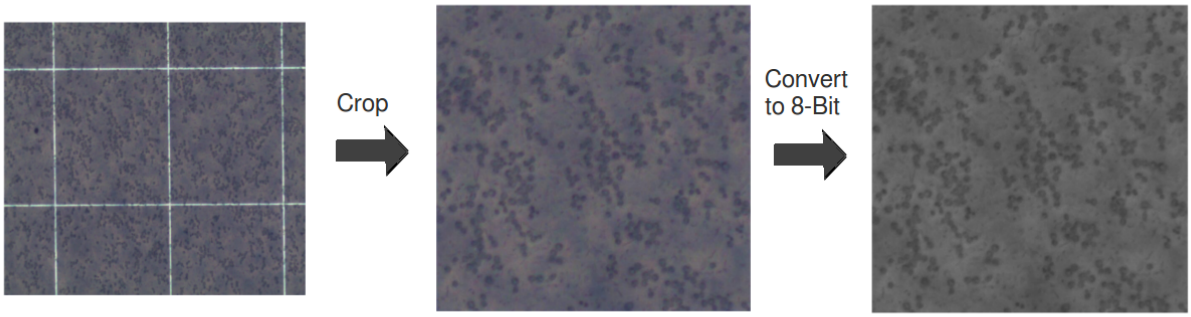
IISc-Bangalore Cell Counting software InterLab Collaboration

Aim: To attempt to make a cell counting software to reduce manual labor required for counting cells.

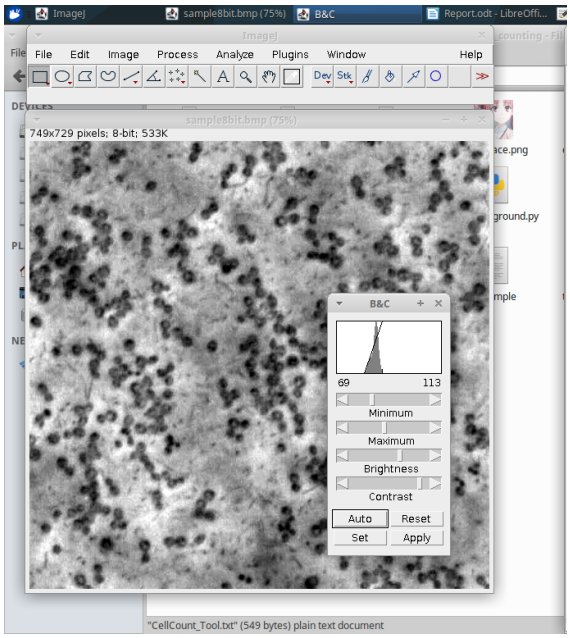
Abstract: The conventional image processing method for counting cells was first tried, using standard image processing techniques like contrasting, background subtraction, binary masking and watershedding through ImageJ. It was found that various parameters differed between images, and optimizing these parameters required a considerable amount of manual work. A machine learning approach was proposed using an Artificial Neural Network (ANN). To generate the dataset for training and testing the ANN the iGEM-IISc team collaborated with other iGEM teams across India. As a proof of concept, another simple dataset was generated, on which the ANN was trained and tested. But the trained ANN was unable to predict the test data accurately, showing that the current approach to this problem would not work.

Image processing through ImageJ

Step 1: The haemocytometer image was first cropped and converted to 8-bit (to make it grayscale and make binary operations easier later).

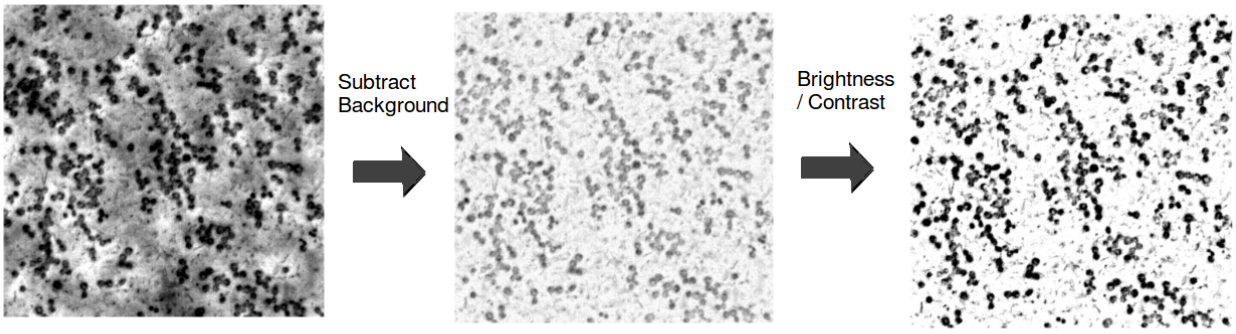


Step 2: Brightness/Contrast levels were changed to darken the cells and lighten the background.

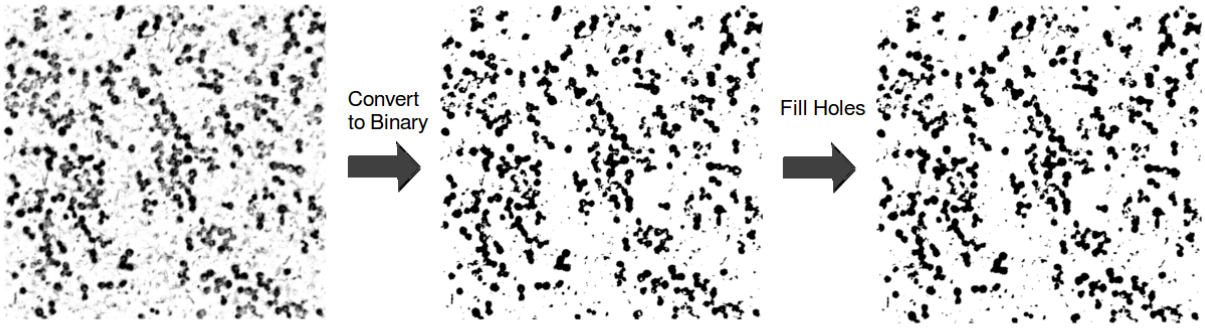


Brightness/Contrast Adjustment

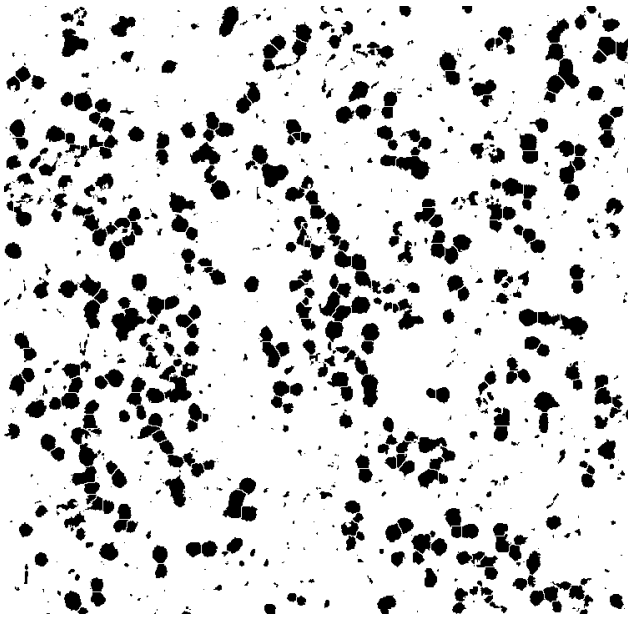
Step 3: Background subtraction is done. The rolling ball radius has to be figured out by trial and error. This is followed by another Brightness/Contrast adjustment.



Step 4: The image is then converted to a binary mask. And the holes in the binary mask are filled using the fill holes option in ImageJ

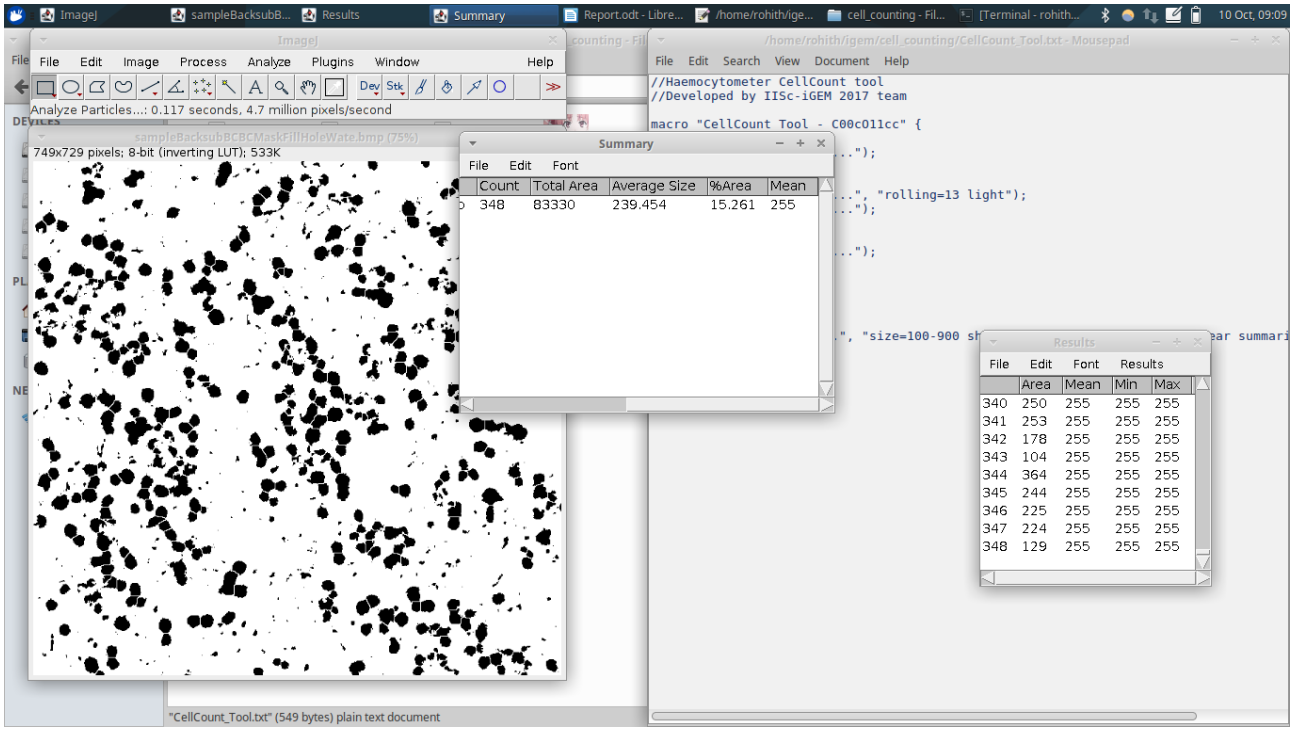


Step 5: The watershed tool in ImageJ is applied. This segments the clustered cells by drawing a 1-pixel line between cells. This step is essential before counting. Proper contrast and binary mapping is essential for the watershed algorithm to yield desirable results.



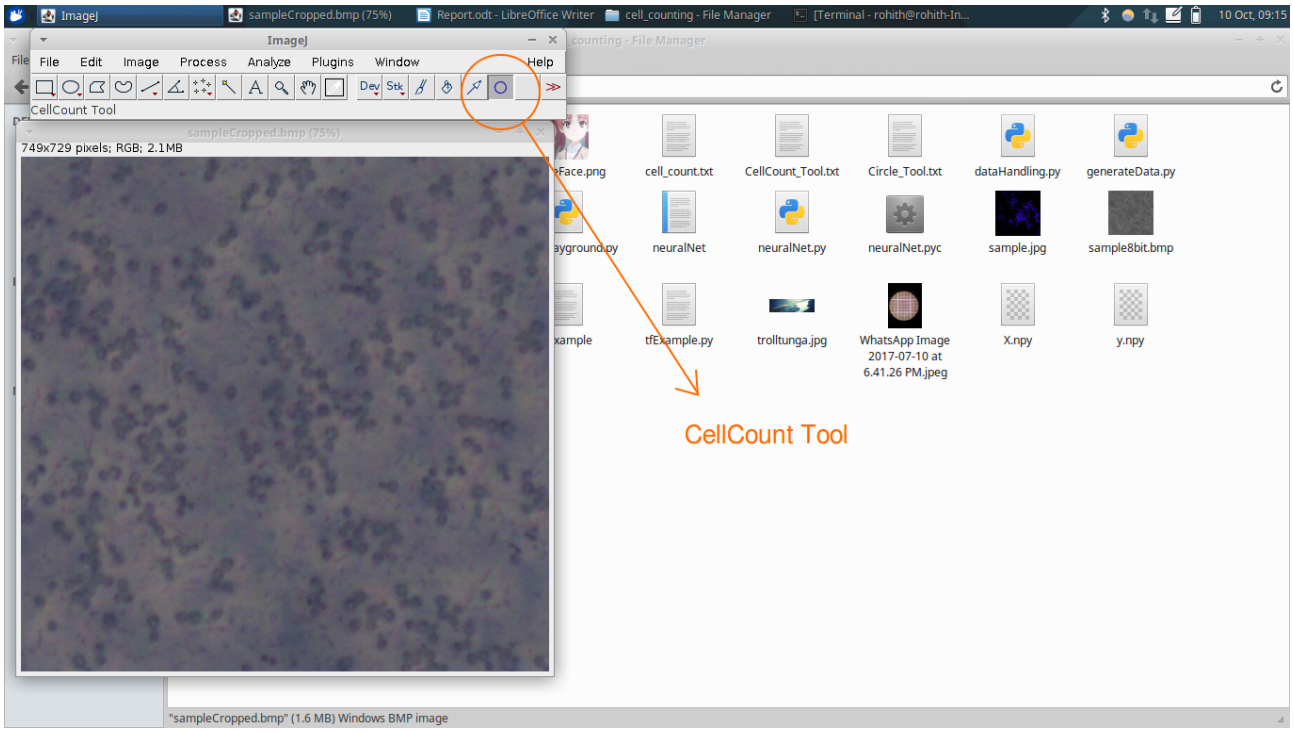
Watershed applied to binary image

Step 6: The Analyze Particles option is used to count the number of cells. The minimum and maximum size are set according to the image.

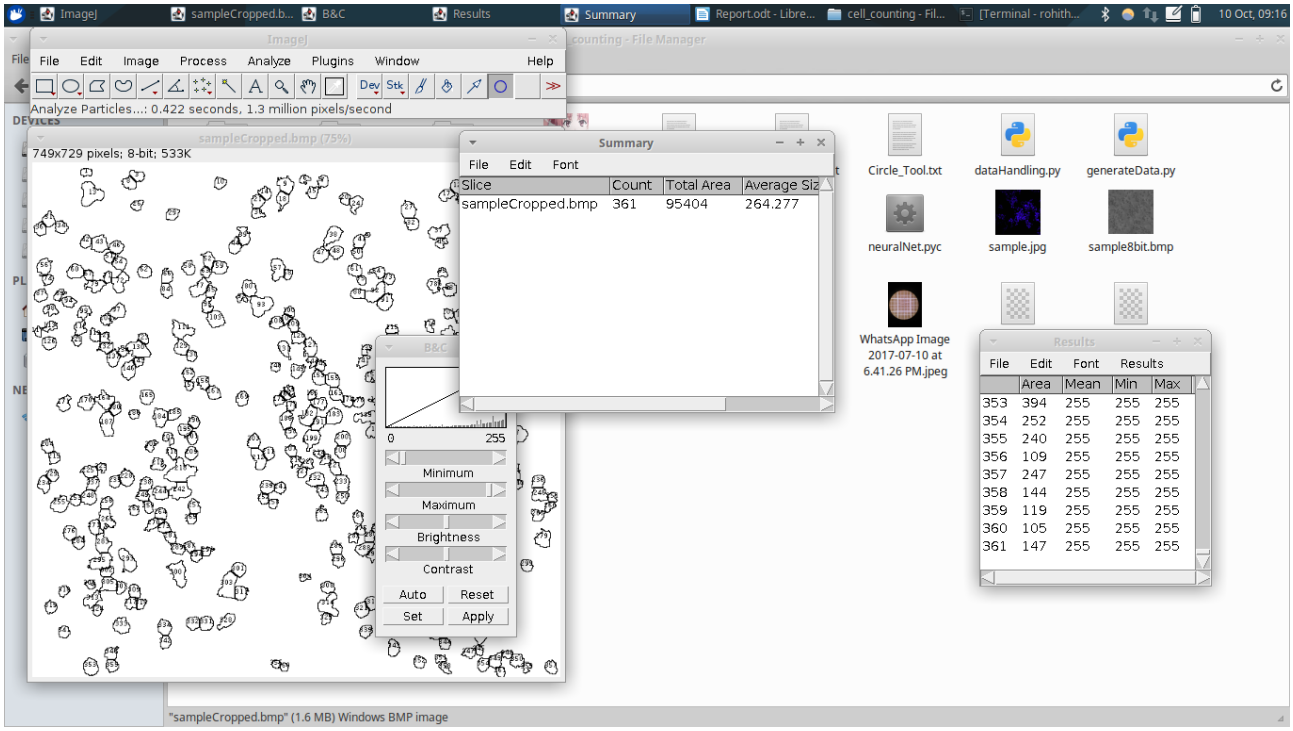


The final count of 348 is shown in the summary window. The actual count is about 370.

Writing a macro for a tool: If there are a set of images taken under similar conditions of lighting, magnification, etc.. There is a high chance that the same processing steps with the same parameters work for all the images in the set. It helps to write a macro/ create a tool in ImageJ for this purpose. This was done for the above cell counting steps.



Toolbar with the CellCount tool. Just select the tool and click on the image.



Applying the CellCount tool gives the cell count in a single step. But this only works with similar images, snce different types of images have different processing parameters.

Macro code [ImageJ macro Language]:

```
//Haemocytometer CellCount tool
//Developed by IISc-iGEM 2017 team

macro "CellCount Tool - C00cO11cc" {
run("8-bit");
run("Brightness/Contrast...");
setMinAndMax(72, 108);
run("Apply LUT");
run("Subtract Background...", "rolling=13 light");
run("Brightness/Contrast...");
setMinAndMax(80, 230);
run("Apply LUT");
run("Brightness/Contrast...");
setMinAndMax(90, 255);
run("Apply LUT");
run("Convert to Mask");
run("Fill Holes");
run("Watershed");
run("Analyze Particles...", "size=100-900 show=Outlines display
exclude clear summarize in_situ");
}
```

Machine Learning Approach

Introduction to Artificial Neural Networks (ANNs):

An Artificial Neural Network (ANN) is a computational model inspired by the architecture and functioning of biological neural networks in the human brain. They consist of interconnected layers of artificial neurons or “nodes”.

Each neuron has multiple inputs and a single output. Each of these inputs have “weight” parameters, and each neuron has a “bias” parameter. For a given input, the output of an ANN is a function of the weights and biases of its constituent neurons. The ANN “learns” the correct weights and biases by changing them to “fit” a set of training examples. The ANN then can predict the results for new inputs based on these weights and biases. Usually, a validation data set distinct from the training data set is used to test the accuracy of the ANN.

The basic idea was to split the images into 100x100 pixel squares, and to pass the entire image as input to the ANN one square at a time. However, considering the tediousness of the task of generating a training data set from real images, an

artificial data set was generated to serve as proof of concept before proceeding. The ANN failed this test, hence prompting us to re-evaluate our idea. Upon further research, it was found that the problem is not as simple as we thought it was, and that current ML-based cell counting algorithms are far more complex than the one we were trying to implement. Hence the efforts were discontinued.

Step 1 - Dataset creation:

We thought creating a data set with the following properties would provide a crude approximation to real-world images:

1. **Varying degrees of background and foreground brightness/contrast:** Since lighting conditions vary considerably between different images, we set the foreground and background brightness to randomly vary within a range of grayscale values.

2. **Varying circularity of cells:** We must be able to adjust the circularity of the cells.

3. **Varying sizes:** Cells may not be of similar size. We must be able to specify a range of cell sizes.

Accounting for all these properties, it was decided that a set of images containing elliptical “cells” varying in size, eccentricity, and brightness and a varying background brightness was suitable. A code to generate a dataset of such images was written, along with the code to “unroll” these images into a suitable input format for the ANN.

Code to generate data set [Python] :

```
from generateEllipses import generateEllipse
import numpy as np

# parameters
IMAGE_SIZE = np.uint(100)
CANVAS_SIZE = IMAGE_SIZE, IMAGE_SIZE, 3
MIN_SIZE = np.uint(2)
MAX_SIZE = np.uint(20)
NUMBER_RANGE = np.uint(20)
MIN_E = 0
MAX_E = 0.9
MAX_BACK_ALPHA = 0.45
MIN_ELLIPSE_ALPHA = 0.65


def generateData(number):

    X = np.zeros((1,IMAGE_SIZE*IMAGE_SIZE))
    y = np.zeros(1)
    for i in range(0,number):
        I, n = generateEllipse(CANVAS_SIZE, MIN_SIZE, MAX_SIZE,
                               NUMBER_RANGE, MIN_E, MAX_E,
                               MAX_BACK_ALPHA, MIN_ELLIPSE_ALPHA)

        I = I[:,:,0]
        I = I/255.0
        I = I.reshape(1,IMAGE_SIZE*IMAGE_SIZE)
        X = np.append(X,I,0)
        y = np.append(y,n)

    np.save('X',X)
    np.save('y',y)

def unrollImage(I):
    I = I*255
    I = np.reshape(I,(IMAGE_SIZE,IMAGE_SIZE))
    img = np.zeros((IMAGE_SIZE,IMAGE_SIZE,3))
    for i in range(0,3):
        img[:,:,i] = I

    img = np.uint8(img)
    return img

generateData(10000)
```

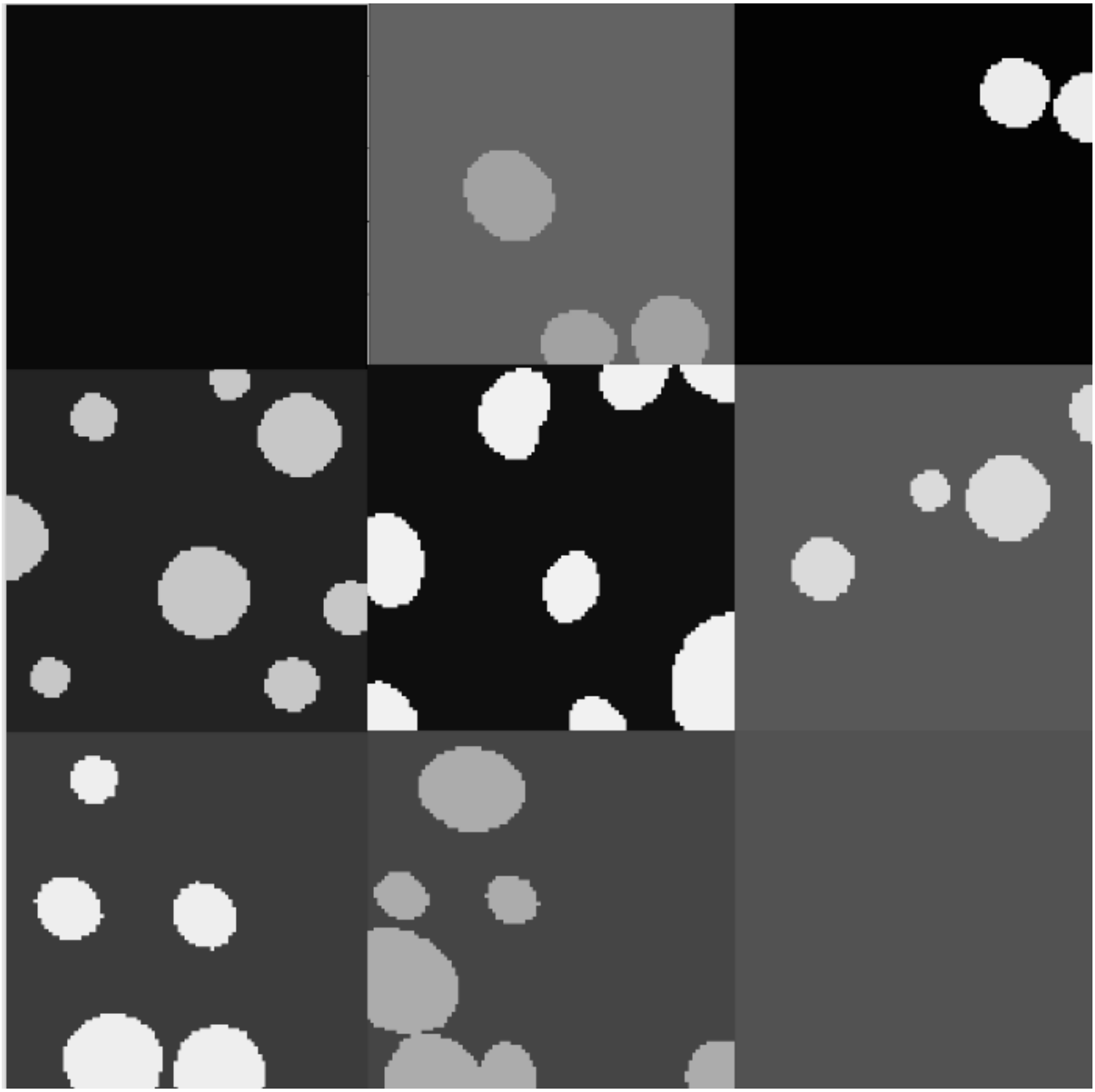
When run, the above code generated the images, unrolls them and saves a numpy array of 10,000 unrolled images as `X.npy` and a numpy array of 10,000 integers as `y.npy`

Code to generate single image [Python] :

```
# Generate a bunch of ellipses
# With varying size,eccentricity,foreground and background brightness

import cv2
import numpy as np
import random
import math
import matplotlib.pyplot as plt

def generateEllipse(canvas_size = (100,100,3),
                    min_size = 5,
                    max_size = 14,
                    number_range = 11,
                    min_e = 0,
                    max_e = 0.7,
                    max_back_alpha = 0.4,
                    min_ellipse_alpha = 0.6):
    img = np.ones(canvas_size, np.uint8)
    back_alpha = random.random() * max_back_alpha
    img = int(math.ceil(back_alpha * 255)) * img

    ellipse_alpha = min_ellipse_alpha + (1 - min_ellipse_alpha) *
random.random()

    n = random.randrange(number_range)
    positions = np.zeros((n,2))
    e = min_e + (max_e - min_e) * random.random()
    height, width, colors = canvas_size

    for i in range(n):
        position = (int(math.floor(random.randrange(width))),
                    int(math.floor(random.randrange(height))))
        positions[i] = position
        major_axis = random.randrange(min_size, max_size)
        minor_axis = int(math.floor(major_axis / math.sqrt(1 -
math.pow(e, 2))))
        axes = major_axis, minor_axis
        color = math.ceil(ellipse_alpha * 255)
        color = color, color, color
        img = cv2.ellipse(img, position, axes,
                          random.randrange(0, 360),
                          0, 360, color, -1)



    return img,n
```

The `generateEllipse()` function generates an image, returns the image and the number of ellipses in the image. A few examples of such generated images are shown here.

Some examples of generated images:



Some examples of generated images. The images show varying foreground and background color.

Step 2 - Build ANN and train it on Data set:

The ANN was coded using Keras in Python runnung on a TensorFlow backend. The generateData() function saves the training data as two numpy files `X.npy` and `y.npy`. The ANN was trained on the 10000 images generated with the `generateData()` function.

Code:

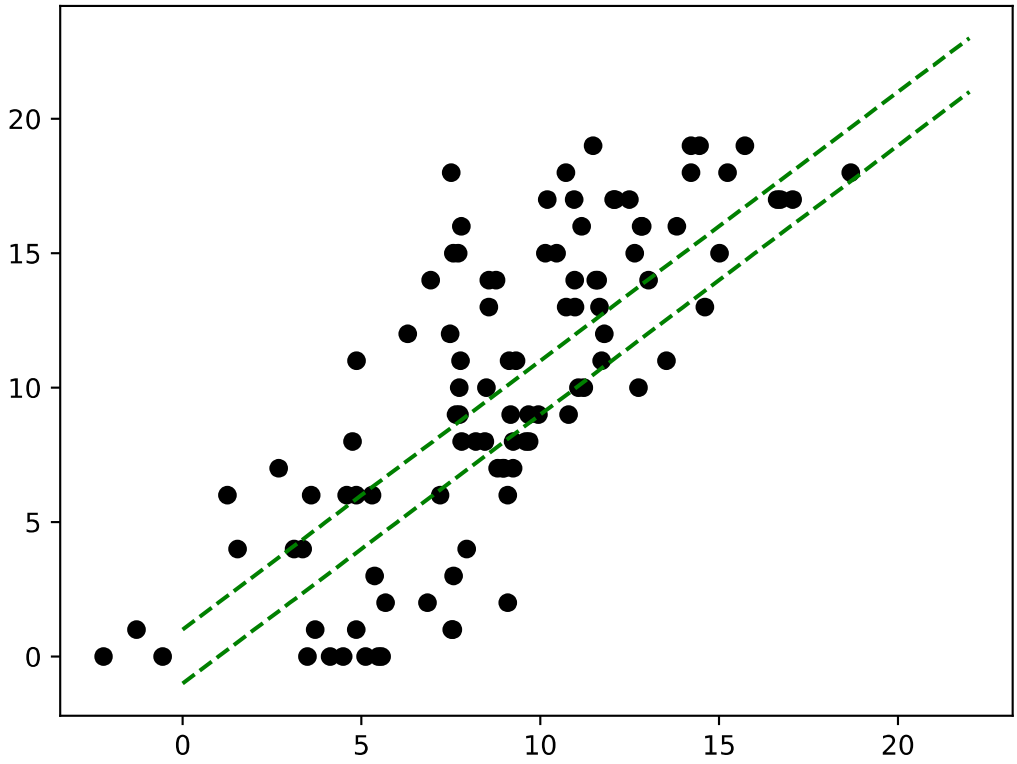
```
from keras.models import Sequential
from keras.layers import Dense, Activation
import h5py

import numpy as np


print "Beginning ANN architecture"

model = Sequential()
model.add(Dense(120, input_dim=10000))
model.add(Activation('sigmoid'))
model.add(Dense(20))
model.add(Activation('sigmoid'))
model.add(Dense(1))
model.compile(optimizer = 'sgd',loss = 'mean_squared_error')

X = np.load('X.npy')
y = np.load('y.npy')

model.fit(X,y)

print "ANN trained. Saving model to file"

model.save("trainedANN.h5")
```

The above shown code is a four-layer implementation. The ANN has one input layer, two hidden layers and an output layer. The above code trains the ANN on the 10,000 images in the training dataset and saves the trained model to a file named `trainedANN.h5`

Step 3 – Test model with test dataset:

The saved model was then tested on a test dataset of 100 images. Analysis of the test results was done by plotting predicted number against actual number of cells on a 2D graph.

Ideally, all points should lie on a 45° line passing through the origin. The correct predictions are the points which lie between the lines of slope 45° with y intercepts 1 and -1.



As we can see from the image, only a small fraction of the predicted results are correct. This exercise has demonstrated that our approach is not compatible with the problem. The model does show some convergence towards the desired result but is simply not complex enough to get significantly correct predictions.

Machine learning approaches to cell counting are a rapidly developing field of research, and this was just our attempt at implementing and testing a simple ML algorithm to a very simple dataset of artificially generated cell images. In reality, cell counting is a much more complex problem, with novel solutions being developed everyday.

We collaborated with teams across our country to get a dataset of hemocytometer images to train our ANN, but we found that the approach we had chalked out was too simple even for simplified generated images of cells, let alone real images of cells under vastly different lighting conditions and varying sizes which come with noise and background discrepancies.

We are extremely thankful to the teams who have helped us in getting the hemocytometer images and we regret that we were not able to use the images.