Report

Vanshita Bansal

25-06-2020

Contents

1 Threshold Based Segmentation **2**
- 1.1 Types of Threshold based segmentation . . . 2
- 1.2 Global Thresholding . . . 2
  - 1.2.1 Histogram based Thresholding . . . 3
  - 1.2.2 Iterative based Thresholding . . . 6
  - 1.2.3 Otsu's Thresholding . . . 6
  - 1.2.4 Maximum Correlation . . . 7
  - 1.2.5 Multithresholding . . . 8
  - 1.2.6 Clustering . . . 8
- 1.3 Local Thresholding . . . 8
- 1.4 Adaptive Thresholding . . . 9
  - 1.4.1 Chow and Kaneko approach . . . 10
  - 1.4.2 Local thresholding . . . 11

2 Region Based Segmentation **12**
- 2.1 Region Growing . . . 12
  - 2.1.1 Algorithm . . . 12
- 2.2 Region Splitting and Merging . . . 16
  - 2.2.1 Region splitting Method . . . 16
  - 2.2.2 Region Merging Method . . . 17

3 Artificial Neural Network Based Segmentation **18**
- 3.1 Mask R-CNN for Image Segmentation . . . 18
- 3.2 U-Net Architecture for Image Segmentation . . . 21

4 Comparison between all the Methods **26**

Chapter 1

Threshold Based Segmentation

It is a type of Image Segmentation, where we change the original pixel such that it is easier to analyse. It is the simplest method of all the other. Thresholding is used to split an image into smaller segments, using at least one color or gray scale value to define their boundary. The advantage of obtaining first a binary image is that it reduces the complexity of the data and simplifies the process of recognition and classification. The most common way to convert a gray level image to a binary image is to select a single threshold value (T).

Simple Thresholding

This method of segmentation applies a single fixed criterion to all pixels in the image simultaneously. The pixels are partitioned depending on their intensity value. Segment image into foreground and background. g(x, y) = 1 if f(x,y) is foreground pixel = 0 if f(x, y) is background pixel In real applications histograms are more complex, with many peaks and not clear valleys and it is not always easy to select the value of T.

1.1 Types of Threshold based segmentation

1) Global Thresholding
2) Local Thresholding
3) Adaptive Thresholding

1.2 Global Thresholding

The global threshold is applicable when the intensity distribution of objects and background pixels are sufficiently distinct. In the global threshold, a single threshold value is used in the whole image.

There are a number of global thresholding techniques such as: Otsu, optimal thresholding, histogram analysis, iterative thresholding, maximum correlation thresholding, clustering and Multithresholding.

1.2.1 Histogram based Thresholding

The histogram based technique is dependent on the success of the estimating the threshold value that separates the two homogenous region of the foreground and background of an image. If there are no distinct peaks, this becomes difficult to implement and may not give desired result.

Step 1: Importing required libraries.

```
[ ]  import sys
     import numpy as np
     import skimage.color
     import skimage.io
     from skimage.viewer import ImageViewer
     from matplotlib import pyplot as plt
```

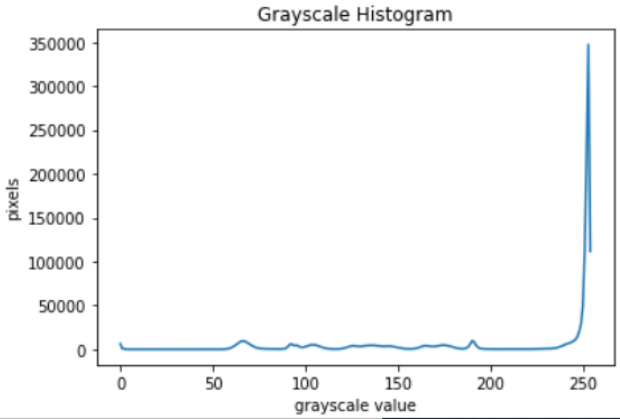
Step 2: Reading the image and display it.

```
# read image, based on command line filename argument;
# read the image as grayscale from the outset
filename='picture.PNG'
sigma=2
t=0.8 #Manually selecting threshold value
image=skimage.io.imread('picture.PNG')

from IPython.display import HTML, Image, display

Image('picture.PNG')
```

Step 3: Creating histogram by making bins.

```
histogram, bin_edges = np.histogram(image, bins=256)
```

```
# configure and draw the histogram figure
plt.figure()
plt.title("Grayscale Histogram")
plt.xlabel("grayscale value")
plt.ylabel("pixels")
# plt.ylim([0, 16000])
# plt.xlim([0.0, 1.0])  # <- named arguments do not work here

plt.plot(bin_edges[0:-1], histogram)  # <- or here
plt.show()
```



Step 4: We see a peak in the image and therefore it is easy to select a threshold value. Here, we take threshold as T=0.8.

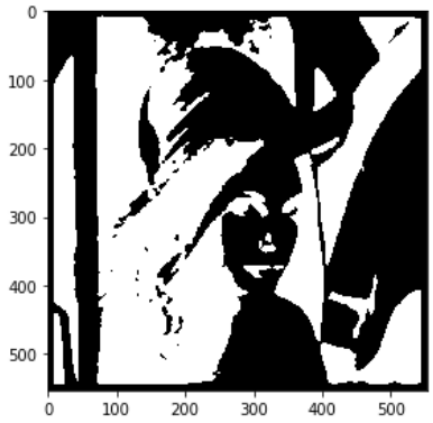
Step 5: When we blur an image, we make the color transition from one side of an edge in the image to another smooth rather than sudden. The effect is to average out rapid changes in pixel intensity. The blur, or smoothing, of an image removes “outlier” pixels that may be noise in the image.

```
[104] blur=skimage.color.rgb2gray(image)
      blur=skimage.filters.gaussian(blur,sigma)
```

Step 6: Creating mask of the image to obtain the colored image at a later stage. We use ”less than” operator for creation.

```
[115] #creating mask to obtain colored image at later stage
      mask=blur < 0.5
      sel = np.zeros_like(image)
      skimage.io.imshow(mask)
```

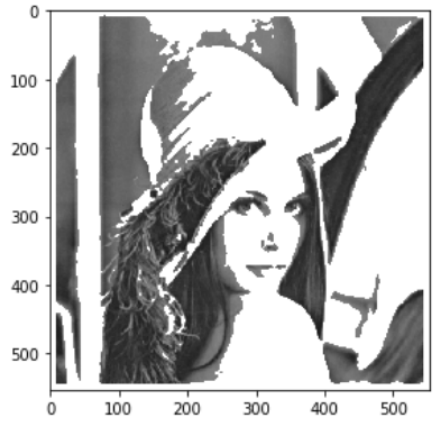
```
<matplotlib.image.AxesImage at 0x7f58eb238fd0>
```



Step 7: Finally, applying the mask to obtain the resulting image.

```
[116] sel[mask] = image[mask]
      skimage.io.imshow(sel)
      # display the result
```

```
<matplotlib.image.AxesImage at 0x7f58eb198ba8>
```



1.2.2 Iterative based Thresholding

Iterative methods give better result when the histogram doesn't clearly define valley point. The method doesn't require any pre knowledge about the image. The initial threshold value, t0, is set equal to the average brightness. Thereafter, the new threshold value tk+1for the (k + 1)-th iteration is calculated. This iterative algorithm is a special one dimensional case of K-means clustering that converges at a local minimum. But the main disadvantage is, a different initial estimate for T may give a different result.

1.2.3 Otsu's Thresholding

Otsu's method is aimed in finding the optimal value for the global threshold. It is used to overcome the drawback of iterative thresholding i.e. calculating the mean after each step. In this method identify the optimal threshold by making use of histogram of the image. The algorithm exhaustively searches for the threshold that minimizes the intra-class variance, defined as a weighted sum of variances of the two classes. The main drawback of Otsu's method of threshold selection is that it assumes that the histogram is bimodal. This method fails if two classes are of different sizes and also with variable illumination.

Step 1: Reading the image and displaying it.

Step 2: Creating histogram using bins.

Step 3: Blurring the image.

Step 4: Selection of threshold is not done manually as otsu thresholding computes it iteratively.
We can see the computed value of threshold.

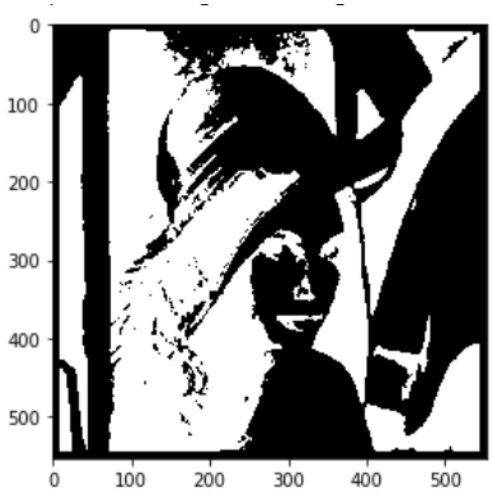
```
[125] # perform otsu thresholding
      t = skimage.filters.threshold_otsu(blur)
      mask2 = blur > t
      t #calculated using iterations
```

```
0.4741574578442087
```

Step 5: Creating mask of the image.

```
[143] mask1=blur < 0.5
      sel1 = np.zeros_like(image2)
      skimage.io.imshow(mask1)
```

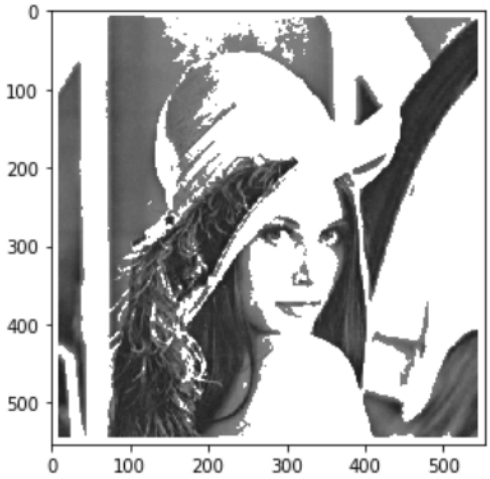
Resulting mask



Step 6: Finally, applying to get the resulting image.

```
[144] # sel1 = np.zeros_like(image2)
      sel1[mask1] = image2[mask1]
      skimage.io.imshow(sel1)
```

```
<matplotlib.image.AxesImage at 0x7f58eb100cc0>
```



1.2.4 Maximum Correlation

It is based on maximizing the correlation between the grayscale image and the threshold image. However, this optimization criterion is identical to the one used by Otsu, despite their different approach.

1.2.5 Multithresholding

The global thresholding methods, can be extended to the case of Multithresholding. It is a process that segments a gray-level image into several distinct regions. This technique determines more than one threshold for the given image and segments the image into certain brightness regions, which correspond to one background and several objects. The method works very well for objects with colored or complex backgrounds, on which bi-level thresholding fails to produce satisfactory results.

1.2.6 Clustering

The purpose of clustering is to get fast and meaningful results. This can be applied in two ways:

Fuzzy c means clustering

Fuzzy c means clustering is a technique in which pixels are grouped into n clusters with every data point belonging to every cluster to a certain degree. It is a method of clustering which allows one piece of data to belong to two or more clusters. Iterations are done to obtain a saddle point where we reach a local optimal solution. Its improvement algorithm and strategies for image segmentation can offer less iteration times to converge to global optimal solution. Its good effect of segmentation can improve accuracy and efficiency of threshold image segmentation.

K-means clustering

The image is divided into k segments using (k-1) thresholds and minimizing the total variance within each segment and maximizing the variance between each segment. It is mostly based on a simple iterative scheme for finding a locally minimal solution. This algorithm is often called the k-means algorithm. This method works well if the spreads of the distributions are approximately equal, but it does not handle well the case where the distributions have differing variances.

1.3 Local Thresholding

It is not possible to obtain a single threshold that could segment an image. The idea is to partition the image into several sub image (say m x m) and then choose a threshold for each sub image. It can be effectively used when the gradient effect is small with respect to sub images. The threshold value T depends on gray levels of f(x, y) and some local image properties of neighboring pixels such as mean or variance.

Threshold function T(x, y) is given by ,
where T(x, y) =f (x, y) + T

1.4 Adaptive Thresholding

The approach is to compute the threshold for each pixel. If the pixel value is below the threshold it is set to the background value, otherwise it assumes the foreground value. In adaptive thresholding, different threshold values for different local areas are used.
Step 1: Reading the image using imread using openCV.

```
import cv2
import numpy as np

# path to input image is specified and
# image is loaded with imread command
# image1 = cv2.imread('threshold_img1.jpg')
image1 = cv2.imread('brain.jpg')
```

Step 2: Converting the image into gray scale image using function.

```
# cv2.cvtColor is applied over the
# image input with applied parameters
# to convert the image in grayscale
img = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
```

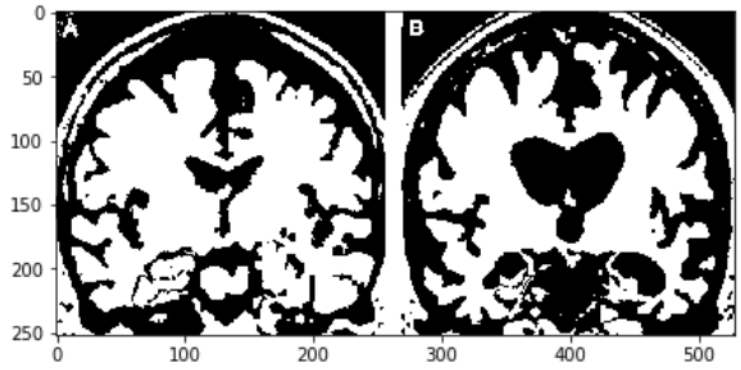
Step 3: Applying two methods of adaptive thresholding. Firstly, Using mean.

```
# applying different thresholding
# techniques on the input image
thresh1 = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH_BINARY, 199,
```

Step 4: Displaying the resulting image.

```
# the window showing output images
# with the corresponding thresholding
skimage.io.imshow(thresh1)
```

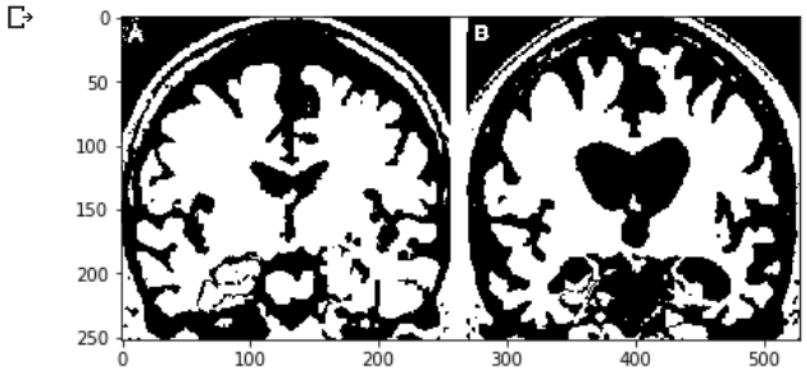
```
<matplotlib.image.AxesImage at 0x7f58c4a8af98>
```



Performing the step 3 and 4 for applying thresholding using Gaussian method.

```
thresh2 = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY, 19
```

```
[149] skimage.io.imshow(thresh2)
      # De-allocate any associated memory usage
      if cv2.waitKey(0) & 0xff == 27:
          cv2.destroyAllWindows()
```



There are two main approaches:

1.4.1 Chow and Kaneko approach

The assumption behind both method is that smaller image regions are more likely to have approximately uniform illumination, thus being more suitable for thresholding. The drawback of this method is that it is computational expensive and, therefore, is not appropriate for real-time applications. Chow and Kaneko suggest the use of a 7 X 7 window for thresholding. In their method, the original

image is divided into 7 x 7 sub images and a threshold is computed for each sub image.

1.4.2 Local thresholding

It selects an individual threshold for each pixel based on the range of intensity values in its local neighborhood. This allows for thresholding of an image whose global intensity histogram doesn't contain distinctive peaks.

Chapter 2

Region Based Segmentation

The method starts by taking a pixel from inside of an object or considering the whole image to be same and then growing outward and splitting respectively. Regions in an image are a group of connected pixels with similar properties. Compared to edge detection method, segmentation algorithms based on region are comparatively manageable and more immune to noise. In this, pixels corresponding to an object are grouped together and marked. Segmentation algorithms based on region mainly include following methods:

2.1 Region Growing

In this approach a seed pixel is chosen and neighboring pixels are joined to form a region based their similarities. This process is iterated for several boundary pixel in the region. If adjacent regions are found then a region-merging algorithm is used in which weak edges are disappeared and strong edges are left intact.

2.1.1 Algorithm

1. Select seed pixels within the image.
2. Then from each seed pixel grow a region:
a. Set the region prototype to be seed pixel;
b. Calculate the similarity between region prototype and candidate pixel;
c. Calculate the similarity between candidate and its nearest neighbor;
d. Include the candidate pixel if both similarity measures are higher than experimental set thresholds;
e. After that Update the region prototype;
f. At last go to the next pixel to be examined.

Step 1: Importing the required libraries.

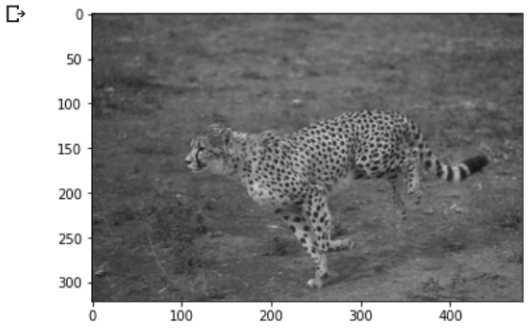
```
[ ]  import math
     # from IPython.display import HTML, Image, display
     from PIL import Image
     from pylab import *
     import matplotlib.cm as cm
     import scipy as sp
     import random
```

Step 2: Converting the image to numpy array.

```
#converting image to array
im = Image.open('input1.jpg').convert('L')
arr = np.asarray(im)

out = Image.open('out1.jpg').convert('L')
arr_out = np.asarray(out)
```

Step 3: Display the image.

```
[ ]  rows,columns = np.shape(arr)
     #print '\nrows',rows,'columns',columns
     plt.figure()
     plt.imshow(im)
     plt.gray()
```



Step 4: Selecting a seed pixel.

```
[ ]  # x=(int)(pseed[0][0])
     # y=(int)(pseed[0][1])
     x=int(179)
     y=int(86)
     seed_pixel=[]
     seed_pixel.append(x)
     seed_pixel.append(y)
```

```
[ ]  print('you clicked:',seed_pixel)
```

```
     you clicked: [179, 86]
```

```
[ ]  plt.close()
```

Step 5: Initializing x and y for 8-connected pixel.

```
#function for implementing region growing
def find_region():
  print('\n loop runs till region growing is complete')
  count=0
  x=[-1,0,1,-1,1,-1,0,1] # 8 connected
  y=[-1,-1,-1,0,0,1,1,1]
```

Step 6: The loop is iterated until length becomes zero.

```
 while(len(region_points)>0):
   if(count==0):
     point = region_points.pop(0)
     i=point[0]
     j=point[1]
   print('loop runs till length become zero:')
   print('len',len(region_points))
   val=arr[i][j]
   lt=val-8
   ht=val+8
```

Step 7: Assigning value to neighboring neighbors 1 or 0 and rest are added to the list.

```
    for  k in range(8):
      if(img_rg[i+x[k]][j+y[k]]!=1):
        try:
          if arr[i+x[k]][j+y[k]]>lt and arr[i+x[k]][j+y[k]]<ht:
            img_rg[i+x[k]][j+y[k]]=1
            p=[0,0]
            p[0]=i+x[k]
            p[1]=j+y[k]
            if p not in region_points:
              if 0<p[0] < rows and 0<p[1] < columns:
                region_points.append([i+x[k],j+y[k]])
          else:
            img_rg[i+x[k]][j+y[k]]=0
        except IndexError:
          continue
```

Step 8: For each pixel assigning the values we obtained using the function region growing.

```
      point=region_points.pop(0)
      i=point[0]
      j=point[1]
      count=count+1
find_region()
ground_out=np.zeros((rows,columns))
for i in range(rows):
  for j in range(columns):
    if arr_out[i][j] > 125:
      ground_out[i][j]=int(1)
    else:
      ground_out[i][j]=int(0)
tp=0
tn=0
fn=0
fp=0
```

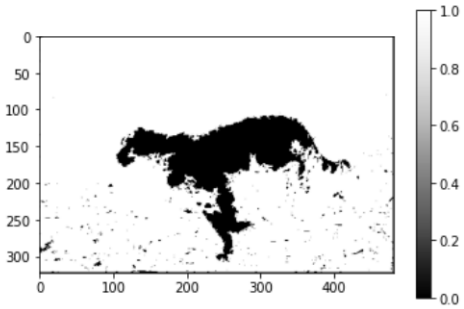
Step 9: Calculating False positive and True positive value to check error.

```
for i in range(rows):
  for j in range(columns):
    if ground_out[i][j] == 1 and img_rg[i][j] == 1:
      tp = tp + 1
    if ground_out[i][j] == 0 and img_rg[i][j] == 0:
      tn = tn + 1
    if ground_out[i][j] == 1 and img_rg[i][j] == 0:
      fn = fn + 1
    if ground_out[i][j] == 0 and img_rg[i][j] == 1:
      fp = fp + 1
```

Step 10: Displaying results.

```
#computing false positive and True positive
tpr= float(tp)/(tp+fn)
print("\nTPR is:",tpr)

#fp rate is
fpr= float(fp)/(fp+tn)
print("\nFPR is:",fpr)

#F-score as 2TP/(2TP + FP + FN)
fscore = float(2*tp)/((2*tp)+fp+fn)
print("\nFscore:",fscore)


plt.figure()
plt.imshow(img_rg, cmap="Greys_r")
plt.colorbar()
plt.show()
```

```
loop runs till region growing is complete

TPR is: 0.9893147738254824

FPR is: 0.2948300310428084

Fscore: 0.9655279951927831
```



This algorithm presents several advantages over other color image segmentation algorithms. Region growing approach is simple. The border of regions found by region growing are perfectly thin and connected. The algorithm is also very stable with respect to noise. Limitation is that, it requires a seed point, which generally means manual interaction. Thus, each region to be segmented, a seed point is needed.

2.2 Region Splitting and Merging

Split and merge method is the opposite of the region growing. This technique works on the complete image. Region splitting is a top-down approach. It appears with a complete image and splits it up such that the segregated sliced are more homogenous than the total. Splitting single is insufficient for sensible segmentation as it severely limits the shapes of segments. Hence, a merging phase after the splitting is always desirable, which is termed as the split- and-merge algorithm. Any region can be split into sub regions, and the appropriate regions can be merged into a region.

Region splitting and merging takes spatial information into consideration. The region-splitting and merging method is as follows:

2.2.1 Region splitting Method

1. Suppose R represent the entire image. Select a predicate P.
2. Split or subdivide the image successively into smaller and smaller quadrant regions.

It is represented in the form of a quad tree, the root of the tree corresponds to the entire image and each node corresponds to subdivision.

2.2.2 Region Merging Method

Merge any adjacent regions that are similar enough. The procedure for split and merge is given.
1. Firstly start with the whole image.
2. If the variance is too large then break it into quadrants.
3. Merge any adjacent regions that are similar enough.
4 .Repeat step (2) and (3) again and again until no more splitting or merging occurs.

Chapter 3

Artificial Neural Network Based Segmentation

Artificial neural networks are parallel computational models, comprised of densely interconnected adaptive processing units. An important feature of these networks is that they learn by example. The adaptive nature of the artificial neural networks makes it more suitable for applications where one has little or incomplete understanding of the problem to be solved but where training data is readily available.

3.1 Mask R-CNN for Image Segmentation

Mask R-CNN is basically an extension of Faster R-CNN. Faster R-CNN is widely used for object detection tasks. For a given image, it returns the class label and bounding box coordinates for each object in the image. The Mask R-CNN framework is built on top of Faster R-CNN. So, for a given image, Mask R-CNN, in addition to the class label and bounding box coordinates for each object, will also return the object mask.
Importing the required libraries.

```
[1]  %tensorflow_version 1.x
```

```
     TensorFlow 1.x selected.
```

```
[2]  !pip install imgaug
     !pip install Cython
     !pip install pycocotools
     !pip install kaggle
```

```
import sys
import random
import math
import numpy as np
import skimage.io
import matplotlib
import matplotlib.pyplot as plt

# Root directory of the project
ROOT_DIR = os.path.abspath("../")

# Import Mask RCNN
sys.path.append(ROOT_DIR)  # To find local version of the library
from mrcnn import utils
import mrcnn.model as modellib
from mrcnn import visualize
# Import COCO config
sys.path.append(os.path.join(ROOT_DIR, "samples/coco/"))  # To find local versior
import coco

%matplotlib inline

# Directory to save logs and trained model
MODEL_DIR = os.path.join(ROOT_DIR, "logs")

# Local path to trained weights file
COCO_MODEL_PATH = os.path.join(ROOT_DIR, "mask_rcnn_coco.h5")
# Download COCO trained weights from Releases if needed
if not os.path.exists(COCO_MODEL_PATH):
    utils.download_trained_weights(COCO_MODEL_PATH)

# Directory of images to run detection on
IMAGE_DIR = os.path.join(ROOT_DIR, "images")
```

Step 1: We will create an inference class which will be used to infer the Mask R-CNN model

```
class InferenceConfig(coco.CocoConfig):
    # Set batch size to 1 since we'll be running inference on
    # one image at a time. Batch size = GPU_COUNT * IMAGES_PER_GPU
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1

config = InferenceConfig()
config.display()
```

Step 2: We will create our model and load the pretrained weights.

```
]  # Create model object in inference mode.
   model = modellib.MaskRCNN(mode="inference", model_dir=MODEL_DIR, config=config)

   # Load weights trained on MS-COCO
   model.load_weights(COCO_MODEL_PATH, by_name=True)
```

Step 3: We will define the classes of the COCO dataset which will help us in the prediction phase

```
# COCO Class names
# Index of the class in the list is its ID. For example, to get ID of
# the teddy bear class, use: class_names.index('teddy bear')
class_names = ['BG', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
               'bus', 'train', 'truck', 'boat', 'traffic light',
               'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird',
               'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear',
               'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie',
               'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
               'kite', 'baseball bat', 'baseball glove', 'skateboard',
               'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup',
               'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple',
               'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
               'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed',
               'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote',
               'keyboard', 'cell phone', 'microwave', 'oven', 'toaster',
               'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors',
               'teddy bear', 'hair drier', 'toothbrush']
```

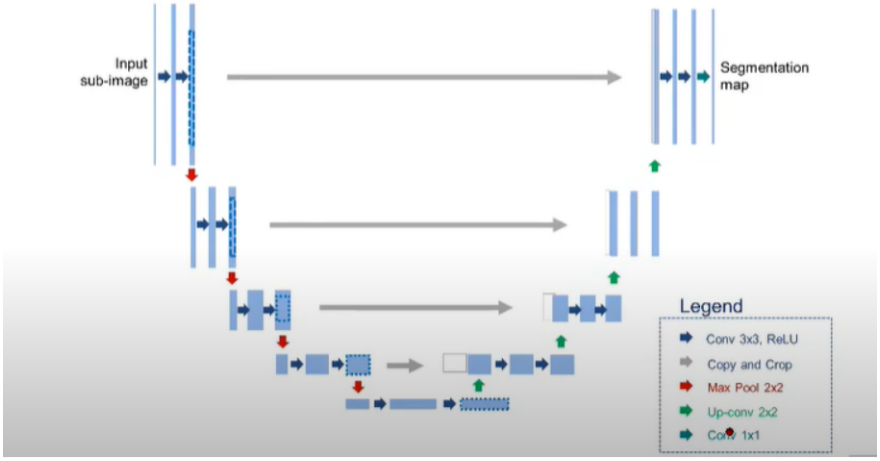
Step 4: Testing the model.

```
# Load a random image from the images folder
file_names = next(os.walk(IMAGE_DIR))[2]
image = skimage.io.imread(os.path.join(IMAGE_DIR, random.choice(file_names)))

# Run detection
results = model.detect([image], verbose=1)

# Visualize results
r = results[0]
visualize.display_instances(image, r['rois'], r['masks'], r['class_ids'],
                            class_names, r['scores'])
```

Output:

3.2 U-Net Architecture for Image Segmentation

U-Net is a convolutional neural network based on the fully convolutional network. The main idea is to supplement a usual contracting network by successive layers, where pooling operations are replaced by upsampling operators. Hence these layers increase the resolution of the output.



Importing the required libraries:

```
import tensorflow as tf
```

```
from tensorflow_examples.models.pix2pix import pix2pix

import tensorflow_datasets as tfds
tfds.disable_progress_bar()

from IPython.display import clear_output
import matplotlib.pyplot as plt
```

Step 1: Loading dataset.

```
dataset,info=tfds.load('oxford_iiit_pet:3.*.*',with_info=True)
```

Step 2: Normalizing the pixel value and augmenting the images.

```
] def normalize(input_image,input_mask):
    input_image=tf.cast(input_image,tf.float32)/255.0
    input_mask -= 1
    return input_image,input_mask
```

```
] @tf.function
  def load_image_train(datapoint):
    input_image = tf.image.resize(datapoint['image'], (128, 128))
    input_mask = tf.image.resize(datapoint['segmentation_mask'], (128, 128))

    if tf.random.uniform(()) > 0.5:
      input_image = tf.image.flip_left_right(input_image)
      input_mask = tf.image.flip_left_right(input_mask)

    input_image, input_mask = normalize(input_image, input_mask)

    return input_image, input_mask
```

```
] def load_image_test(datapoint):
    input_image = tf.image.resize(datapoint['image'], (128, 128))
    input_mask = tf.image.resize(datapoint['segmentation_mask'], (128, 128))

    input_image, input_mask = normalize(input_image, input_mask)

    return input_image, input_mask
```

Step 3: Applying some data shuffling.

```
[ ] TRAIN_LENGTH = info.splits['train'].num_examples
    BATCH_SIZE = 64
    BUFFER_SIZE = 1000
    STEPS_PER_EPOCH = TRAIN_LENGTH // BATCH_SIZE
```

```
[ ] train = dataset['train'].map(load_image_train, num_parallel_calls=tf.data.experimental.AUTOTUNE)
    test = dataset['test'].map(load_image_test)
```

```
[ ] train_dataset = train.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()
    train_dataset = train_dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
    test_dataset = test.batch(BATCH_SIZE)
```

```
[ ] def display(display_list):
      plt.figure(figsize=(15, 15))

      title = ['Input Image', 'True Mask', 'Predicted Mask']

      for i in range(len(display_list)):
        plt.subplot(1, len(display_list), i+1)
        plt.title(title[i])
        plt.imshow(tf.keras.preprocessing.image.array_to_img(display_list[i]))
        plt.axis('off')
      plt.show()
```

Step 4: Three channels for three class labels.

```
OUTPUT_CHANNELS = 3
# The reason to output three channels is because there
#  are three possible labels for each pixel
```

Step 5: The encoder will be a pretrained MobileNetV2 model which is prepared and ready to use in tf.keras.applications.

```
base_model = tf.keras.applications.MobileNetV2(input_shape=[128, 128, 3], include_top=False)

# Use the activations of these layers
layer_names = [
    'block_1_expand_relu',   # 64x64
    'block_3_expand_relu',   # 32x32
    'block_6_expand_relu',   # 16x16
    'block_13_expand_relu',  # 8x8
    'block_16_project',      # 4x4
]
layers = [base_model.get_layer(name).output for name in layer_names]

# Create the feature extraction model
down_stack = tf.keras.Model(inputs=base_model.input, outputs=layers)

down_stack.trainable = False
```

Step 6: Defining the model and applying skip connections.

```
def unet_model(output_channels):
  inputs = tf.keras.layers.Input(shape=[128, 128, 3])
  x = inputs

  # Downsampling through the model
  skips = down_stack(x)
  x = skips[-1]
  skips = reversed(skips[:-1])

  # Upsampling and establishing the skip connections
  for up, skip in zip(up_stack, skips):
    x = up(x)
    concat = tf.keras.layers.Concatenate()
    x = concat([x, skip])

  # This is the last layer of the model
  last = tf.keras.layers.Conv2DTranspose(
      output_channels, 3, strides=2,
      padding='same')  #64x64 -> 128x128

  x = last(x)

  return tf.keras.Model(inputs=inputs, outputs=x)
```

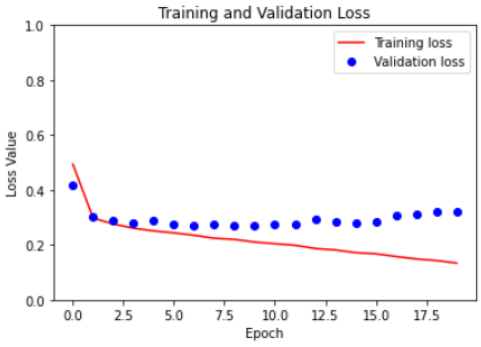
Step 7: For this dataset we will use Sparse Categorical cross entropy function.

```
model = unet_model(OUTPUT_CHANNELS)
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Step 8: To stop the epoch when we reach a decent accuracy and storing loss for displaying.

```
class DisplayCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs=None):
    clear_output(wait=True)
    show_predictions()
    print ('\nSample Prediction after epoch {}\n'.format(epoch+1))
```

```
EPOCHS = 20
VAL_SUBSPLITS = 5
VALIDATION_STEPS = info.splits['test'].num_examples//BATCH_SIZE//VAL_SUBSPLITS

model_history = model.fit(train_dataset, epochs=EPOCHS,
                          steps_per_epoch=STEPS_PER_EPOCH,
                          validation_steps=VALIDATION_STEPS,
                          validation_data=test_dataset,
                          callbacks=[DisplayCallback()])
```



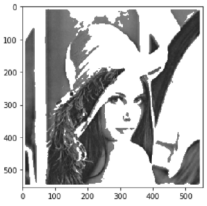
Results:

```
[ ]  show_predictions(test_dataset, 3)
```

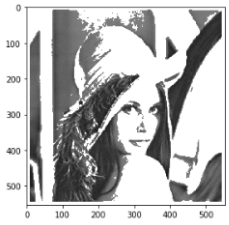
Chapter 4

Comparison between all the Methods

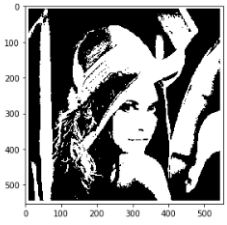
Some of the results of Image segmentation using thresholding are:



Histogram based Thresolding



Otsu Threshold



Adaptive Thresholing


Local Median


Local Mean

1. Histogram based thresolding involves manual intervention and threshold needs to be found out using threshold and therefore cannot be used in real time.

2. Otsu method works well for some images and give poor results for certain types of images. The results from Otsu have too much of noise in the form of the background being detected as foreground. The main advantage is the simplicity of calculation of the threshold. Since it is a global algorithm it is well suited only for the images with equal intensities. The method does not work well with variable illumination This might not give a good result for the images with lots of variation in the intensities of pixels.

3. The drawback of adaptive thresholding is that it is computational expensive and, therefore, is not appropriate for real-time applications.

The results of Image segmentation using Region growing are:

We can see that the result is better that those obtained by thresolding on this image but may not be the same case for other. Also,this algorithm presents several advantages over other color image segmentation algorithms. Region growing approach is simple. The border of regions found by region growing are perfectly thin and connected. The algorithm is also very stable with respect to noise.

Limitation is that, it requires a seed point, which generally means manual interaction. Thus, each region to be segmented, a seed point is needed.

The results of Image Segmentation using Neural network:

Using U-Net Model:

The disadvantage is that that learning may slow down in the middle layers of deeper models. This is due to gradients becoming diluted further away from the output of a network, where the error is computed, resulting in slower learning for far-removed weights. The advantages of U-Net models are probably worth the risk, though, and choosing the right activation functions and regularization parameters can probably mitigate adverse effects caused by vanishing gradients in the middle layers of U-Net models.

Using Mask R-CNN Model:

We see that mask R-CNN works very well in image segmentation. But requires pre knowledge to predict anything i.e. it requires dataset to train it. It is an extension of Faster R-CNN, therefore it is better as compared U-Net model.

There is no single method which can be considered good for all type of images or all methods equally good for a particular type of image. Depending upon the type image we need to select the type of algorithm for Image Segmentation.
References:
https://www.researchgate.net/publication/309209325_A_Survey_on_Threshold_Based_Segmentation_Technique

https://pdfs.semanticscholar.org/b5ba/bd20d9409d3dfef8fc84c196fbeacc616247.pdf