ECE 69600

Liquid Level Detection using Caffe based Convolutional Neural Network in Neural Compute Stick

PROJECT REPORT

Submitted by:

Raghavan Naresh Sarangapani

0003440222

Under the guidance of:

Dr. Mohamed El-Sharkawy,

Electrical and Computer Engineering,

Indiana University Purdue University Indianapolis

CONTENTS:

1 Acknowledgement 3

2 Abstract 4

3 Image Capture using GoPro 5

3.1 GoPro API's for handling images 6

4 Preprocessing Image and Dataset preparation 6

5 Training images using modified Caffe LeNet 9

6 Testing the network and obtaining accuracy 11

7 Network deployment using Neural Stick 13

8 Complete Setup of Project demo 15

9 Project Flowchart 17

10 Observations: 17

11 Learning outcomes 19

12 Future Enhancements 19

13 Results 20

14 References 22

1 Acknowledgement

I would like to acknowledge the support from the Department of Electrical and Computer Engineering in setting up the Internet of Things Collaboratory Lab at the Purdue School of Engineering and Technology, IUPUI.

I would also like to express my gratitude to Dr. Mohammed El-Sharkawy for his timely guidance during the project and for helping with the resources required for the successful completion of the project.

Finally, I would like to thank my fellow researchers at the IOT Lab, Durvesh Pathak, Akash Gaikwad and Dewant Katare for their valuable guidance in the project.

2 Abstract

The goal of this project is to identify liquid levels on captured images, by using a combination of OpenCV and Convolutional Neural Network for better accuracy. The system uses a GoPro Hero 5 camera and external lighting to capture 12 MP images of a test tube. Using the Wi-Fi support of GoPro, the image is sent to the Rasp Raspberry Pi3 board for further processing. The OpenCV level detection processing is done locally at the Pi board. A modified Caffe LENET network is used to detect levels in the image. The Caffe network training is done using the captured image dataset on a standard Ubuntu PC with a GPU, for faster training times. The low power Vision Processor, Intel Neural Compute Stick is used to run the deployed Caffe Network and identify the levels.

The input image is processed using OpenCV to segment the input image to a Region of Interest comprising of just the test tube and this image is further divided into ten levels to identify if water level is present in the segmented images. Initially, an adaptive Canny Edge detector with variable the threshold, is applied to all the ten sub-segments of the cropped image. The contour output of the edges is pruned to remove any vertical edges as the goal is to identify the horizontal water level. This pruned contour image is used to train the modified LeNet network and will be used to classify between level and non-level images.

Finally, the combination of OpenCV and LeNet is used to identify the segment consisting of the water level. This is done by identifying the image segment with the largest contour and passing it to through the network to identify the presence of a water level. The dataset comprised of 600 test tube images comprising of 60 images for each level, each image was segmented to get a level and non-level image leading to 1200 images. The training: validate ratio was set at 70:30, leading to 840 images to train and 360 images to test LeNet network. With the individual network and accuracy of 68% was achieved. When combined with the OpenCV contour recognition, an accuracy of 92% was achieved.

3 Image Capture using GoPro

The project setup consists of a test tube with water, placed in front of a black back ground and provided with external LED lighting. This project uses a GoPro Hero 5 camera to capture 12MP images for level detection. The Raspberry Pi 3 connects to the GoPro Wi-Fi and the frame is grabbed using goprocam python api's.

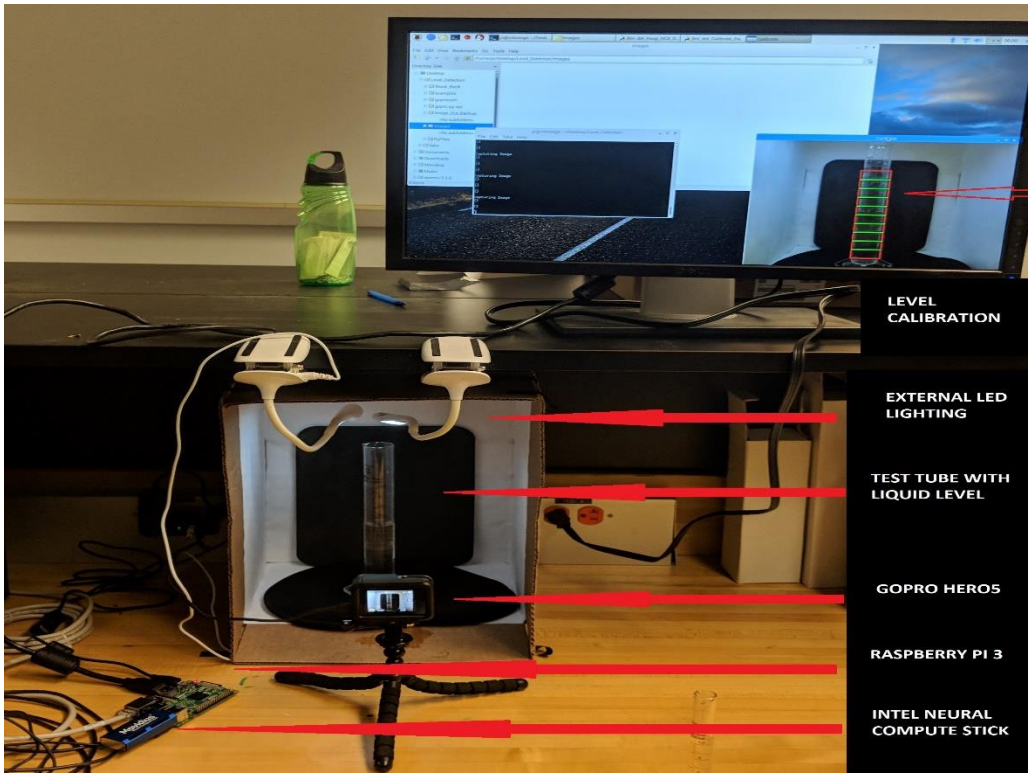


Figure 1: Project Setup

To install the goprocam API's use the following command in Ubuntu:

```
pip install goprocam
```

Further information of the API's can be found at the following repository:

https://github.com/KonradIT/gopro-py-api

The camera is set to 12 MP Narrow mode to capture the test tube with limited field of view. Initially calibration is to be performed to align the image with the GoPro.

3.1 GoPro API's for handling images

The following set of API's are used to handle the GoPro images:

1. The GoProCamera control is initiated and a handle is received after initialization. This handle will be used to process further control of the camera.

```
gpCam = GoProCamera.GoPro()
```

2. An image grab is initiated using the take photo API and the corresponding image URL is obtained.

```
gpCam.take_photo(0)
photo_url = gpCam.getMedia()
```

3. From the URL the actual image is grabbed and converted to NumPy for further processing.

```
url = urllib.request.urlopen(photo_url)
photo = np.array(bytearray(url.read()), dtype=np.uint8)
input = cv2.imdecode(photo, -1)
```

4. The last image is deleted from the GoPro.

```
gpCam.delete("last")
```

4 Preprocessing Image and Dataset preparation

The Image dataset is prepared by using the following methods.

1. The project setup is done by using similar conditions like position of camera, test tube and lighting conditions.
2. For each water level 30 images are taken in low light and other 30 images are taken using good lighting conditions.
3. The final dataset consists of 600 images, with 60 images for each level. The field of view is fixed, and the levels are updated by adding water as shown in Figure 2.
4. The images are saved in folder with the level name, from which they can read again with the appropriate levels for training and testing datasets.

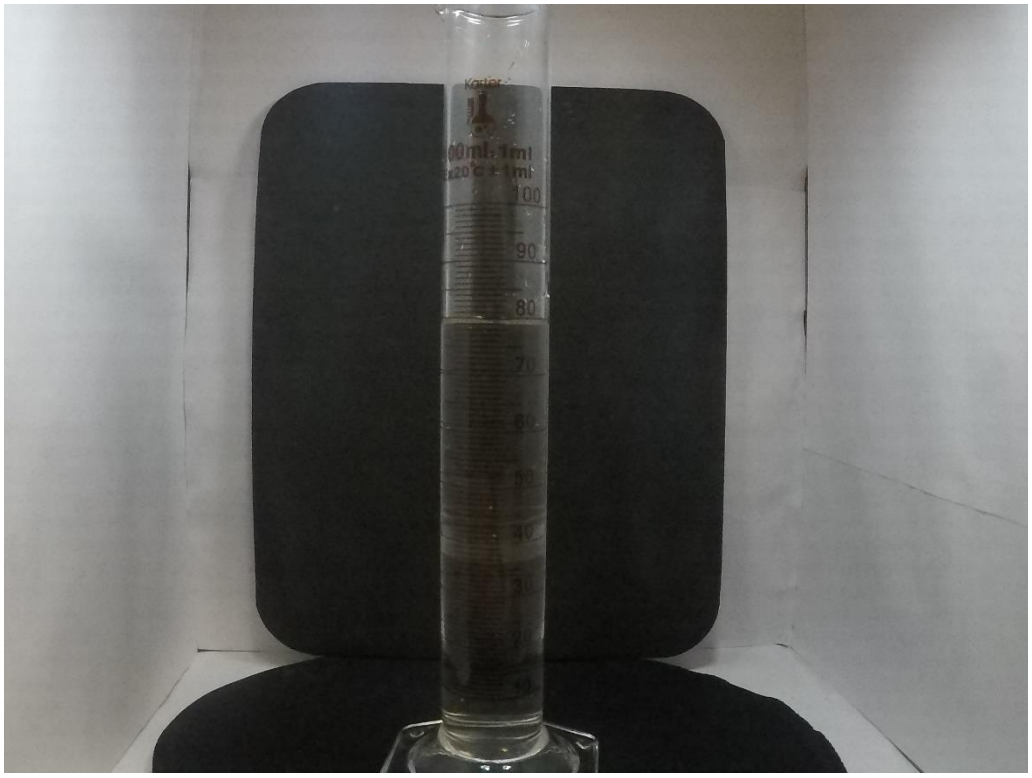


Figure 2: Dataset Image

5. Each image is cropped at the center (bound by the red box in Figure 3) to select the ROI.

```
cropped = input[690:2790, 1685:2145] # Black Image crop
```

Figure 3: Cropped levels

6. The cropped image is converted to a grayscale image and a uniform gaussian blur is added to the image. This is done in accordance with steps to apply a Canny Edge detector.
7. From each of the cropped image two types of image is obtained:
    a. One image with the water level present, identified by the folder it is stored.
    b. One non-level image obtained randomly from the remaining 9 levels.
8. This is done to maintain equal number of level and non-level images in the training and test datasets.
9. Next the images are passed through an adaptive Canny Edge detector. In this the lower and upper thresholds are set by calculating the median of the segmented image.

```
def auto_canny(image, sigma=0.33):
    v = np.median(image)
    lower = int(max(0, (1.0 - sigma) * v))
    upper = int(min(255, (1.0 + sigma) * v))
    edged = cv2.Canny(image, lower, upper)
    return edged
```

10. The edged output obtained from the previous level is used to find contours in the image. In this step only, the external contours are found.
11. In the contours obtained, only the horizontal contours are taken because they represent the liquid level better than vertical contours. In these horizontal contours a region of interest comprising of the midsection of the cropped image is chosen as the water level appears in that region.

```
def draw_cnt(image):
    cont_out = np.zeros((210,460,1),np.uint8)
    contours = cv2.findContours(image, cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
    if len(contours) != 0:
        for c in contours:
            x,y,w,h = cv2.boundingRect(c)
```

```
            if(w>h and (y%210)>= 55 and (y%210) <= 155):
                cv2.drawContours(cont_out, c, -1, 255, 3)
    return cont_out
```

12. Next both the images are resized to a resolution of 227x227 which is suitable for the LeNet.

Figure 4: Level image(left) and non-level image(right), used in network

13. Finally, these images are moved to the corresponding folder with the following structure:
    - Training: Folder contains 840 images (420 level and 420 non-level)
        - Level: folder contains 420 level segmented images for training
        - Non-level: folder contains 420 non-level segmented images for training
    - Training: Folder contains 360 images (180 level and 180 non-level)
        - Level: folder contains 180 level segmented images for training
        - Non-level: folder contains 180 non-level segmented images for training
14. The ratio of 42:18 is maintained for each level to get 70:30 Training Validation ratio.

5 Training images using modified Caffe LeNet

The training phase of the network is done using a modified version of the Caffe LeNet. The following steps are followed:

1. The images from the previous round of pre-processing are mixed randomly by level and non-level images separately in the Training and Testing folders. This is done to prevent the network from over-fitting based on a file pattern.
2. The following folder structure is followed:

- Training: Folder contains 840 images (level, non-level mixed randomly)
- Validation: Folder contains 360 images (level, non-level mixed randomly)

3. During this creation, the filenames of the images in Training and Validation folders are saved in separate train and test txt files. These txt files have the values of 1 for level and 2 non-level with each filename, the network uses them to correct the weights. Both the files are used in the LMDB creation.
4. Next the LMDB file required to train the Caffe network is obtained by using the create_imagenet.sh script. The appropriate path to the training and validation images should be set in this script. This leads to the generation of two LDMB files, one for training and other for validation.
5. The solver.prototxt file is set to the required solver parameters. In this project a Gradient descent model is used.
6. Next the network model for training is designed. Initially the basic LeNet was found to give lesser accuracy of 50% and hence with further understanding, the LeNet was modified (as shown in Figure 5) by using more ReLU layers and a single drop layers to improve the accuracy to 62%.

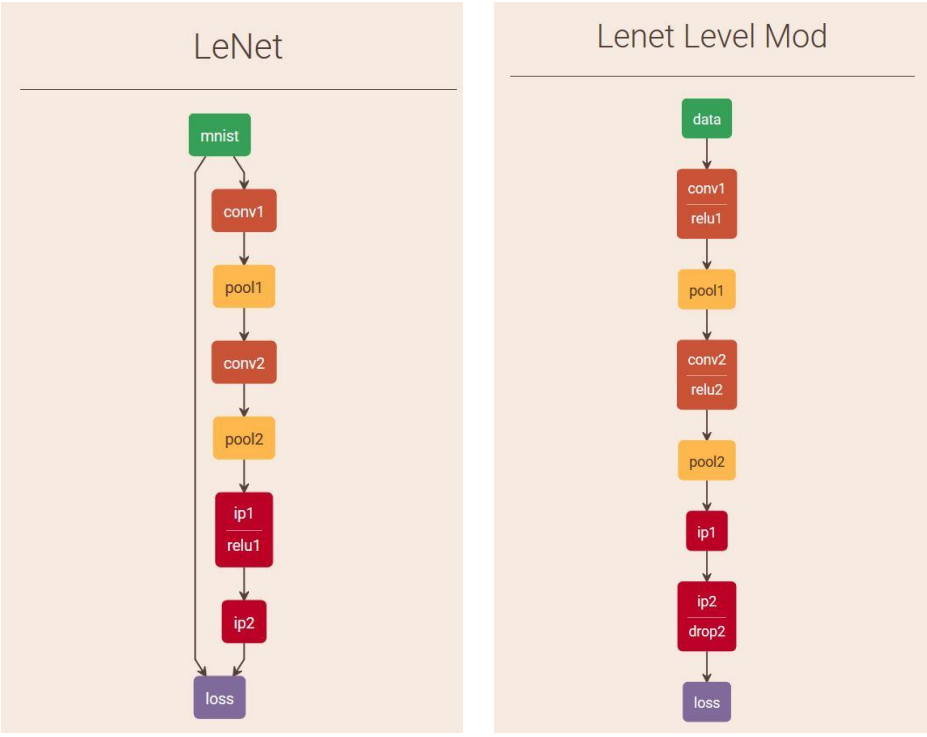


Figure 5: Comparison of original with modified LeNet

7. Using this network, the training is started by using the script train_caffenet.sh. The number of iterations in the Solver file is set to 10000 with the ability to capture the model and solver file at every 5000 iterations. This process takes some time depending on the number of input images.
8. Finally, the trained Caffe model is derived. Using this we can test the network using validation images to measure the accuracy of the model.

6 Testing the network and obtaining accuracy

Once the model has been trained the following steps are followed to test the network:

1. To test the Caffe network we need the following files:
    a. Deploy.prototxt: which contains the network model for testing.
    b. Caffemodel: the file Caffe model weights file obtained after training.
    c. Validation images:
    d. Validation file names with level values.
2. Initially the Caffe mode is set to GPU for faster processing times and the network is loaded using the following deploy model. In the deploy model (as shown in Figure 6) the drop layer is removed as there is no need to drop the weights.

```
caffe.set_mode_gpu()
net = caffe.Net(Caffe_proto_path,Caffe_model_path, caffe.TEST)
[(k, v.data.shape) for k, v in net.blobs.items()]
[(k, v[0].data.shape, v[1].data.shape) for k, v in
net.params.items()]
```

3. Using the test_file.txt as reference the files are opened and passed through the network in the form of blobs.

```
File_path = Image_path+File_name
im = np.array(Image.open(File_path))
im_input = im[np.newaxis, np.newaxis, :, :]
net.blobs['data'].reshape(*im_input.shape)
net.blobs['data'].data[...] = im_input
```

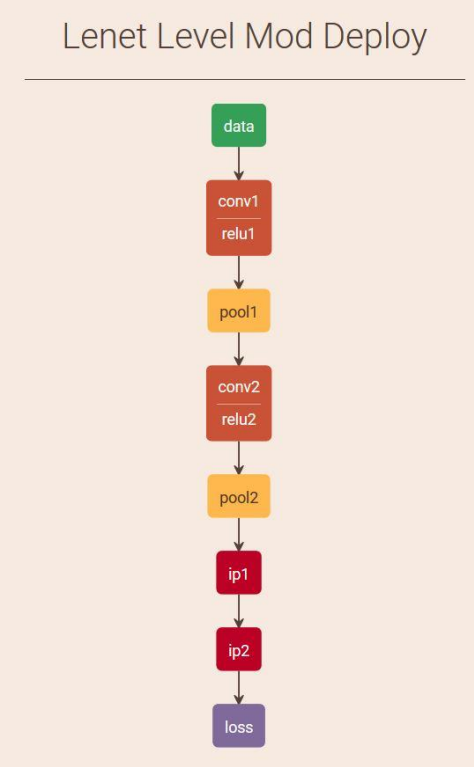


Figure 6: LeNet Deploy Model

4. The network is forwarded with the input image and the probabilities of the level and non-level classifications are used to tell if a level is present or not.

```
res = net.forward()
data = res['loss']
if((data[0,0] == 0) and (data[0,1] == 0)):
      Lev_found = 2
elif (data[0,0]<data[0,1]):
      #print ("level not found")
      Lev_found = 2
else:
      #print ("level found")
      Lev_found = 1
```

5. The files in which the levels were found and those in which were missed are kept in record separately to analyze the failure.

6. A final summary is done to calculate the overall accuracy based on number of correct predictions divided by the total number of images.
7. The failed images can be used for further learning.

7 Network deployment using Neural Stick

The Intel Neural Compute stick has a Visual Parallel Processor to facilitate faster Network deployment and classification. Through the USB interface it is connected to the Raspberry Pi and can be accessed provided by the Movidius ncsdk. The stick does not support the training procedure and hence that is done in the desktop. The stick has support for deploying Caffe and Tensor flow based networks. The modified network is deployed to the Neural compute stick using the following procedures.

1. The setup requires a desktop to profile the generated Caffe model and a raspberry pi to deploy the network. The Neural Compute SDK and tools should be installed in both locations using the link 1.
2. Next the trained Caffe model is profiled in the desktop using the NC profiling tool. This process takes the Caffe model and the deploy network prototxt as inputs and produces a graph output that is suitable for porting the network to the stick and an analysis of the network performance in the stick. To initiate this the following command is used, further information can be obtained from the link 2.

```
mvNCProfile -s 12 deploy.prototxt -w Trained_weights.caffemodel
```

3. This step produces the graph output and the analysis of the modified network as shown in Figure 7. The analysis shows the processing time of the Stick in each block.
4. The final graph file should be copied to the Raspberry PI to be deployed in the connected Neural compute stick.
5. In the Rpi , the stick device is first identified and paired with using the following code.

```
mvnc.SetGlobalOption(mvnc.GlobalOption.LOG_LEVEL, 2)
devices = mvnc.EnumerateDevices()
if len(devices) == 0:
      print('No devices found')
      quit()
```

```
device = mvnc.Device(devices[0])
device.OpenDevice()
```

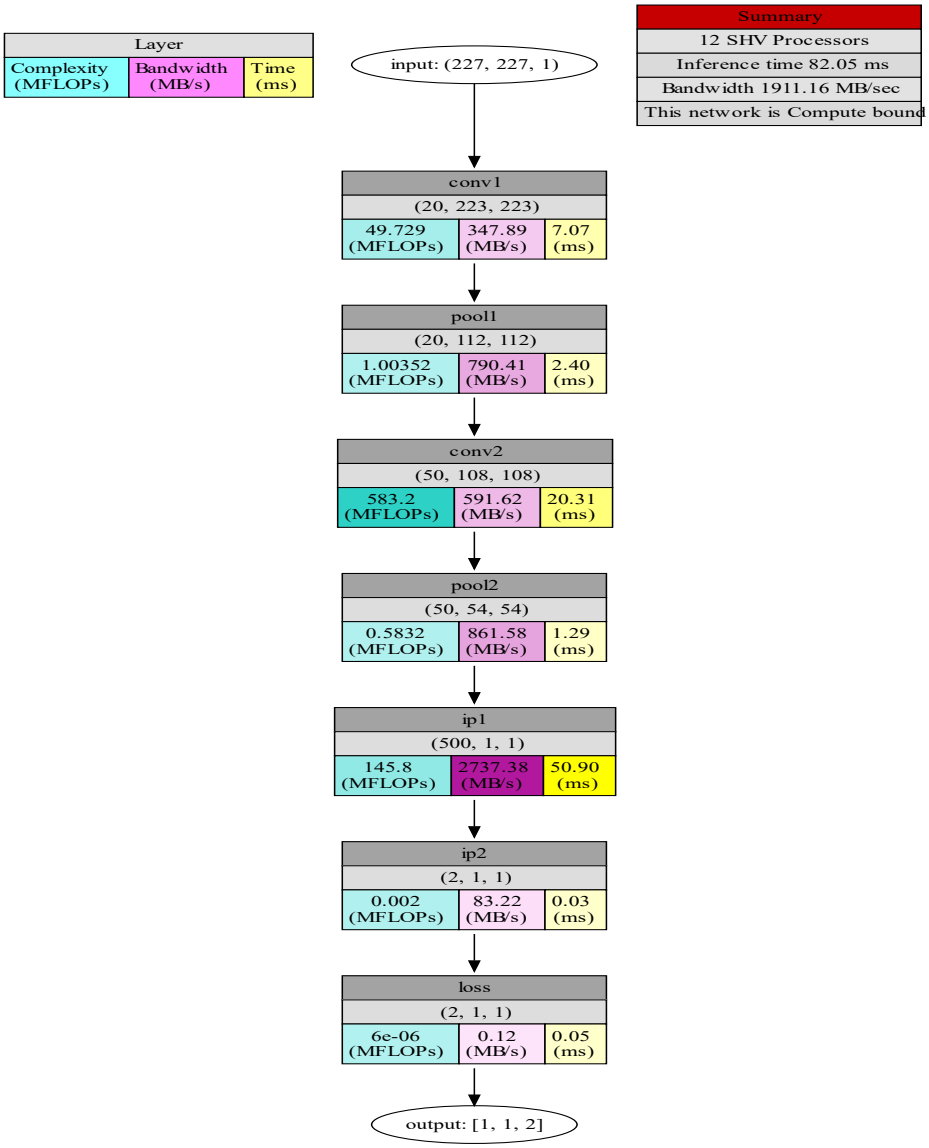


Figure 7: Network processing time analysis

6. Next the corresponding graph file is read and allocated as blobs to the stick.

```
with open( Graph_path, mode='rb' ) as f:
    blob = f.read()
graph = device.AllocateGraph(blob)
```

7. Next the processed image can be input to the network and the level output can be obtained by using the NCS api's.

```
image = image.astype(np.float32)
graph.LoadTensor(image.astype(np.float16), 'user object')
data, userobj = graph.GetResult()
```

8. The previous step can be repeated multiple times by loading new images to the stick.

8 Complete Setup of Project demo

Once all the previous processes are done, the system is ready to be deployed and tested in real-time. The following steps should be followed for the same.

1. Setup the test-tube with water and appropriate lighting conditions.
2. Power up the GoPro camera and mount it on rigid stand.
3. Enable the WIFI mode in GoPro and test by connecting the Raspberry Pi 3 using the SSID and password.
4. Run the code lev_det_Calibrate_Rasp_NCS_GP.py to calibrate the setup.
5. The Calibration process is a onetime setup and encloses the following steps.
    a. An image is grabbed in the GoPro.
    b. This image is grabbed by the R-pi and sent for further processing.
    c. Bounding boxes are drawn on the image using the rectangle and line functions of OpenCV. The red bounding box is for the test tube and line function is used to mark the level separations. The image must be calibrated to fit in these boxes.

```
cv2.rectangle(input_image,(x1,y1),(x2,y2),Red_RGB_val,Thickn
ess)
    for level in range(1,10):
        cv2.line(input_image,(x1,y1_level ),(x2,y2_level),
                 Green_RGB_val,Thickness)
```

    d. The output image is displayed in the screen and is refreshed every few seconds.
    e. The test tube is adjusted to position the levels properly within the marked edges, once done the setup should not be disturbed.

6. Once the calibration is over the level can be detected by running the python script lev_der_Rasp_NCS_GP.py.
7. In this the NCS is first initiated, followed by loading the deployed network graph to it.
8. An Image is grabbed from the GoPro camera every 20 seconds and the OpenCV pre-processing steps followed in training phase are done on the input image.
9. Each of the level is cropped and passed through an adaptive canny edge detector and the horizontal contours are drawn in the given ROI.
10. In this stage for each image the number of contours of each level is stored in a list.

```
cnt_len_list = []
ret_cnt_len = 0
for level in range(1,11):
      output = blur_image[((10-level)*210):(((10-
      level)*210)+210),0:460]
      auto_edge = auto_canny(output)
      cont_out,ret_cnt_len = draw_cnt(auto_edge)
      cnt_len_list.append([ret_cnt_len,level])
```

11. This contour + level array is sorted to identify the image with largest contour, which with good level of accuracy corresponds to the level detected image.

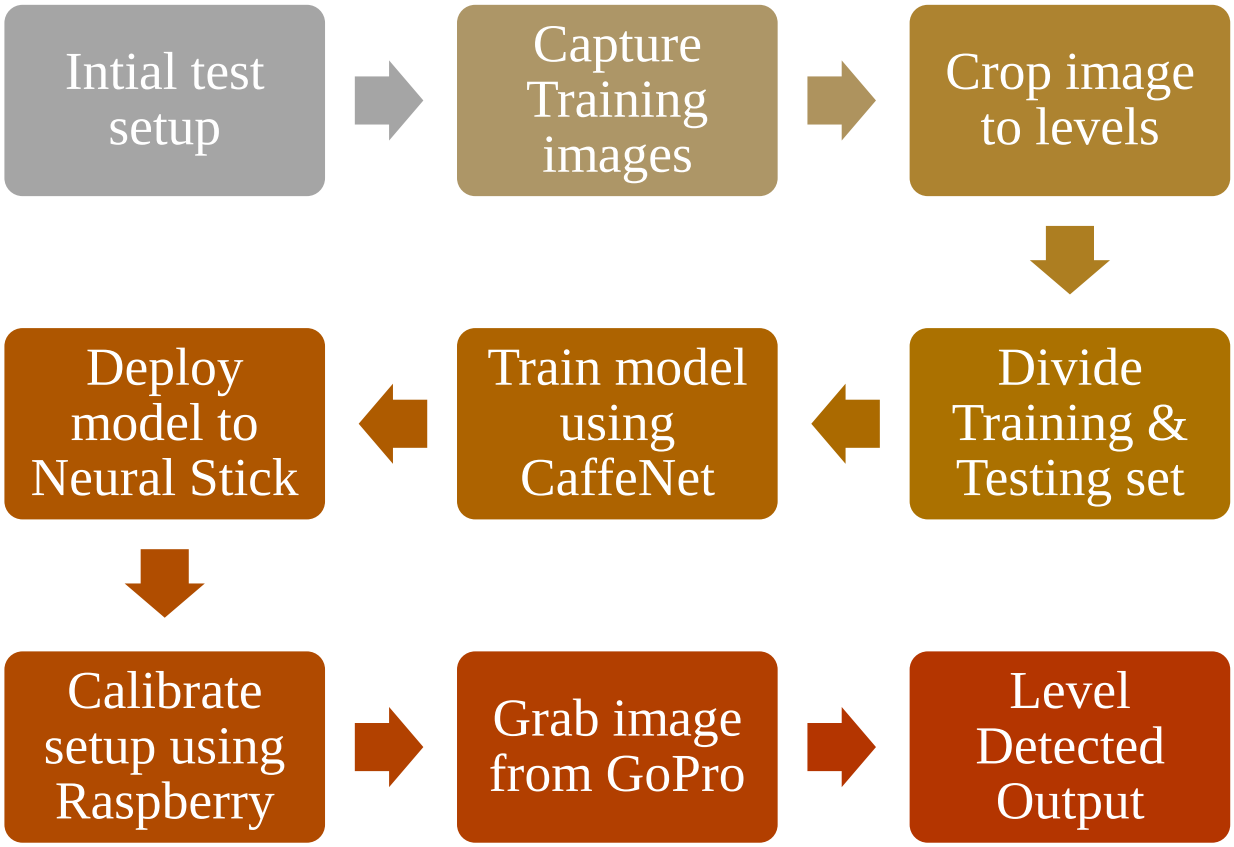
```
cnt_len_sort = np.asarray(cnt_len_list)
cnt_len_sort =
cnt_len_sort[np.lexsort(np.fliplr(cnt_len_sort).T)]
cv_level = cnt_len_sort[9,1]
```

12. The largest contour image is passed through the network to verify if the level is present or not.
13. This combined method of level identification using OpenCV and Neural network products an accuracy of 92%.
14. Finally, the appropriate level is marked in a copy of the input image and stored back in the Raspberry Pi.

```
cv2.rectangle(input_image,(1685,(690+((10-
      cv_level)*210))),(2145,(690+((10-
            cv_level)*210)+210)),(0,0,255),10)
```

```
Text_out = "Water Level: "+str(cv_level*10)
cv2.putText(input_image,Text_out,(2145+20,(690+((10-
      cv_level)*210)+210)),cv2.FONT_HERSHEY_SIMPLEX,3,(0,255,0),10
           , cv2.LINE_AA)
Output_FIlename = Root_path + "Images/" +filename_str+".JPG"
cv2.imwrite(Output_FIlename,input_image)
```

9 Project Flowchart



10 Observations:

The following observations were made during the development of the project.

1. Initially the images were taken by random positions of the test tube which made generalizing of images difficult. Then the image dataset was generated by using a fixed position and multiple lighting.

2. Another dataset was created using a white background for the test tube. In this case the Canny edge detector did not yield the edges for the water level.
3. To identify the water level a HSV based thresholding method was tried but the results were not satisfactory.
4. The black background image was used to train the SqueezeNet Caffe net, but due to the poor quality of the image and lack of distinct features, the SqueezeNet did not offer good results. Another reason for this be the multiple convolution layers of it.
5. A total of 5 datasets were tried to train both Squeezenet and LeNet , the best performance was obtained in the Canny edge output.
    a. Black background images with increased gain.
    b. White background images.
    c. Canny edge contour output.
    d. Edge amplified output obtained by using black images with Canny edge as overlay.
    e. Combination of multiple thresholds of Canny edge images.
6. Initial LeNet offered around 50% accuracy. The modified network was formed by removing the ReLU activation layer after the inner product and adding one after each convolution before max pooling. Further accuracy was achieved by adding a Drop Layer to drop the weights after each iteration.
7. Further attempts to increase the accuracy were futile. Methods like adding more ReLU layers, adding more IP layers to reduce the number of outputs gradually, increasing number of iterations, re-training of images using same network and increasing number of drop layers did not help the accuracy.
8. The GoPro live preview mode to the PC was not stable and hence a continuous method system was established for the calibration.
9. It was found during testing that without the appropriate light conditions the system fails.
10. In some cases, the contour method was wrong, but the network predicted correctly.
11. Another method to identify the level with largest probability using the network did not provide satisfactory results.
12. The processing time in the system is mainly through the OpenCV processing in the Raspberry pi.

11 Learning outcomes

By working on this project as part of ECE 69600 course, I had the opportunity to learn the following topics:

1. Training a Caffe Network.
2. Pruning of Caffe network to achieve better results.
3. Deploying Caffe network in Intel Neural Compute Stick.
4. Accessing images from GoPro camera.
5. Performing OpenCV processing for edge detection.

12 Future Enhancements

The following future enhancements can be done in the future to ensure better performance.

1. To avoid initial calibration, OpenCV processing can be used to identify the shape of the test tube for cropping the levels.
2. Other networks can be explored to improve the performance.
3. Reducing the OpenCV processing to be done as it slows down the Raspberry Pi.
4. Using GoPro video for real time level detection.
5. Grabbing multiple images using more test-tube patterns to improve model accuracy.
6. Using the Vertical orientation to get 4000 pixels of accuracy on the test tube image instead of 3000 currently obtained.

13 Results

The following results were obtained in the project:

1. The standalone trained network produced an accuracy of 61.94 % based on a test set of 360 images of level and non-level combinations.

```
Summary of Files
####################

Root path - /home/iot_lab_dnn/Level_Detection_Files/
Image path - /home/iot_lab_dnn/Level_Detection_Files/
Canny/Level_Det_Final_Images_Canny/Validate/
Text files path - /home/iot_lab_dnn/Level_Detection_Files/Canny/
Image input train file - lev_det_test_Canny.txt
Image HIT file - Canny_test_Hit.txt
Format: Filename-Train Level-Found Level
Image MISS file - Canny_test_Miss.txt
Format: Filename-Train Level-Found Level-Res out


Number of Images = 360
Hit num = 223
Hit rate = 61.94
Miss num = 137
Miss rate = 38.06
```

Figure 8: Trained Network Accuracy

2. The final model which is combination of OpenCV contour-based decision and Trained network-based verification provided an accuracy of 91.67 % with a test set of 180 level and non-level combination images. In this case it was found that the average time to process a single image was 830.83 milli-seconds out of which Image grabbing and OpenCV took 817.2 milli seconds and the Neural compute stick took 6.82 milli seconds to give a decision. This proves the advantage of using the Neural compute stick to deploy networks when using low compute embedded devices like Raspberry Pi.

```
Summary of Files
####################

Root path - /home/pi/Desktop/Level_Detection/
Input Image path - /home/pi/Desktop/Level_Detection/Black_Back/
Image input train file - Test_Black_BB.txt
Image HIT file - Test_Black_BB_Net_CV_Hit.txt
Format: Filename-Train Act_Lev Fnd_Lev Net_Out Res_out
Image MISS file - Test_Black_BB_Net_CV_Miss.txt
Format: Filename-Train Act_Lev Fnd_Lev Net_Out Res_out


Number of Images = 180
Hit num = 165
Hit rate = 91.67
Miss num = 15
Miss rate = 8.33


Total process Time = 149.54999 seconds
Average process Time = 830.83 milli-seconds
Total OpenCV Time = 147.095204 seconds
Average OpenCV Time = 817.2 milli-seconds
Total Network Time = 1.227393 seconds
Average Network Time = 6.82 milli-seconds
```

Figure 9: Final Model Accuracy

14 References

I would like to acknowledge the following References for providing valuable information:

1. https://movidius.github.io/ncsdk/
2. https://movidius.github.io/blog/deploying-custom-caffe-models/
3. https://github.com/KonradIT/gopro-py-api
4. https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet_train_test.prototxt
5. http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe/
6. https://www.pyimagesearch.com/2015/04/06/zero-parameter-automatic-canny-edge-detection-with-python-and-opencv/