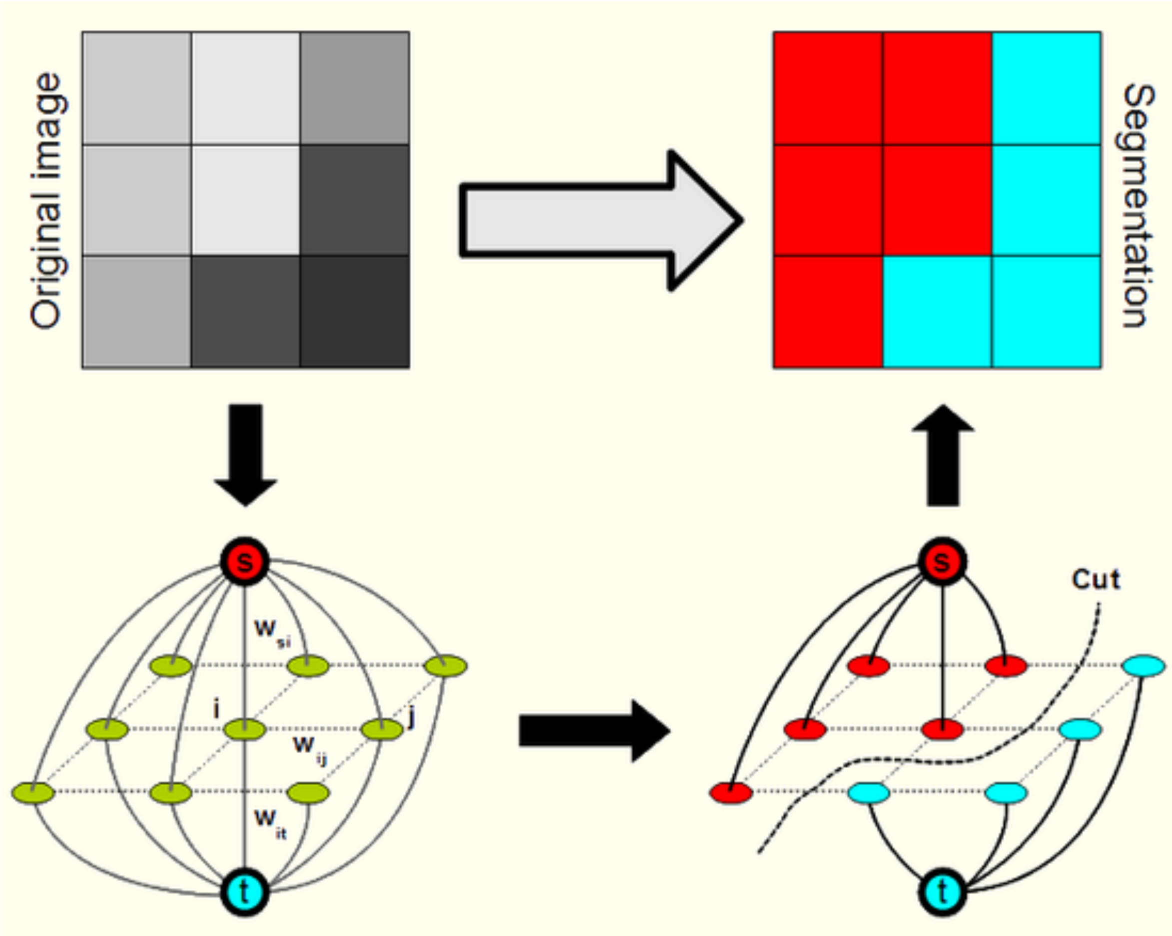
Assignment 3 - Image Segmentation using MRFs

GrabCut

TA : Rohan, Prajwal
Release date: 05/03/21
Submission date : 16/03/21

For this assignment you will implement the GrabCut method mentioned in this paper. It is essentially an iterative version of GraphCut as shown in the figure below.



The code below takes an input image and follows these steps:

- It requires a bounding box to be drawn by the user to roughly segment out the foreground pixels
- It runs an initial min-cut optimization using the provided annotation
- The result of this optimization gives an initial segmentation
- To further refine this segmentation, the user provides two kinds of strokes to aid the optimization
  - strokes on the background pixels

        - strokes on the foreground pixels
- The algorithm now utilizes this to refine the original segmentation

You are allowed to use standard GMM libraries for the implementation. For usage of other libraries, please contact the TAs.

You can view this video to get a better idea of the steps involved.

Image segmentation is one exciting application of MRFs. You can further read about other applications of MRFs for Computer Vision here.

Useful Links

- https://courses.engr.illinois.edu/cs543/sp2011/lectures/Lecture%2012%20-%20MRFs%20and%20Graph%20Cut%20Segmentation%20-%20Vision_Spring2011.pdf

In [1]:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
```

In [2]:

```
class EventHandler:
    """
    Class for handling user input during segmentation iterations
    """

    def __init__(self, flags, img, _mask, colors):

        self.FLAGS = flags
        self.ix = -1
        self.iy = -1
        self.img = img
        self.img2 = self.img.copy()
        self._mask = _mask
        self.COLORS = colors

    @property
    def image(self):
        return self.img

    @image.setter
    def image(self, img):
        self.img = img

    @property
    def mask(self):
        return self._mask

    @mask.setter
    def mask(self, _mask):
        self._mask = _mask

    @property
    def flags(self):
        return self.FLAGS
```

```
    @flags.setter
    def flags(self, flags):
        self.FLAGS = flags

    def handler(self, event, x, y, flags, param):

        # Draw the rectangle first
        if event == cv2.EVENT_RBUTTONDOWN:

            self.FLAGS['DRAW_RECT'] = True
            self.ix, self.iy = x,y

        elif event == cv2.EVENT_MOUSEMOVE:

            if self.FLAGS['DRAW_RECT'] == True:

                self.img = self.img2.copy()
                cv2.rectangle(self.img, (self.ix, self.iy), (x, y), self.CO
#                 cv2.rectangle(self._mask, (self.ix, self.iy), (x, y), sel
                self.FLAGS['RECT'] = (min(self.ix, x), min(self.iy, y), abs
                self.FLAGS['rect_or_mask'] = 0

        elif event == cv2.EVENT_RBUTTONUP:

            self.FLAGS['DRAW_RECT'] = False
            self.FLAGS['rect_over'] = True
            cv2.rectangle(self.img, (self.ix, self.iy), (x, y), self.COLORS
            cv2.rectangle(self._mask, (self.ix, self.iy), (x, y), 127, -1)
            self.FLAGS['RECT'] = (min(self.ix, x), min(self.iy, y), abs(sel
            self.FLAGS['rect_or_mask'] = 0

        # Draw strokes for refinement

        if event == cv2.EVENT_LBUTTONDOWN:
            if self.FLAGS['rect_over'] == False:
                print('Draw the rectangle first.')
            else:
                self.FLAGS['DRAW_STROKE'] = True
                cv2.circle(self.img, (x,y), 6, self.FLAGS['value']['color']
                cv2.circle(self._mask, (x,y), 6, self.FLAGS['value']['val']

        elif event == cv2.EVENT_MOUSEMOVE:
            if self.FLAGS['DRAW_STROKE'] == True:
                cv2.circle(self.img, (x, y), 6, self.FLAGS['value']['color'
                cv2.circle(self._mask, (x, y), 6, self.FLAGS['value']['val'

        elif event == cv2.EVENT_LBUTTONUP:
            if self.FLAGS['DRAW_STROKE'] == True:
                self.FLAGS['DRAW_STROKE'] = False
                cv2.circle(self.img, (x, y), 3, self.FLAGS['value']['color'
                cv2.circle(self._mask, (x, y), 3, self.FLAGS['value']['val'
```

In [6]:

```
from sklearn.mixture import GaussianMixture

class GMM(object):

    def __init__(self, samples, K):
```

```
        assert len(samples) > K, "No samples or not enough samples found"
        assert K > 0, "KMeans must have atleast 1 component"

        self.K = K
        self.samples = samples.reshape(-1,3)
        self.weights = np.array([1/self.K for i in range(self.K)])

        self.model = GaussianMixture(
                                    n_components=self.K, covariance_type="f
                                    weights_init=self.weights, n_init = 2
        ).fit(samples)

        self.means = self.model.means_
        self.covs = self.model.covariances_

        self.predictions = self.model.predict(samples)

    def update_model(self, samples):

        self.samples = samples.reshape(-1,3)
        self.predictions = self.model.predict(self.samples)

        for i in range(self.K):

            F = self.samples[self.predictions == i]

            if len(F) < 2:
                self.weights[i] = 0
                for j in range(self.K):
                    if j == i:
                        continue
                    if self.weights[j] > 1e-2:
                        self.weights[j] -= 1e-2
                        self.weights[i] += 1e-2
                        break
                continue

            ni = len(F)
            self.weights[i] = ni/len(self.samples)
            if self.weights[i] == 0.:
                for j in range(self.K):
                    if j == i:
                        continue
                    if self.weights[j] > 1e-2:
                        self.weights[j] -= 1e-2
                        self.weights[i] += 1e-2
                        break

            self.means[i] = F.mean(axis=0)
            self.covs[i] = np.cov(F.T).T

        for i in range(self.K):
            while np.linalg.det(self.covs[i]) == 0:
                self.covs[i][0,0] += 0.001
                self.covs[i][1,1] += 0.001
                self.covs[i][2,2] += 0.001

        try:
            self.model = GaussianMixture(
                                        n_components=self.K, covariance_typ
                                        means_init=self.means, precisions_i
```

```
                                             weights_init=self.weights, n_init =
            ).fit(samples)
        except:
            self.model = GaussianMixture(
                                             n_components=self.K, covariance_typ
                                             n_init = 1
            ).fit(samples)

            self.predictions = self.model.predict(self.samples)

            for i in range(self.K):
                F = self.samples[self.predictions == i]

                if len(F) < 2:
                    self.weights[i] = 0
                    continue

                ni = len(F)
                self.weights[i] = ni/len(self.samples)

        self.means = self.model.means_
        self.covs = self.model.covariances_

        self.predictions = self.model.predict(self.samples)

    def prob(self, i, samples):

        samples = samples.reshape(-1,3)

        assert i < self.K, "Exceeded number of components"
        mu, cov = self.means[i], self.covs[i]

        mu = mu.reshape(-1,3)

        fac = []
        for j in samples:
            fac += [(j - mu) @ np.linalg.pinv(cov) @ (j - mu).T]
        fac = np.array(fac).reshape(-1,1)

        return (1/np.sqrt(2*np.linalg.det(cov))) * np.exp(-0.5*fac)

    def calculate_U(self, samples):

        samples = samples.reshape(-1,3)
        return -self.model.score_samples(samples).reshape(-1,1)
```

In [13]:

```
import igraph as ig

class GrabCut(object):

    def __init__(self, img, mask, K=5, n_iters=5, gamma=50):

        print("Initializing...")

        self.h = img.shape[0]
        self.w = img.shape[1]

        img = img.astype('float32')
        self.original_img = img.copy()
```

```
        self.img = img.reshape(-1,3)
        self.mask = mask.reshape(-1)

        # Initializing parameters
        self.K = K
        self.n_iters = n_iters
        self.gamma = gamma
        self.lamda = 1
        self.calc_beta()

        # Initializing Graph
        self.s = self.h*self.w
        self.t = self.h*self.w + 1
        self.calculate_V()

        # Initializing GMMs
        self.gm_fg = GMM(self.img[(self.mask == 1) + (self.mask == 2)], sel
        self.gm_bg = GMM(self.img[self.mask == 0], self.K)

        # Finding trimap pixels
        self.probable_fg = self.img[self.mask == 2]
        self.fg = self.img[self.mask == 1]
        self.bg = self.img[self.mask == 0]

        # Getting indices of trimap pixels
        self.probable_indices = (np.where(self.mask == 2))[0]
        self.fg_indices = (np.where(self.mask == 1))[0]
        self.bg_indices = (np.where(self.mask == 0))[0]

        # Initializing alphas
        self.alphas = np.zeros(self.h*self.w)
        self.alphas[(self.mask == 1) | (self.mask == 2)] = 1

    def calc_beta(self):

        # Gets differences between pixels in the required directions
        self.l = (self.original_img[:, 1:] - self.original_img[:, :-1])**2
        self.u = (self.original_img[1:, :] - self.original_img[:-1, :])**2
        self.ur = (self.original_img[1:, :-1] - self.original_img[:-1, 1:])
        self.ul = (self.original_img[1:, 1:] - self.original_img[:-1, :-1])

        # Uses the formula for beta
        self.beta = np.sum(self.l) + np.sum(self.u) + \
                    np.sum(self.ur) + np.sum(self.ul)
        self.beta = (4*self.h*self.w - 3*(self.w + self.h) + 2)/(2*self.bet

    def calculate_V(self):

        # Calculates the V term to add to the graph's edges capacity
        cap_l = self.gamma*np.exp(-self.beta*np.sum(self.l, axis=2))
        cap_u = self.gamma*np.exp(-self.beta*np.sum(self.u, axis=2))
        cap_ul = self.gamma*np.exp(-self.beta*np.sum(self.ul, axis=2))
        cap_ur = self.gamma*np.exp(-self.beta*np.sum(self.ur, axis=2))

        # Dividing by distance for diagonal terms
        cap_ul/=np.sqrt(2)
        cap_ur/=np.sqrt(2)

        # Initializing variables
        self.edges_V = []
        self.capacity_V = []
```

```python
        l_edges = []
        u_edges = []
        ul_edges = []
        ur_edges = []

        # Adds the edges in the required directions
        for i in range(self.h):
            for j in range(self.w):
                curr = i*self.w+j
                if i:
                    u_edges += [((i-1)*self.w+j, curr)]
                if j:
                    l_edges += [(i*self.w+j-1, curr)]
                if i and j:
                    ul_edges += [((i-1)*self.w+j-1, curr)]
                if i and j+1 < self.w:
                    ur_edges += [((i-1)*self.w+j+1, curr)]

        # Updates edges and capacities
        self.edges_V = l_edges + u_edges + ul_edges + ur_edges
        self.capacity_V += cap_l.ravel().tolist() + \
                            cap_u.ravel().tolist() + \
                            cap_ul.ravel().tolist() + \
                            cap_ur.ravel().tolist()

    def reclassify_img(self):

        self.fg = self.img[self.alphas == 1]
        self.bg = self.img[self.alphas == 0]

    def create_graph(self):

        self.graph = ig.Graph(self.h * self.w + 2)

        edges = []
        self.capacity = []

        # getting already created n-links
        edges = self.edges_V.copy()
        self.capacity = self.capacity_V.copy()

        # creating t-links
        U_pr_bg = self.gm_bg.calculate_U(self.probable_fg).reshape(-1)
        U_pr_fg = self.gm_fg.calculate_U(self.probable_fg).reshape(-1)

        for u in self.probable_indices:
            edges += [(self.s, u)]
        for u in self.probable_indices:
            edges += [(self.t, u)]

        self.capacity += (self.lamda*U_pr_bg.ravel()).tolist()
        self.capacity += (self.lamda*U_pr_fg.ravel()).tolist()

        for i in self.fg_indices:
            edges += [(self.s, i), (self.t, i)]
            self.capacity += [20*self.gamma, 0]

        for i in self.bg_indices:
            edges += [(self.s, i), (self.t, i)]
            self.capacity += [0, 20*self.gamma]
```

```
        self.graph.add_edges(edges)

    def find_mincut(self):

        mincut = self.graph.st_mincut(self.s, self.t, self.capacity)

        self.alphas = np.zeros((self.h*self.w + 2))
        self.alphas[np.array(mincut.partition[0])] = 1
        self.alphas = self.alphas[:-2].reshape(self.h, self.w)

        img2 = self.img.copy().reshape(self.h, self.w, 3)
        img2[self.alphas == 0] = 0
#         plt.imshow(img2[:,:,::-1])
#         plt.show()

        self.alphas = self.alphas.reshape(self.h*self.w)

    def run_grabcut(self):

        for i in range(self.n_iters):

            print(f"\nIteration {i+1}")
            self.reclassify_img()

            print("Updating model...")
            self.gm_fg.update_model(self.fg)
            self.gm_bg.update_model(self.bg)

            print("Creating graph..")
            self.create_graph()

            print("Finding mincut..")
            self.find_mincut()

        # Returning the final mask
        self.alphas[self.alphas == 1] = 255
        self.alphas = self.alphas.reshape(self.h, self.w)
        return self.alphas.astype('uint8')
```

In [47]:

```
def run(filename: str):
    """
    Main loop that implements GrabCut.

    Input
    -----
    filename (str) : Path to image
    """

    COLORS = {
    'BLACK' : [0,0,0],
    'RED'   : [0, 0, 255],
    'GREEN' : [0, 255, 0],
    'BLUE'  : [255, 0, 0],
    'WHITE' : [255,255,255]
    }

    DRAW_BG = {'color' : COLORS['BLACK'], 'val' : 0}
    DRAW_FG = {'color' : COLORS['WHITE'], 'val' : 1}
```

```
    FLAGS = {
        'RECT' : (0, 0, 1, 1),
        'DRAW_STROKE': False,         # flag for drawing strokes
        'DRAW_RECT' : False,          # flag for drawing rectangle
        'rect_over' : False,          # flag to check if rectangle is  draw
        'rect_or_mask' : -1,          # flag for selecting rectangle or str
        'value' : DRAW_FG,            # drawing strokes initialized to mark
    }

    img = cv2.imread(filename)
    h,w = img.shape[:2]
#     h/=2
#     w/=2
#     img = cv2.resize(img, (int(w), int(h)))
    img2 = img.copy()
    mask = np.zeros(img.shape[:2], dtype = np.uint8) # mask is a binary arr
                                                     #
    output = np.zeros(img.shape, np.uint8)           # output image to be s

    # Input and segmentation windows
    cv2.namedWindow('Input Image')
    cv2.namedWindow('Segmented output')

    EventObj = EventHandler(FLAGS, img, mask, COLORS)
    cv2.setMouseCallback('Input Image', EventObj.handler)
    cv2.moveWindow('Input Image', img.shape[1] + 10, 90)

    while(1):

        img = EventObj.image
        mask = EventObj.mask
        FLAGS = EventObj.flags
        cv2.imshow('Segmented image', output)
        cv2.imshow('Input Image', img)

        k = cv2.waitKey(1)

        # key bindings
        if k == 27:
            # esc to exit
            break

        elif k == ord('0'):
            # Strokes for background
            FLAGS['value'] = DRAW_BG

        elif k == ord('1'):
            # FG drawing
            FLAGS['value'] = DRAW_FG

        elif k == ord('r'):
            # reset everything
            FLAGS['RECT'] = (0, 0, 1, 1)
            FLAGS['DRAW_STROKE'] = False
            FLAGS['DRAW_RECT'] = False
            FLAGS['rect_or_mask'] = -1
            FLAGS['rect_over'] = False
            FLAGS['value'] = DRAW_FG
            img = img2.copy()
            mask = np.zeros(img.shape[:2], dtype = np.uint8)
```

```python
            EventObj.image = img
            EventObj.mask = mask
            output = np.zeros(img.shape, np.uint8)

        elif k == 13:
            # Press carriage return to initiate segmentation

            #-------------------------------------------------#
            # Implement GrabCut here.                         #
            # Function should return a mask which can be used #
            # to segment the original image as shown on L90   #
            #-------------------------------------------------#

            EventObj.flags = FLAGS
            mask = EventObj.mask
            mask[mask==255] = 1
            mask[mask==127] = 2

            if np.sum(mask) == 0:
                print("Please reselect points")
                continue

            grabcut_obj = GrabCut(img2, mask, 5, 3)
            mask2 = grabcut_obj.run_grabcut()
            output = cv2.bitwise_and(img2, img2, mask=mask2)

#             To reuse resultant mask
#             mask = mask2
#             mask[mask==255] = 1

        elif k == ord('q'):
            break

    # uncomment to save outputs (will overwrite existing)
#     plt.imsave(f"../images/outputs/study_output_{filename.split('/')[2]}
#     plt.imsave(f"../images/outputs/study_input_{filename.split('/')[2]}",
```

In [50]:

```python
if __name__ == '__main__':
    filename = '../images/person8.jpg' # Path to image file
    run(filename)
    cv2.destroyAllWindows()
```

```
Initializing...

Iteration 1
Updating model...
Creating graph..
Finding mincut..

Iteration 2
Updating model...
Creating graph..
Finding mincut..

Iteration 3
Updating model...
Creating graph..
Finding mincut..
```

Report

Contents

Introduction

The GrabCut algorithm works robustly and gives a good output on almost all the images with the parameters mentioned in the paper, ie.

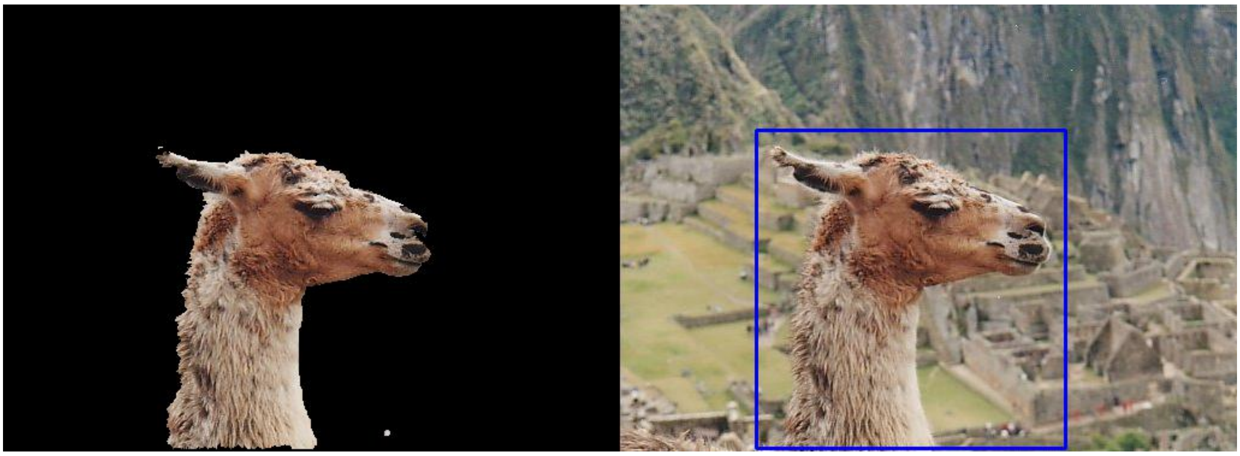
- γ = 50
- λ = 9γ
- GMM Components = 5
- Iterations = 3

Study of the Effect of Changing Parameters

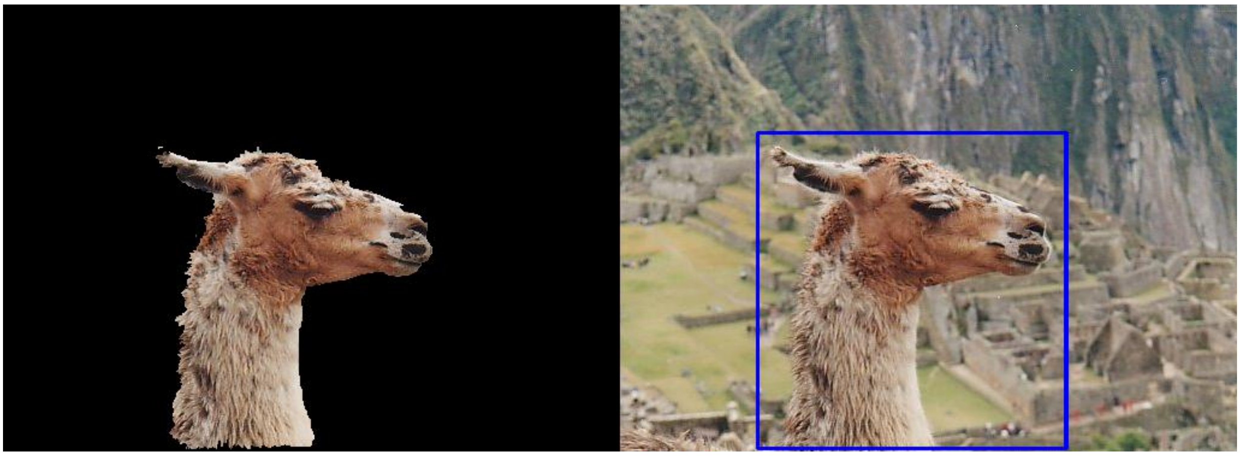
1. Varying γ

$\gamma = 2$

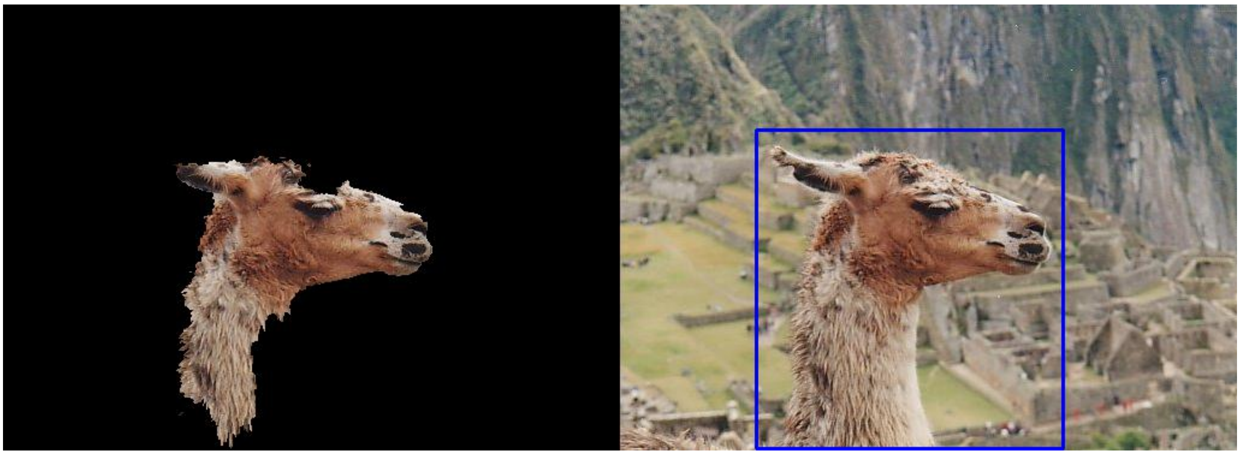
$\gamma = 20$



$\gamma = 50$



$\gamma = 200$



Inference

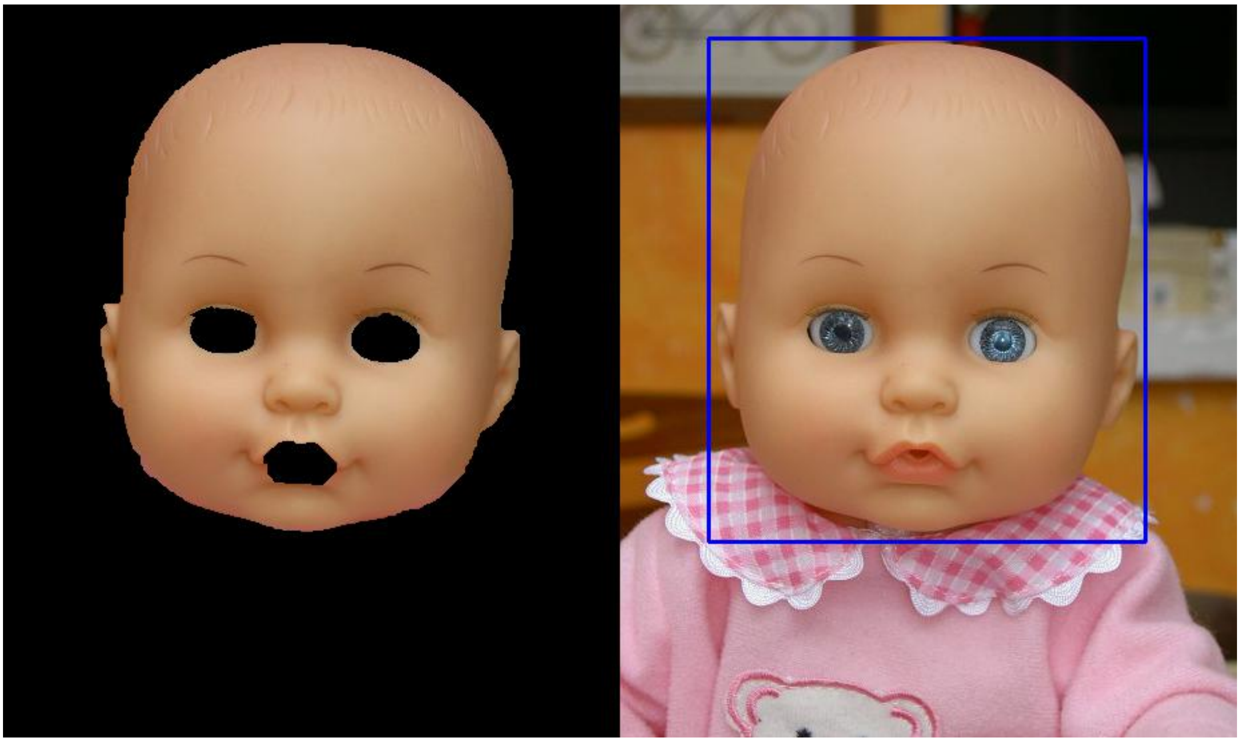
As we can see, increasing γ makes the segmentation smoother. This is because it increases the cost of removing the links between neighboring pixels, thus forcing the algorithm to assign nearby pixels the same label unless there's a big difference in the intensities of the nearby pixel.

With a lower value of γ, this inter-pixel weight is low and hence there is more "noise" in the sense that there are many areas where nearby pixels are assigned opposite labels creating 'islands' of wrongly marked foregrounds. With extremely high values of γ, this inter pixel weight is extremely high, erasing useful parts of the foreground too. At moderate values of γ (eg. 50), the inter pixel weights are perfectly balanced with respect to the unary weights, giving a good segmentation.

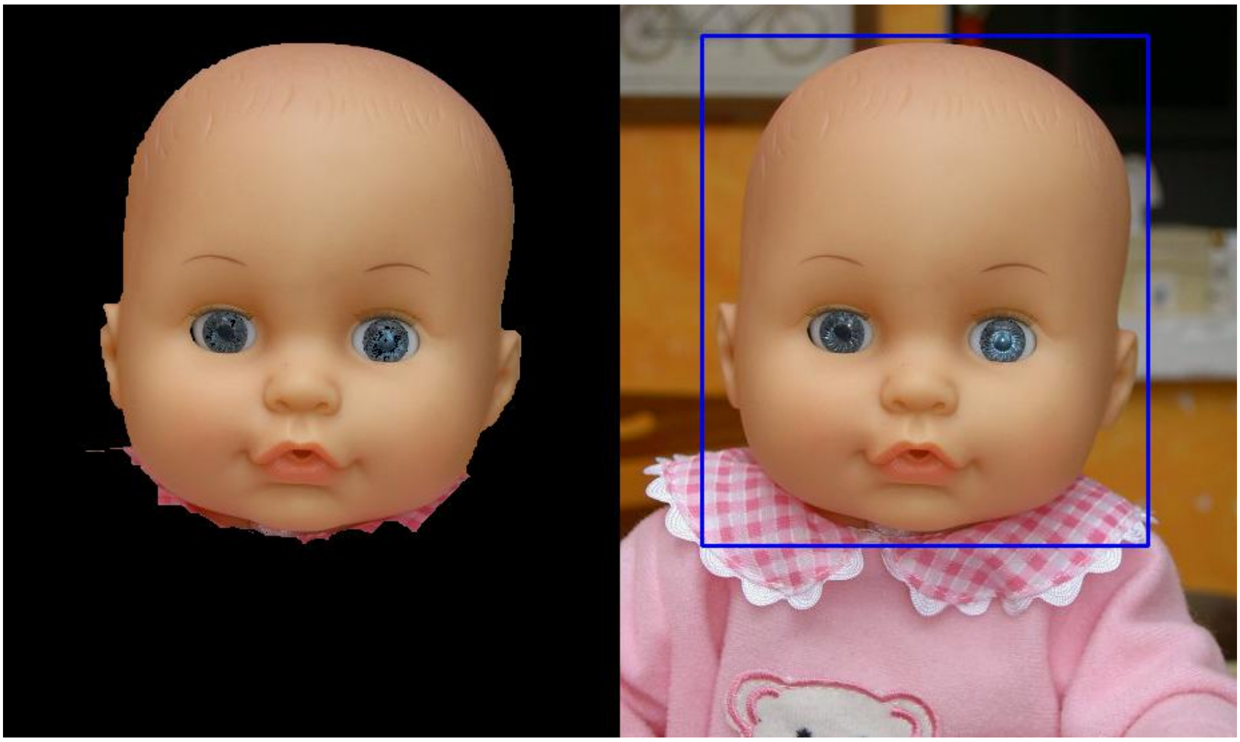
---

2. Varying No. of GMM Components

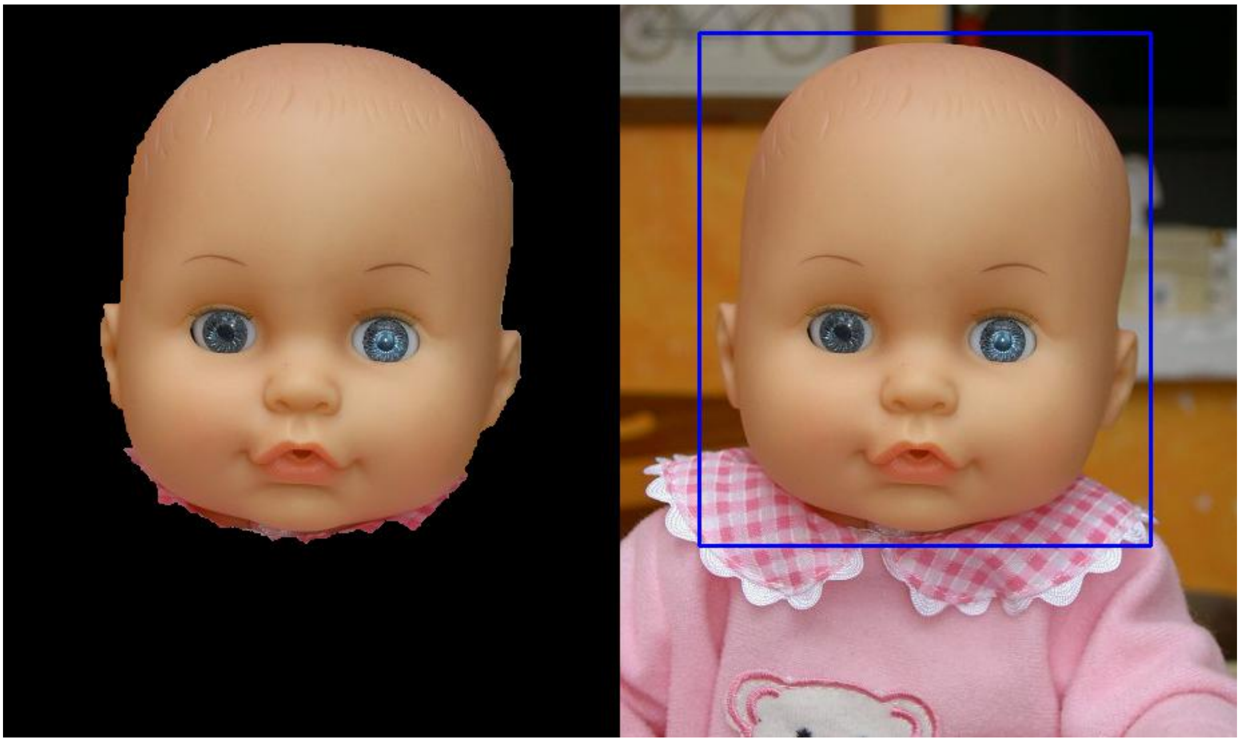
$K = 1$



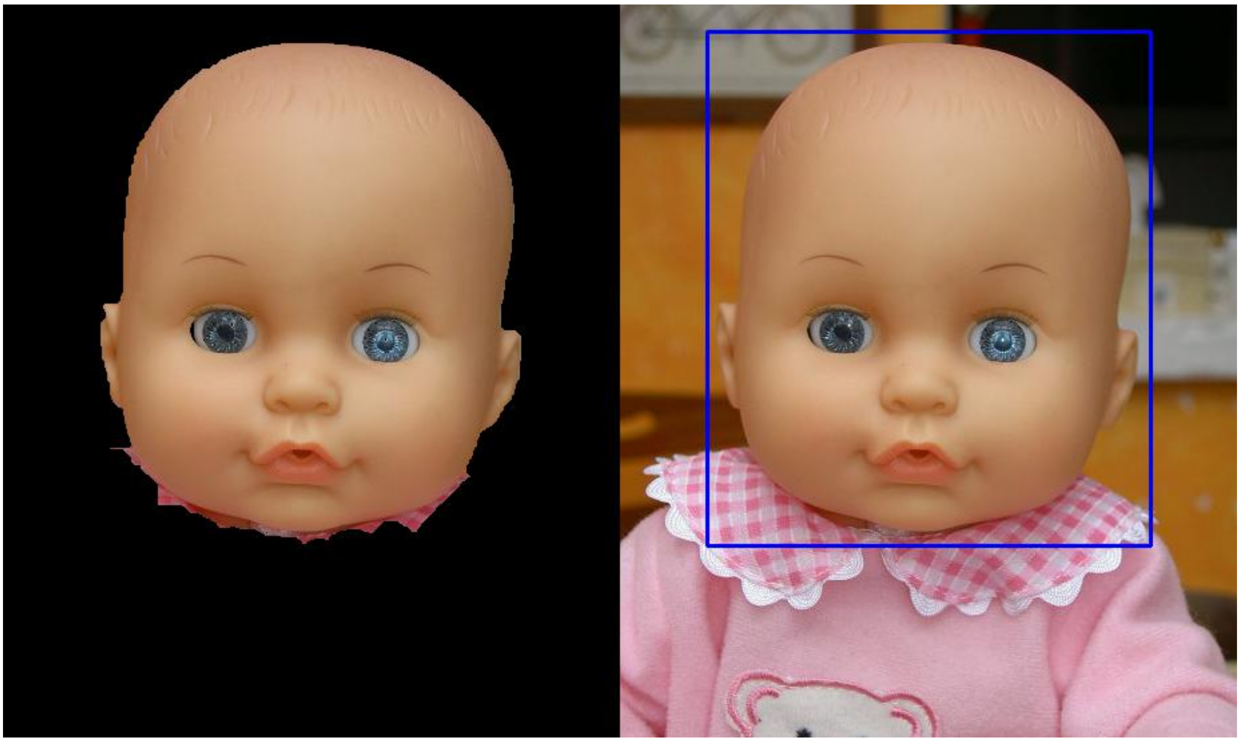
$K = 3$



$K = 5$



$K = 20$



Inference

From the above results, we see that increasing the number of components gives the algorithm greater flexibility while marking FG and BG, and hence the results are better and more detailed as we increase the number of components.

At $K = 1$, we can see that since GrabCut can choose only one mean and covariance to model the FG and BG, it may not be able to effectively capture the entire marked BG or probably FG region. Hence in this case, we can see that the eyes and mouth although being FG, are marked as BG. This demonstrates how the single component GMM is not able to capture the different colors and intensities of the eyes and mouth as it is more tuned towards the skin (as it is the most present).

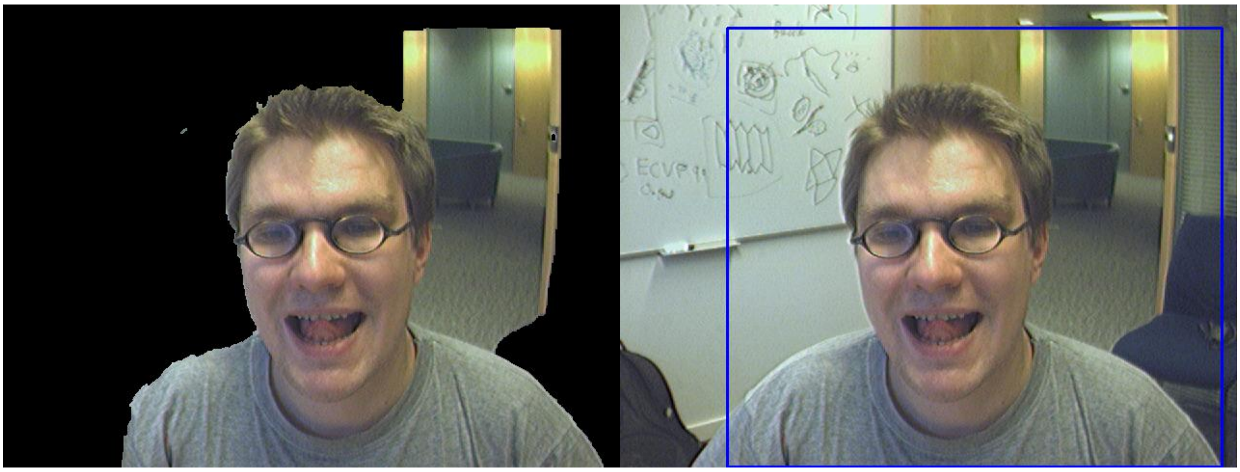
For higher K's, we consistently get good segmentation results, with the tradeoff being the time taken to run. Initializing a GMM with higher number of components takes longer than that with a lesser number of components.

---

3. Varying the Fit of the Bounding Box

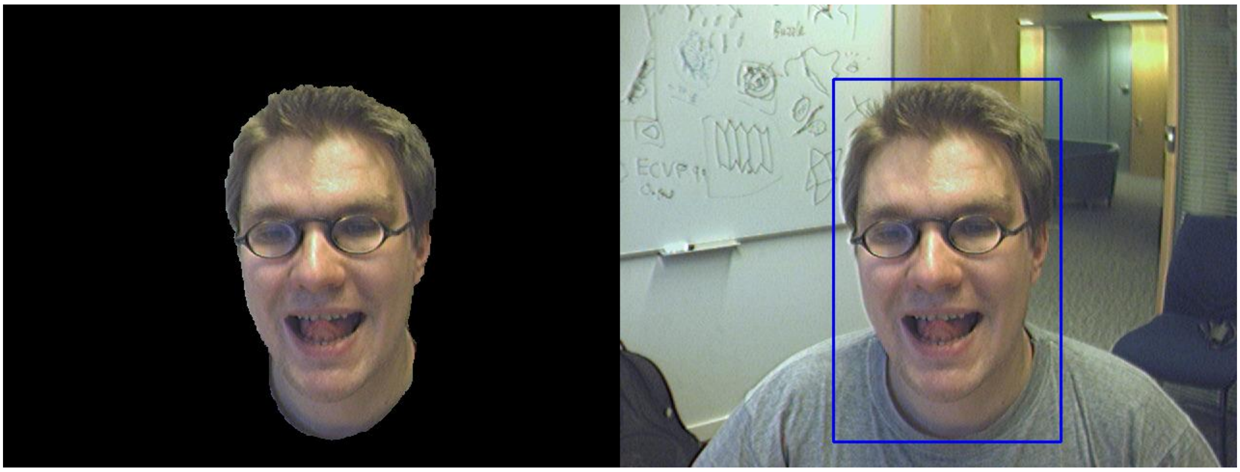
Loosest Fit

Loose Fit



Slightly Loose Fit

Tight Fit



Inference

A tight bounding box gives us the best results as it gives the background GMM model more background samples to model the background distribution. This is demonstrated above by using bounding boxes of varying fits. However, good results can be obtained using a loose bounding box but would need a lot of user editing to assist the algorithm in modelling the background properly.

Results