+im_name + '_out_.jpg',img2)
    E.append(Energy)
```

Step2: For the obtained pixel set of Background and Foreground sets obtain the Gaussian Mixture Model for each of them.

Step3: Obtain the cluster index and model parameters for each unknown pixel with respect to tentative foreground and fixed background model.

```
def Model_fit(X_fore,fix_back,X_un,fix_back_idx,idx_un):
    g_f = mixture.GaussianMixture(n_components=num_cluster).fit(X_fore)
    g_b = mixture.GaussianMixture(n_components=num_cluster).fit(fix_back)
    g_b_CI=g_b.fit_predict(X_un)
    g_f_CI=g_f.fit_predict(X_un)
    #finding component index
    Comp_index=np.empty((h,w,2))#0-->Comp_Index for Foreground or unknown model 1-->for background mode
    idx_u=np.asarray(idx_un)
    cnt=0
    for i,j in zip(idx_u[:,0],idx_u[:,1]):
        Comp_index[i,j,0]=g_f_CI[cnt]
        Comp_index[i,j,1]=g_b_CI[cnt]
        cnt+=1
    return(g_f,g_b,Comp_index)
```

Step 4: Calculate the Terminal weights for each pixel in unknown region with respect to fixed background GMM and new foreground GMM. The weights for each parameter is calculated using the model parameters of the cluster in which it belongs in each model using the following equation-

$$D(\alpha_n, k_n, \underline{\theta}, z_n) = -\log \pi(\alpha_n, k_n) + \frac{1}{2}\log\det\Sigma(\alpha_n, k_n) + \frac{1}{2}[z_n - \mu(\alpha_n, k_n)]^{\top}\Sigma(\alpha_n, k_n)^{-1}[z_n - \mu(\alpha_n, k_n)].$$

Here α_n=Matte Value, k_n=Cluster Indices, $\underline{\theta}$=Model Parameters, z_n=Pixel Values

```
#This is calculated in every iteration
def D_calc(X,model,j):     #Function for calculating th weigght for a given pixel basd on tee GMM Clustr it belongs to
    j=int(j)
    Gauss=model.weights_[j]*(gaussian(X, model.means_[j], model.covariances_[j]))
    if Gauss==0:
        Gauss=0.0000000000000001
    D=-np.log(Gauss)
    return D
def gaussian(X, mu, cov):   #Function to calculate the Gauassian Probability for a given pixel
    n = X.shape[0]
    diff = (X - mu).T
    g=1 / ((2 * np.pi) ** (n / 2) * np.linalg.det(cov) ** 0.5) * np.exp(-0.5 * np.dot(np.dot(diff.T, np.linalg.inv(cov)), diff))
    if g==0:
        g=0.0000000000000001
    return g
def T_Calc(g_f,g_b,comp_id,X_un,idx_un): #The main function to calculate the Terminal Edges
    #Allocation of t links
    #T-Edges
    T=[]
    idx_u=np.asarray(idx_un)

    cnt=0
    for i,j in zip(idx_u[:,0],idx_u[:,1]):
        D_fore=D_calc(X_un[cnt],g_f,comp_id[i,j,0])
        D_back=D_calc(X_un[cnt],g_b,comp_id[i,j,1])
        #print(D_fore,D_back)
        T.append(((i,j),Back,D_back))
        T.append((Fore,(i,j),D_fore))
        cnt+=1
        #print(cnt)
    return(T)    #Returns all T edges with their weights
```

Step5: Calculate the neighbor links for each element in the unknown region. The neighborhood can be 4 or 8.For each neighbor of each pixel calculate the weight by the following formula-

$$V(\underline{\alpha},\mathbf{z}) = \gamma \sum_{(m,n)\in\mathbf{C}} dis(m,n)^{-1}\,[\alpha_n \neq \alpha_m]\exp -\beta(z_m - z_n)^2,$$

Where $dis(m,n)^{-1}$ is Euclidean Distance, $(Z_m\text{-}Z_n)^2$ is distance between two neighbor in Color Space.

and $\beta = \left(2\left\langle (z_m - z_n)^2 \right\rangle\right)^{-1}$ i.e.

Beta=1/(2*(sum of Euclidian distance between all neighbouring edges in colour space)/(number of edges).

```
#This Cell deals with all the calculations required for the Neighbourhood edges
#This is called only once at the time of intializing the graph
def neighbours(x,y,num,size):  #returns array index of pixels in neighbourhood of pixel location(x,y)
    neighbour=[]
    val_x=[x-1,x,x+1]
    val_y=[y-1,y,y+1]
    pos_x=[]
    pos_y=[]
    for i in val_x:
        if i<0 or i==size[0]:
            pass
        else:
            pos_x.append(i)
    for i in val_y:
        if i<0 or i==size[1]:
            pass
        else:
            pos_y.append(i)
    if num==4:
        for i in pos_x:
            for j in pos_y:
                if (i==x and j==y) or (i!=x and j!=y):
                    pass
                else:
                    neighbour.append((i,j))
    if num==8:
```

```
    if num==8:
        for i in pos_x:
            for j in pos_y:
                if i==x and j==y:
                    pass
                else:
                    neighbour.append((i,j))
    return(neighbour)


def Dist_Dist_C(N,img):   #Calculates Parameter for N-link Calculation
    a=N[0]
    b=N[1]
    num=N[2]
    Dist= (N[0][0]-N[1][0])**2 + (N[0][1]-N[1][1])**2    # Dist= Euclidean in pixel location space
                                                         # Beta=1/(2*dist_C/number of edges)
                                                         # Dist_C= Euclidean in pixel color space

    Dist_C=np.linalg.norm(img[N[0][0],N[0][1]]-img[N[1][0],N[1][1]])
    return Dist,Dist_C

def N_Calc(X_unknown,pix):     #Main function for N-link Calculation:Finding neighbours
    N_edges=[]
    #print(X_unknown.shape)
    for i in range(X_unknown.shape[0]):   #Loop for finding neighbourhood for each pixel
        for j in range(X_unknown.shape[1]):
            neighbour=neighbours(i,j,neighbourhood,X_unknown.shape)
            for n in range(len(neighbour)):
                mapped_x=i+pix[1]
                mapped_y=j+pix[0]
                N_edges.append(((mapped_x,mapped_y),(neighbour[n][0]+pix[1],neighbour[n][1]+pix[0]),len(neighbour))) #
```

```
  weight_N=np.empty(len(N_edges))
  #print(img.shape)
  sum_Dist_C=0
  for i in range(len(N_edges)):
      sum_Dist_C+=(img[N_edges[i][0]]-img[N_edges[i][1]])**2
  Beta=(np.linalg.norm(sum_Dist_C))/len(N_edges)
  Beta=1/(2*Beta)
  #print(Beta)
  for i in range(len(N_edges)):
          Dist,Dist_C=Dist_Dist_C(N_edges[i],img)
          temp=-Beta*(Dist_C**2)
          temp=gamma*math.exp(temp)
          weight_N[i]=temp/Dist            #N link edge weight allocatio
  N=[]#list of edges and weights together
  for i in range(len(N_edges)):
      N.append((N_edges[i][0],N_edges[i][1],weight_N[i]))
  L=[]
  for i in range(len(N_edges)):
      L.append(weight_N[i])

  return(N) #Returns all the nighbourhood edges possible for each pixel along with their weights
```

Step 6: Create a graph of unknown pixels as vertices and to each pixel assign 3 types of edges: 1)Edge with the Foreground Node; 2)Edge with the Background Node; 3)Edge with the neighbors.

Step7: On the created graph apply the st mincut to get the minimum Energy for the Cut

```
#This function deals with graph creation and performing min-cut by taking All the possible Edges and their weights
#It gives the Min Cut obtained as the output
def Graph_Cut__(Edges,fix_back_idx):
    Vertices=[]
#     idx_u=np.asarray(idx_un)
#     x=list(idx_u[:,0])
#     y=list(idx_u[:,1])

#     for i,j in zip(x,y):
#         Vertices.append((i,j))
    for i in range(h):
            for j in range(w):
                if(i,j) in fix_back_idx:
                    pass
                else:
                    Vertices.append((i,j))
    Vertices.append(Fore)
    Vertices.append(Back)
    g=Graph()
    for i in range(len(Vertices)):
        g.add_vertex(name=Vertices[i])
    #print(g,len(idx_un))
    V_dict={}
    vs = VertexSeq(g)
    for v in g.vs:
        V_dict[v["name"]]=v.index


    g=g.TupleList(Edges, directed=False, vertex_name_attr="name", edge_attrs="weight")
    es = EdgeSeq(g)
    #print(len(es))
```

```
  a=V_dict[(-1,-1)]
  b=V_dict[(h,w)]
  #print(a,b)
  min_cut=g.st_mincut(a,b,capacity="weight")
  #print(min_cut)
  #print(len(min_cut.partition[0]),len(min_cut.partition[1]),len(Vertices))
  #pos1=min_cut.partition[0].index(a)
  #pos2=min_cut.partition[1].index(b)
  #print(pos1,pos2)
  #Energy=min_cut.value
  #print(Energy)
  iV_dict = {v: k for k, v in V_dict.items()}
  return(min_cut,iV_dict)
```

Step 8: Update the Matte values i.e. the segmentation values for each pixel based on the results obtained from min-cut.

```
#This function updates the Matte which contains the segmentation for the given image, after every iteration
def Update_Matte(min_cut,iV_dict,Matte):
    for i in min_cut.partition[0]:
        x_loc=iV_dict[i][0]
        y_loc=iV_dict[i][1]
        if(x_loc!=-1 and Matte[x_loc,y_loc]!=1):
            Matte[x_loc,y_loc]=1
    for i in min_cut.partition[1]:
        x_loc=iV_dict[i][0]
        y_loc=iV_dict[i][1]
        if(x_loc!=h and Matte[x_loc,y_loc]!=0):
            Matte[x_loc,y_loc]=0
    #print(Matte.shape)
    return Matte
```

Step9: Repeat the above steps until convergence form the set of unknown pixels.

```
#This is the main calling function for every iteration
def Grab_Cut_ITER(Matte,X_un,idx_un):
    X_fore=[]
    idx_fore=[]
    for i in range(h):
        for j in range(w):
            if Matte[i,j]==1:
                X_fore.append(img[i,j])
                idx_fore.append((i,j))

    foreground_model,background_model,Cluster_index=Model_fit(X_fore,fix_back,X_un,fix_back_idx,idx_un)
    T_links=T_Calc(foreground_model,background_model,Cluster_index,X_un,idx_un)
    Edges=[]
    Edges.extend(T_links)
    Edges.extend(N_links)
    min_cut,correspondences=Graph_Cut__(Edges,idx_un)
    Matte=Update_Matte(min_cut,correspondences,Matte)
    return(min_cut.value,Matte)
```

Analysis of Hyper-parameters:

1) **Number of Iterations and size of bounding box:** Major convergence occurs in first and second iteration in general but if the bounding box is loose and foreground is in small portion of the bounding box then it takes larger number of iteration .In general 2 iterations were used.
2) **Neighborhood:** The results are better with 8 neighbourhood than 4 neighbourhood as it considers the smoothness factor more. In general,8 neighbourhood is used when there is lot of pixel value variation in the image.
3) **Choice of Beta and Gamma:** When the constant Beta =0, the smoothness term is simply the well-known Ising prior, encouraging smoothness everywhere, to a degree determined by the constant gamma. However, it is far more effective to set Beta > 0 as this relaxes the tendency to

smoothness in regions of high contrast. Gamma basically decides how much importance is being given to the neighborhood links. In the experiment if the Beta was set to be zero then whole bounding box area was considered to be foreground irrespective of everything if gamma was>20.So Beta is chosen as Beta=1/(2*(sum of Euclidian distance between all neighbouring edges in colour space)/(number of edges). This choice of Beta ensures that the exponential term in N_weight calculation switches appropriately between high and low contrast. For this Beta the perfect Gamma was determined to be 50 by the authors.

4) **Number of GMM Components and its effect on Hardness of segmentation:** If the number of components is taken to be large when the number of colours in the image is lesser lead to assignment of multiple components to different shades of same color leading to hard segmentation. So we attain soft segmentation when number of components is comparable to number of color present in the image. In the paper it was taken to be 5.

Results:

Results can be found at following link: https://iiitaphyd-my.sharepoint.com/:f:/g/personal/apoorva_srivastava_research_iiit_ac_in/EoY9VNEiaFdJlkFpi_A2ziQBHvcb8ytQQASvIFGyUucISg?e=Fg18AY

Few good and bad results are shown below-

Input:

Output: