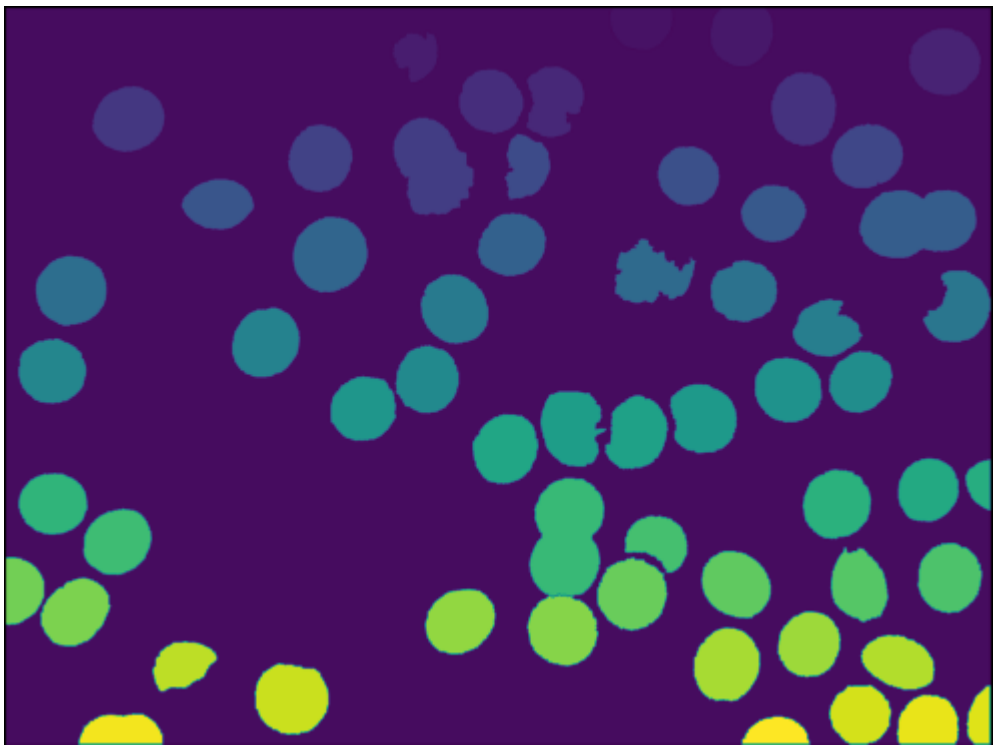
Computer vision:
Image segmentation
&
Object counting



Oleg Bissing
Applied AI course, WS 2023/24
Prof. Dr. Bahrpeyma

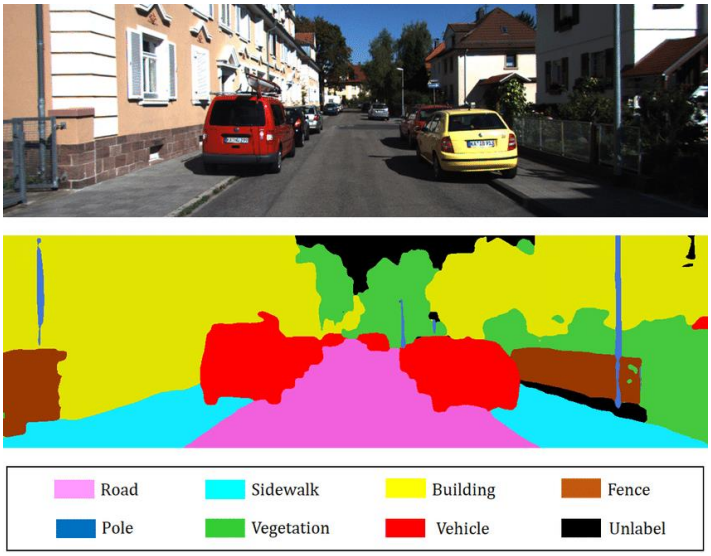
Content

1. Introduction 3
2. First approach: Watershed algorithm 4
3. Second approach: U-NET model 10
4. Evaluation 14

1. Introduction

Computer vision (CV) tasks include methods for acquiring, processing, analysing and understanding digital images, and extraction of high-dimensional data from the real world in order to produce numerical or symbolic information, e.g. in the forms of decisions[1].

In digital image processing and computer vision, **image segmentation** is the process of partitioning a digital image into multiple image segments, also known as image regions or image objects (sets of pixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyse[2].



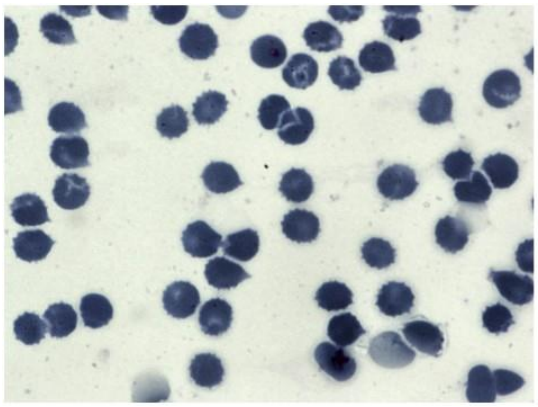
Object counting with computer vision is a process of using computer algorithms to automatically detect and count the number of objects in an image or video stream.

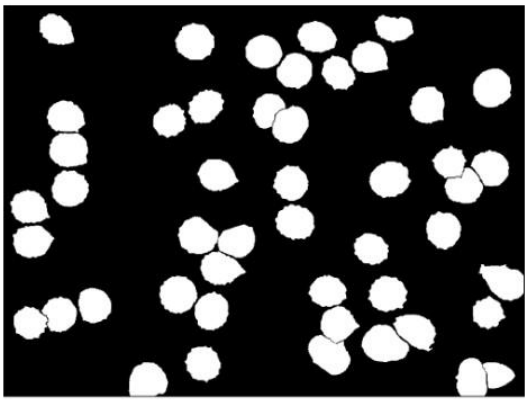
The goal of this project is to use different computer vision techniques to segment the image, then count all the objects presented and after that to compare the results.

Possible approaches:

1. To use some CV algorithm for both segmentation and counting;
2. To use come ML model for segmentation and then count.

Dataset for this project is “Blood Cell Segmentation Dataset” free available on Kaggle[3]. A total of 2656 images are available. 1328 Original blood cell images with 1328 corresponding ground truths.





[1] Reinhard Klette (2014). Concise Computer Vision. Springer. ISBN 978-1-4471-6320-6.

[2] Linda G. Shapiro and George C. Stockman (2001): "Computer Vision", pp 279–325, New Jersey, Prentice-Hall, ISBN 0-13-030796-3

[3] https://www.kaggle.com/datasets/jeetblahiri/bccd-dataset-with-mask

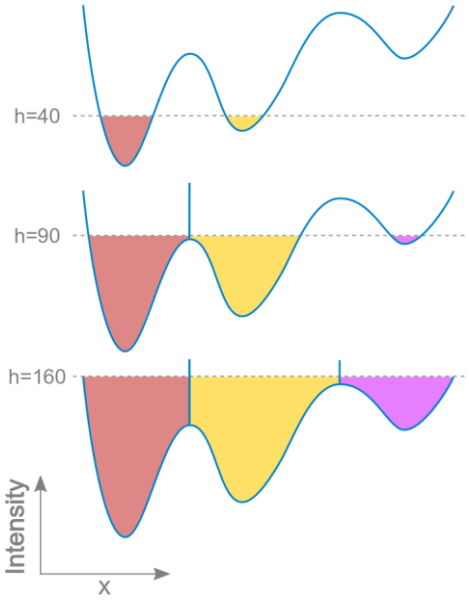
2. First approach: Watershed algorithm

The algorithm visualizes an image as a topographic landscape, producing ‘catchment basins’ and ‘watershed ridge lines’ within the image to segregate different objects.

In a simplified manner, any grayscale image can be viewed as a topographic surface where high intensity denotes peaks and hills while low intensity denotes valleys.

As the first step was to make out of RGB image a grayscale one because colours in this case are not needed. On top of it was used threshold with Otsu algorithm.

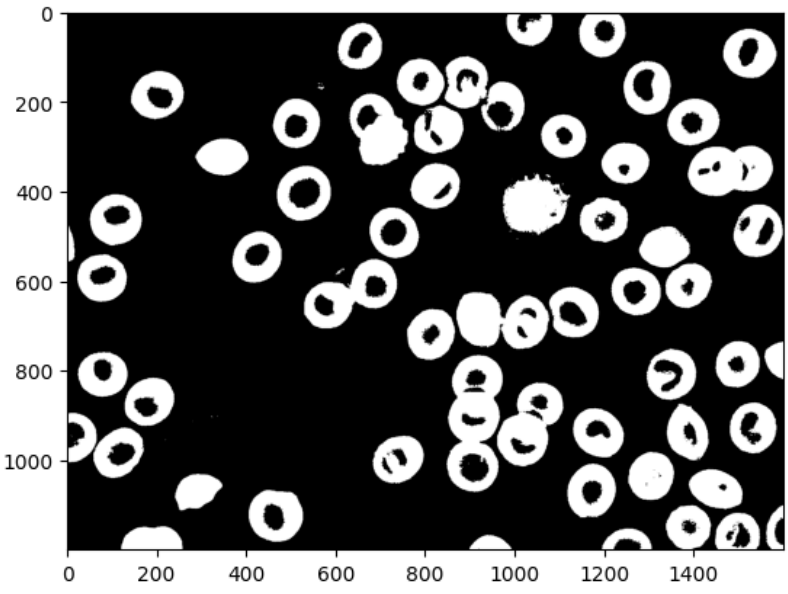
Thresholding is a technique that involves assigning pixel values in proportion to the provided threshold. Thresholding compares each pixel value to the threshold value.



If the pixel value is less than the threshold, it is set to 0, otherwise it is set to a maximum value (generally 255). Thresholding is a very popular segmentation technique used to separate an object viewed as the foreground from its background. A threshold is a value that has two ranges on either side, i.e. below the threshold or above the threshold.

Otsu algorithm separates an image into two classes, foreground, and background, based on the grayscale intensity values of its pixels. Furthermore, Otsu's method uses the grayscale histogram of an image to detect an optimal threshold value that separates two regions with maximum inter-class variance.

```
In [4]: gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

In [5]: # Threshhold + Otsu algorithm
        ret, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)

In [7]: plt.imshow(thresh, cmap='gray')

Out[7]: <matplotlib.image.AxesImage at 0x7e0f32d481f0>
```



Despite of being not very noisy (at least this particular one) these images are still better to denoise because it is a common approach which provides better performance.

In [8]:
```
# Noise removal
kernel = np.ones((3,3),np.uint8)
kernel
```

Out[8]:
```
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]], dtype=uint8)
```

In [9]:
```
opening = cv2.morphologyEx(thresh,cv2.MORPH_OPEN,kernel,iterations=2)
```

The next step was to fill the “holes” in the cells to be able to use the watershed algorithm.

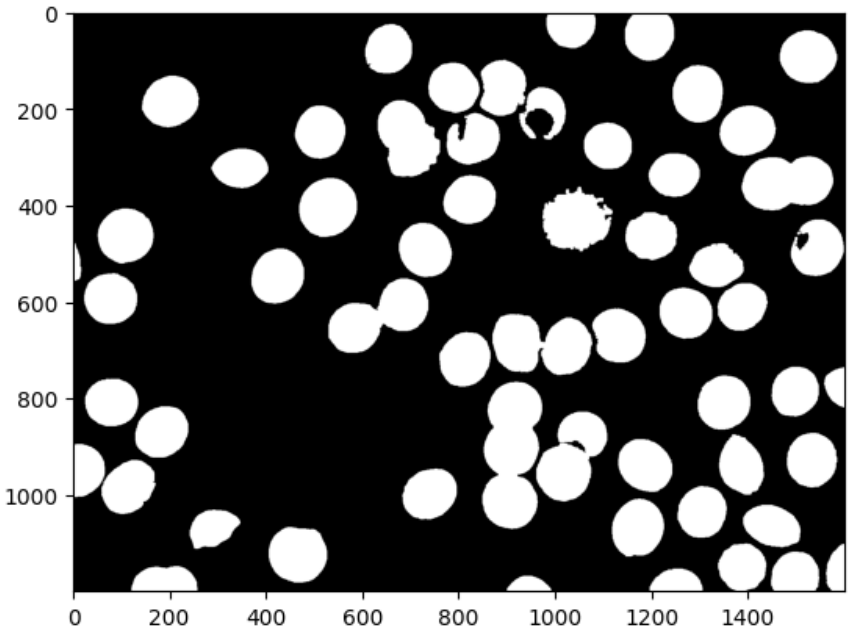
In [11]:
```
# Filling the holes
test = opening.copy()
```

In [12]:
```
th,im_th = cv2.threshold(test,220,255,cv2.THRESH_BINARY_INV)
im_floodfill = im_th.copy()
h,w = im_th.shape[:2]
```

In [13]:
```
mask = np.zeros((h+2,w+2), np.uint8)
cv2.floodFill(test,mask,(0,0),255)
im_floodfill_inv = cv2.bitwise_not(test)
im_out = im_floodfill_inv + opening.copy()
```

In [14]:
```
plt.imshow(im_out,cmap='gray')
```

Out[14]:
```
<matplotlib.image.AxesImage at 0x7e0f32a82b30>
```



Using of the watershed algorithm consists of the number of important steps where the right order is essential. First of them is a distance transform.

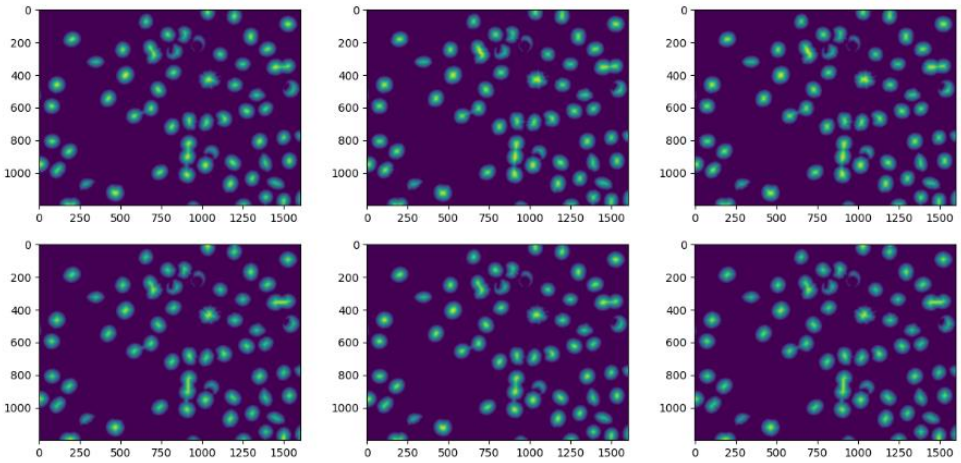
The distance transform (sometimes called the Euclidean distance transform) replaces each pixel of a binary image with the distance to the closest background pixel. If the pixel itself is already part of the background, then this is zero. The result is an image called a distance map.

There are different distance types, that could be used depending on the task. For this case after some trial with six of them was obvious, that there is no noticeable difference.

In [51]:
```
# Distance transformation
dist_transform_1 = cv2.distanceTransform(im_out,cv2.DIST_L2,5)
dist_transform_2 = cv2.distanceTransform(im_out,cv2.DIST_LABEL_PIXEL,5)
dist_transform_3 = cv2.distanceTransform(im_out,cv2.DIST_L1,5)
dist_transform_4 = cv2.distanceTransform(im_out,cv2.DIST_MASK_3,5)
dist_transform_5 = cv2.distanceTransform(im_out,cv2.DIST_LABEL_CCOMP,0)
dist_transform_6 = cv2.distanceTransform(im_out,cv2.DIST_C,5)
```

In [59]:
```
image_datas = [dist_transform_1,dist_transform_2,dist_transform_3,dist_transform_4,dist_transform_5,dist_transform_6]
f, axarr = plt.subplots(2,3,figsize=(15, 7))
axarr[0,0].imshow(image_datas[0])
axarr[0,1].imshow(image_datas[1])
axarr[0,2].imshow(image_datas[2])
axarr[1,0].imshow(image_datas[3])
axarr[1,1].imshow(image_datas[4])
axarr[1,2].imshow(image_datas[5])
```

Out[59]: <matplotlib.image.AxesImage at 0x7e0f2dbcae00>

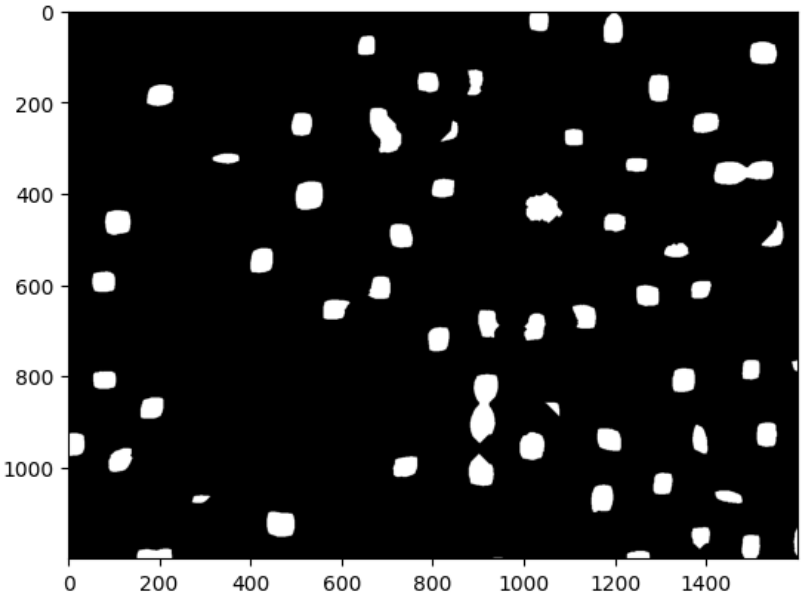


After applying a distance transformation image is separated into following regions: “sure it is the foreground”, “sure it is the background”, “not sure” (last one by subtracting the other two from one another).

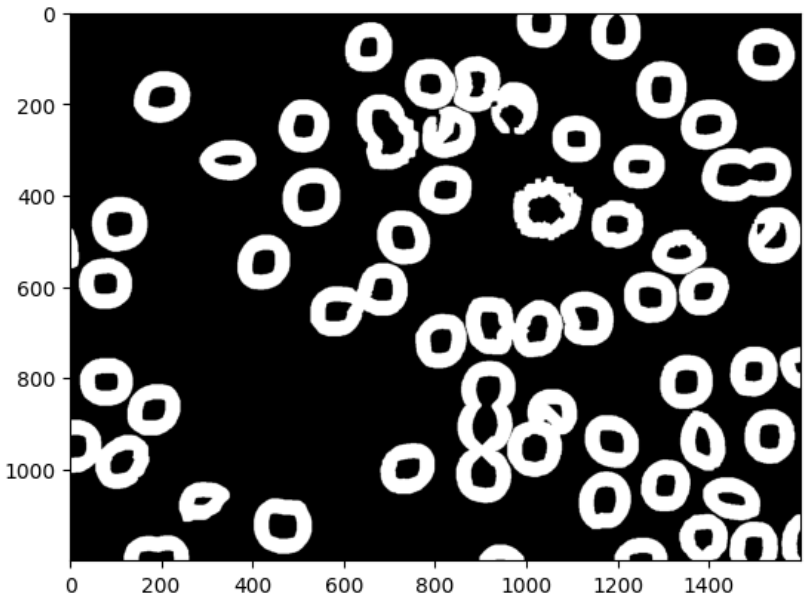
In [61]:
```
ret, sure_fg = cv2.threshold(dist_transform_2,0.5*dist_transform_2.max(),255,0)
```

In [62]:
```
plt.imshow(sure_fg,cmap='gray')
```

Out[62]: <matplotlib.image.AxesImage at 0x7e0f2dba7190>

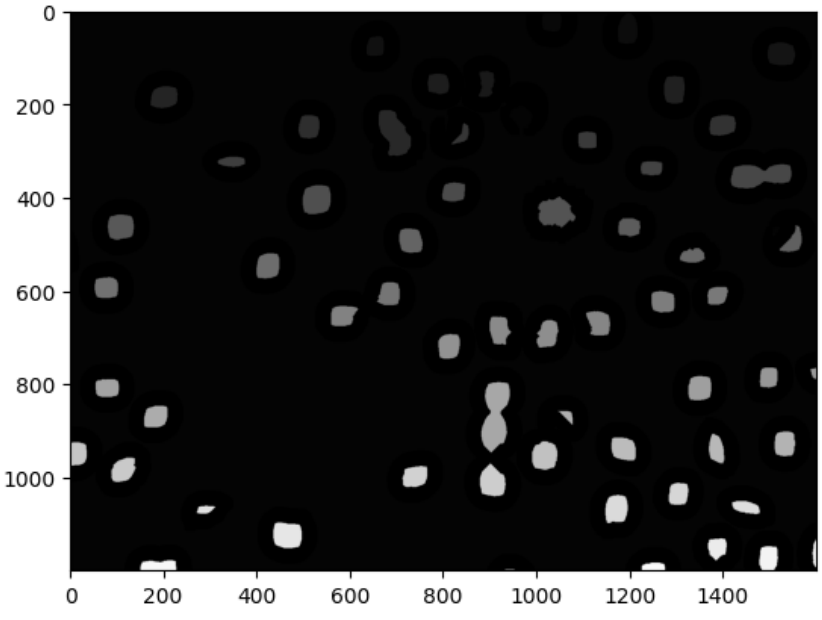


```
In [64]: sure_fg = np.uint8(sure_fg) # casting as an integer

In [65]: unknown = cv2.subtract(sure_bg,sure_fg)

In [67]: plt.imshow(unknown,cmap='gray')

Out[67]: <matplotlib.image.AxesImage at 0x7e0f2da9c730>
```



After that so-called markers are used to mark the objects, located in the foreground.

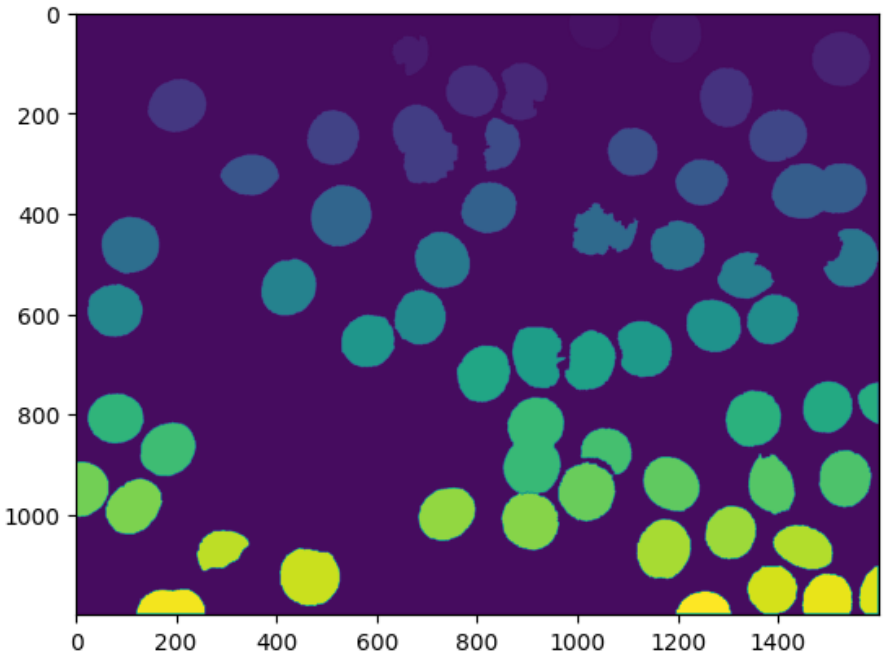
```
In [68]: ret,markers = cv2.connectedComponents(sure_fg)

In [69]: markers = markers + 1

In [70]: markers[unknown==255] = 0

In [72]: plt.imshow(markers,cmap='gray')

Out[72]: <matplotlib.image.AxesImage at 0x7e0f2d7882b0>
```



In [73]:
```
markers = cv2.watershed(img,markers)
```

In [81]:
```
plt.imshow(markers)
```

Out[81]:
```
<matplotlib.image.AxesImage at 0x7e0f2d4527d0>
```

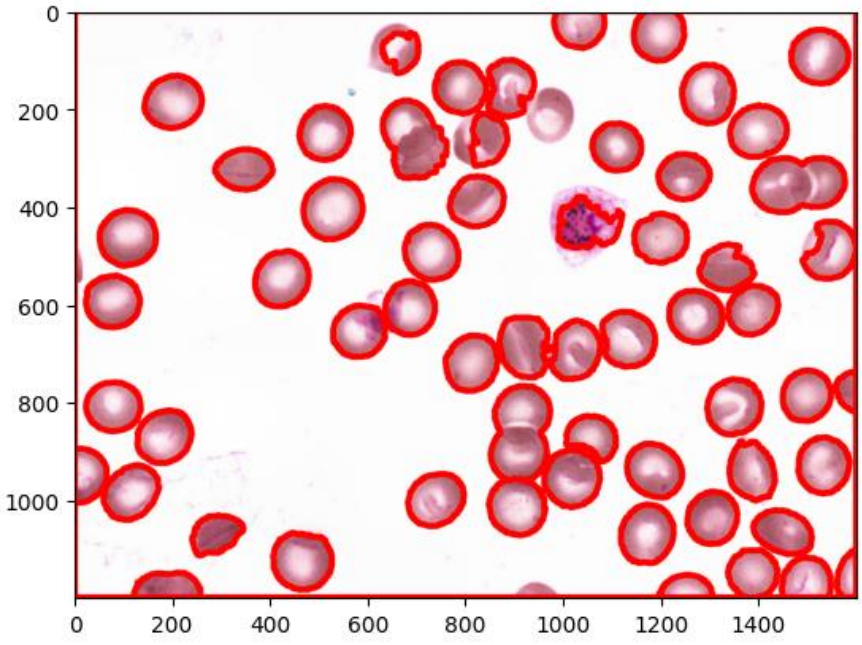


Finally, findContours() method is used to count the number of cells.

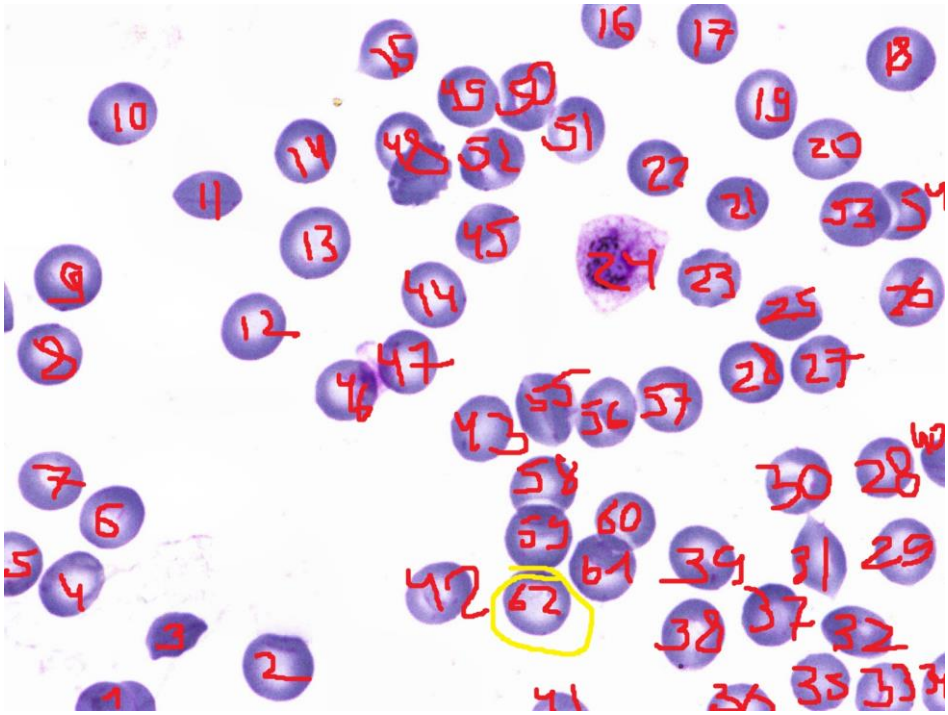
In [75]:
```
contours,hierarichy = cv2.findContours(markers.copy(),cv2.RETR_CCOMP,cv2.CHAIN_APPROX_SIMPLE)

for i in range(len(contours)):
  if hierarichy[0][i][3] == -1:
    cv2.drawContours(img,contours,i,(255,0,0),10)
```

In [76]:
```
plt.imshow(img,cmap='gray')
print('Objects number is:', ret-1)
```

```
Objects number is: 60
```



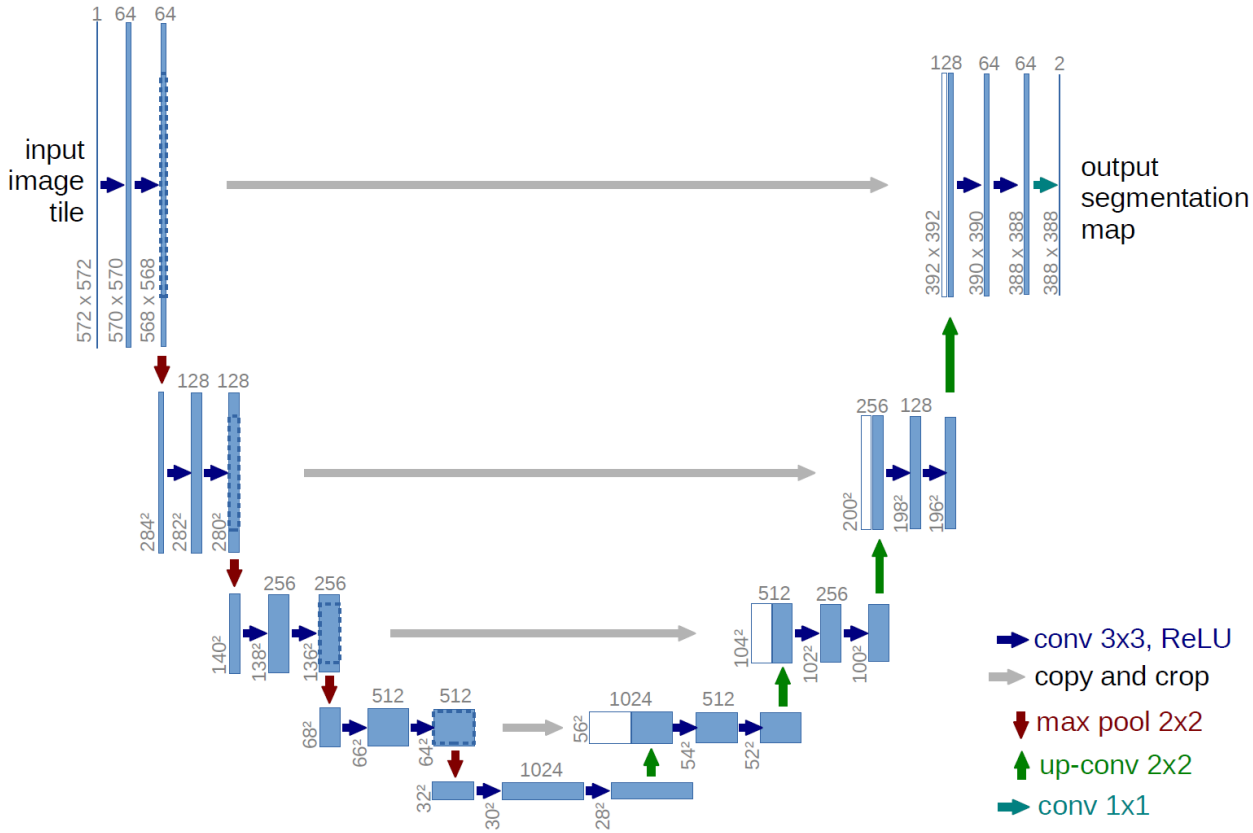
Algorithm counted 60 cells which is a very high result – when counting per hand it is around 62.



3. Second approach: U-NET model

A lot of time was invested researching how to complete the same task using machine/deep learning. U-NET model was chosen because it looked convenient and there was enough information and examples on the Internet to find. As a starting point were used a YouTube video, where a related, but not similar problem was solved using the U-NET model[4]; and also a code posted on the GitHub[5].

The U-NET is convolutional network architecture for fast and precise segmentation of images. 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. A successive convolutional layer can then learn to assemble a precise output based on this information.



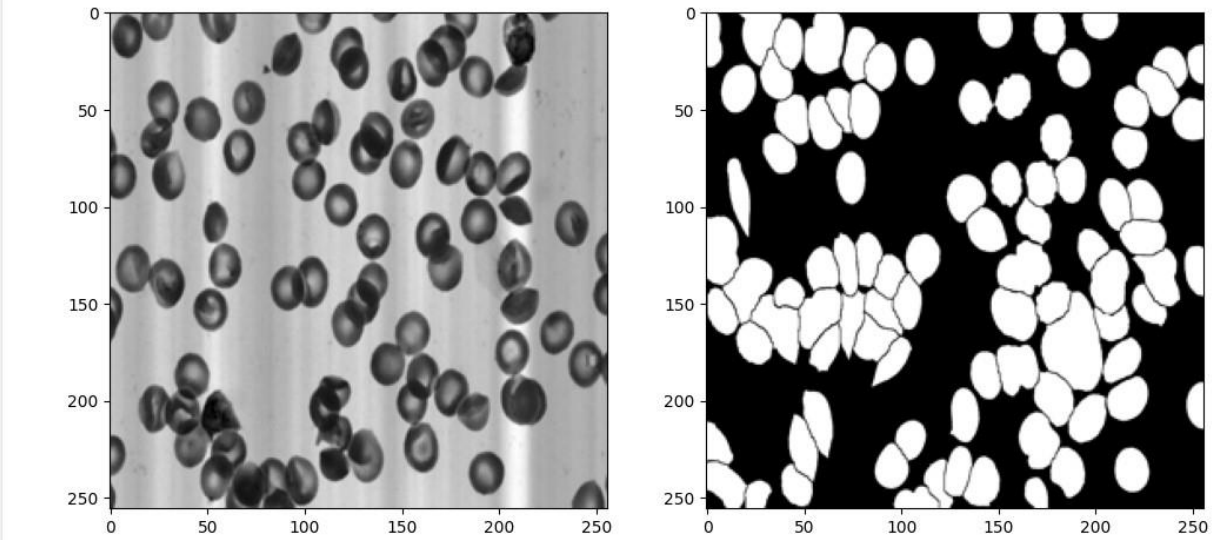
After downloading the dataset data had to be prepared for training. Some of the images we in .png, some in .jpg, so it had to be taken into account.

```
for i, image_name in enumerate(images):    #Remember enumerate method adds a counter
    if (image_name.split('.')[1] == 'png' or image_name.split('.')[1] == 'jpg'):
```

After data processing during a sanity check was clear, that some of the masks were not paired with the right image:

[4] https://www.youtube.com/watch?v=csFGTLT6_WQ&t=957s&ab_channel=DigitalSreeni
[5] https://github.com/Deponker/Blood-cell-segmentation/blob/main/U-net/training-validation.ipynb

```
#Sanity check, view few mages
import random
import numpy as np
image_number = random.randint(0, len(X_train))
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.imshow(np.reshape(X_train[image_number], (256, 256)), cmap='gray')
plt.subplot(122)
plt.imshow(np.reshape(y_train[image_number], (256, 256)), cmap='gray')
plt.show()
```



At first no attention was paid to it, but then it was found out that it could severely damage the results if there are too many of such pairs. So different ways to fix it were tried, the best one was to use sort() method.

The next step was to create and prepare the model. The model was first used without any changes or manipulations, but in this state it did not learn.

```
history = model.fit(X_train, y_train,
                    batch_size = 16,
                    verbose=1,
                    epochs=10,
                    validation_data=(X_test, y_test),
                    shuffle=False)

model.save('Image_segmentation_1.hdf5')
```

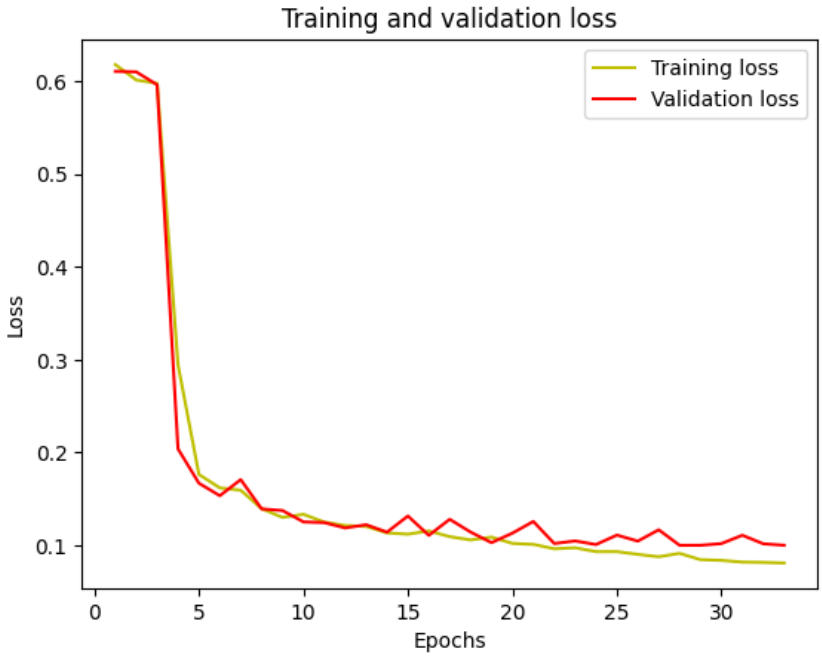
```
Epoch 1/10
66/66 [==============================] - 958s 14s/step - loss: 0.6104 - accuracy: 0.6631 - val_loss: 0.5946 - val_accuracy: 0.6777
Epoch 2/10
66/66 [==============================] - 974s 15s/step - loss: 0.6036 - accuracy: 0.6633 - val_loss: 0.5919 - val_accuracy: 0.6777
Epoch 3/10
66/66 [==============================] - 989s 15s/step - loss: 0.6023 - accuracy: 0.6633 - val_loss: 0.5917 - val_accuracy: 0.6777
Epoch 4/10
66/66 [==============================] - 981s 15s/step - loss: 0.6017 - accuracy: 0.6633 - val_loss: 0.5913 - val_accuracy: 0.6777
Epoch 5/10
66/66 [==============================] - 954s 14s/step - loss: 0.6016 - accuracy: 0.6633 - val_loss: 0.5912 - val_accuracy: 0.6777
Epoch 6/10
 2/66 [..............................] - ETA: 15:43 - loss: 0.5762 - accuracy: 0.6936
```

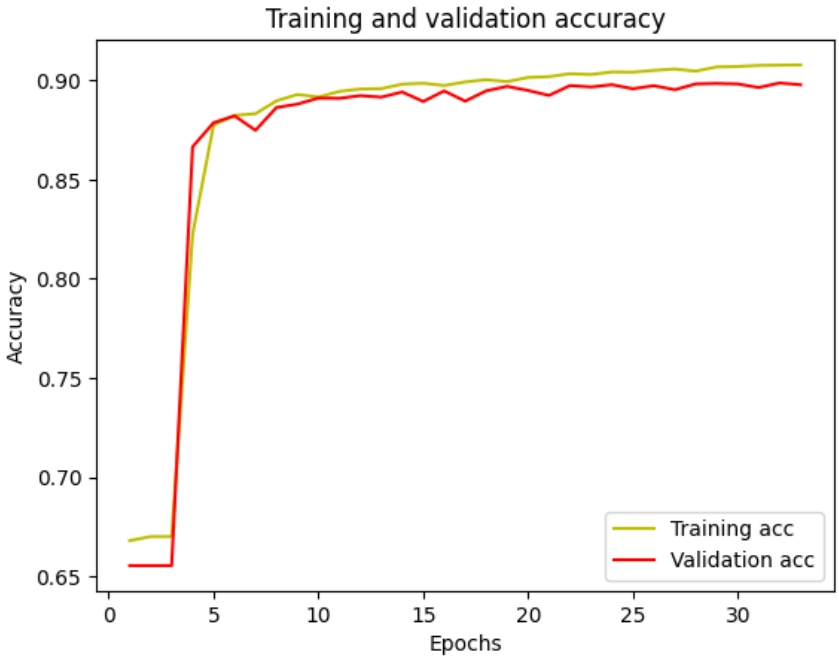
At this point hours of research, trial and error were needed to figure out how to fix this problem. Among other changes were attempts to change the model build (does not help), change the batch size 32 => 16 => 1 (does help a little), were tried a lot of different metrics to see whether these ones just not show the progress properly (which was not the reason of not changing metric values, but of course accuracy is definitely not the best metric for this type of task), were tried different amount of epochs to see whether the model just need much more of them to perform (which was not the case).

Turning point was to change the learning rate from 0.001 to 0.0001. After that change model performed straight away. Epochs = 10 was enough (were tried several times using early stopping => always stopped around this point).

In [24]:
```
# evaluate model
_, acc = model.evaluate(X_test, y_test)
print("Accuracy = ", (acc * 100.0), "%")
```

```
4/4 [==============================] - 8s 1s/step - loss: 0.1104 - accuracy: 0.8946
Accuracy =  89.45555090904236 %
```



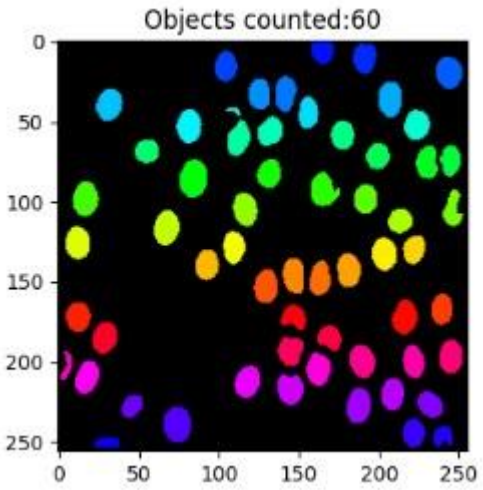


In [27]:

```
#IOU
y_pred=model.predict(X_test)
y_pred_thresholded = y_pred > 0.5

intersection = np.logical_and(y_test, y_pred_thresholded)
union = np.logical_or(y_test, y_pred_thresholded)
iou_score = np.sum(intersection) / np.sum(union)
print("IoU socre is: ", iou_score)
```

```
4/4 [==============================] - 1s 78ms/step
IoU socre is:  0.8274875328678938
```

Probably the most important value was the counted number of cells on the same image as the watershed algorithm counted 60 (the model have not seen this image not only while training but also during evaluation). The number was also 60.



4. Evaluation

It is clear, that for competitive evaluation it is not enough to test approaches on a single image. Unfortunately, taking this project even that far costed a lot of time and effort, much more that was planned before start. This is the reason why more tests were not possible.

Some conclusions to validate results of working on this project could still be made.

1. Within computer vision field for solving the same problem different approaches are possible. It could be both machine/deep learning and algorithmic approach.

2. Watershed algorithm is a solid, general purpose easy-to-use algorithm for image segmentation which allows to count object after applying it without using additional tools.

3. U-NET model, which was specially designed to solve the bio-medical segmentation problems still needs a good understanding of the hyperparameters and the right use of them for each task.

4. During this project both approaches showed exactly the same result (60 cells counted) on the same image, which was very close to the right count (62 per hand). At this point should be said, that for objective comparison to decide which approach is better for particular task more tests are needed, but first glance looks very promising for both of them.

5. This project was a great opportunity for the student to get first experience and basic understanding in image segmentation as a part of computer vision.