CROWD DENSITY ESTIMATION USING DEEP LEARNING

EE 528 - DEEP LEARNING FOR VIDEO SURVIELLANCE SYSTEMS

Project by:

Jayesh Gupta 1701EE16

Karhad Raman Vijayrao 1701EE17

Under the guidance of:

Dr. Mahesh Kumar Kolekar

Department of Electrical Engineering,

Indian Institute of Technology, Patna

ABSTRACT

As a powerful method for crowd manipulation and management, crowd density estimation is an essential subject of matter in AI applications. Since the prevailing strategies are tough to meet the accuracy and speeds necessary for engineering and practical applications, we have proposed to estimate crowd density through an optimized convolutional neural community (ConvNet). The work is done in two steps: first, convolutional neural network is implemented for crowd density estimation. The estimation speed is drastically increased through eliminating few network connections. Second, a cascade of Convolutional Net classifier has been used, which improves the accuracies and speed. The method is tested on three data sets: PETS_2009, a Subway image sequence and a ground truth image sequence. Experiments confirm the good performance of the method on the same data sets compared with the state of the art works.

TABLE OF CONTENTS

1. Introduction

Nowadays intelligent surveillance has been widely used in fire detection, vehicle identification, crowd management and so on. Crowd density estimation which is a crucial part of crowd management has become more and more important for the happening of stampede. In general, to estimate the crowd density, crowd features need to be designed first and then a classifier needs to be trained to discriminate the crowd density by these features. In the past few decades, many efforts have been made to find out both a better way for crowd feature description and a more efficient classifier for classification. Previously crowd estimation was done using image processing, using background removal and edge-detection for stationary crowd estimation and optical flow computation for the estimation of crowd motion. However, these methods get 80% accuracy in average. To enhance the estimation performance, the support vector machine (SVM) was generally adopted. But, because the used features are not designed for the special practical environments, these SVM based methods also cannot obtain satisfactory results in applications. Meanwhile, artificial neural network, which is another popular and efficient approach, has been widely used. Because it discriminates features more precisely like human neuron network. Recently, convolutional neural network has become research focus again for its three brilliant ideas of local receptive fields, shared weights and spatial sub-sampling. Besides, ConvNet can directly process images and learn features because of the multiple convolutional layers whose convolutional weights are trained by the back propagation algorithms. For these advantages, ConvNets were widely applied into different fields such as document recognition, traffic sign recognition, face detection. The standard ConvNet is made up by multiple layers. As the layer goes up, the distortion will get down. So some details of the image will probably be lost for the reason that only the feature maps in the last sub-sampling layers are accepted by the nets. To get better performance for shift, scale and distortion invariance, the multi-stage ConvNet has been adopted. Besides, considering practical engineering application, some important improvements are made for the speed and accuracy.

The main contribution contains two aspects. Firstly, convolutional network is optimized by integrating similar feature maps, so as to decrease the computation cost and accelerate the speed of both training and detection stages. Secondly, a cascade of two classifiers is also designed. The first one picks out samples which are obviously misclassified, and the second one, which is trained especially for hard samples, will reclassify those rejected samples. By this strategy, the accuracy is improved after handling hard samples more precisely

2. Design Analysis

Crowd density estimation is mainly used in public places such as train station, subway station and the square. In these places, from the video, people always overlap when it is crowded and sometimes only their heads protrude. So the method to estimate crowd density by analyzing the individuals to count their numbers is out of the ability. Through years of researches, a general procedure of crowd estimation has been established. The first step is to fi nd out an effective way to extract crowd features of different density levels and the second step is to discriminate these features with a classification model. According to the observation that textures are becoming more distinctive with the growing of crowd, texture analysis has been chosen many times and shows good performance. So, in this project, texture is also used by the convolutional neural network. Generally, the neural network is a mechanism for machine learning which is constituted of an input layer, multiple hide layers and an output layer. Each layer is made up by several neuron nodes and the output of one node from the layer in the front connects to the input of a node from the layer in the next. Each connection carries a weight which indicate s the influence of one node to another. Generally, the nodes in the front layers are full connected, which is known as full connection, to every node in the next layer. With the desired outputs, the network can learn the optimal solution of a special probl em by adjusting the weights of connections. In this project, a special type of neural network has been applied: the convolutional neural network which brings us the benefits of feature learning. Multi-stage convolutional neural network Unlike

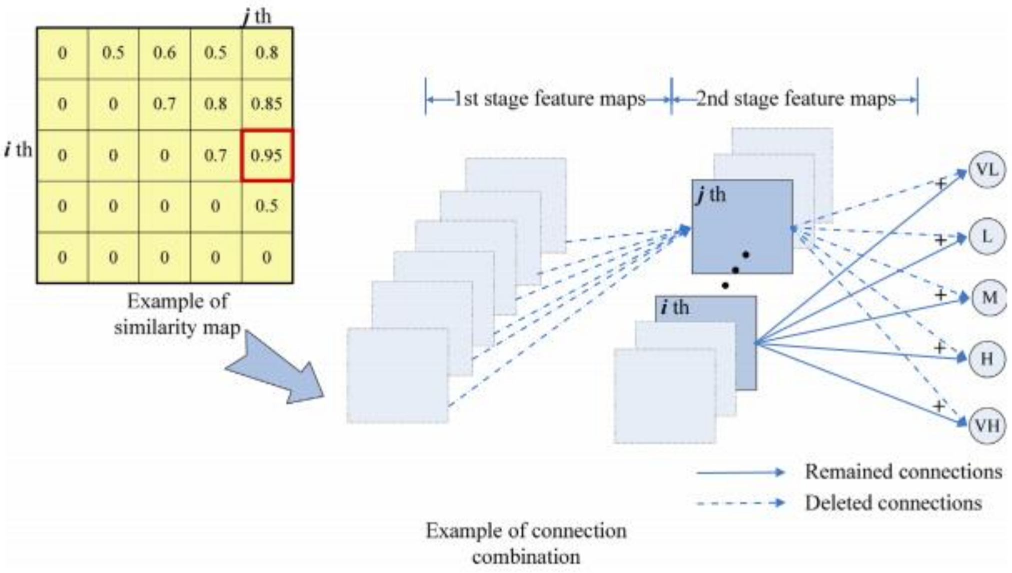


Fig 1. The similarity matrix and connection deletion methodology

in the general neural network, the neurons in ConvNet are organized as feature maps in each layer. Benefiting from the convolution operation, images can directly be used as the input and feature maps are acquired. Several feature maps constitute the convolutional layers. A feature map represents a special kind of feature by convoluting with different kernels. A kernel is a filter which is a small set of connection weights in general neural network.

The multi-stage ConvNet increases the quantity of features in the final classifier as well as the connections. This seriously increases the computation time at both training and detection. stages, which affects its engineering applications. Inspired by the observation that there exists some redundant connections among two similar feature maps, we have proposed an optimization method, whose efficiency is confirmed in experimentation to move out these extra connections based on similarity to accelerate the speed. once the similarity between two maps is confirmed to satisfy the condition, one of them is randomly chosen to be deleted together with the connections. The weights of the connections from the deleted map are added up to the kept ones before being removed. This is to avoid causing instability and oscillations in networks. After these processing, the network is simplified which contains only half or less connections according to experiments. As a result, speed is significantly accelerated.

Once the similarity between two maps is confirmed to satisfy the condition, one of them is randomly chosen to be deleted together with the connections. Similarity matrix M is derived by,

$$M = \left(\sum_{i=1}^{N} M_k \right) \Big/ N,$$

where N is the total number of samples, and M_k is the similarity measure for one sample, which is given by cosine similarity between the map A_i and A_j,

$$M_k(i,j) = sum(A_i.{*}A_j)/(\|A_i\| \times \|A_j\|)$$
$$\text{for all } 1 \le i,j \le dim(M_k).$$

Hard samples have always been a problem for learning. In our work, hard samples mean those samples with very complicate backgrounds which are hard to fit. With the existences of these samples, sometimes it will be hard for the network to converge. So picking out those hard samples and training them individually is a good choice. Inspired by the idea of boosting algorithm, two optimized multi-stage ConvNets are cascaded as a strong classifier to discriminate sample crowd densities.

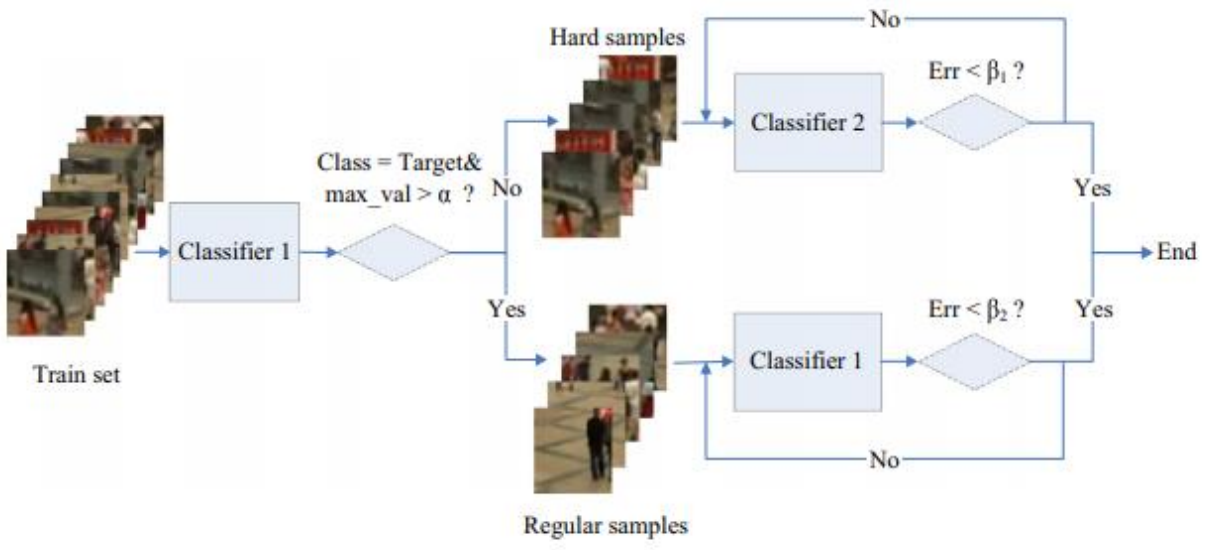


Fig 2. The final Classifier architecture

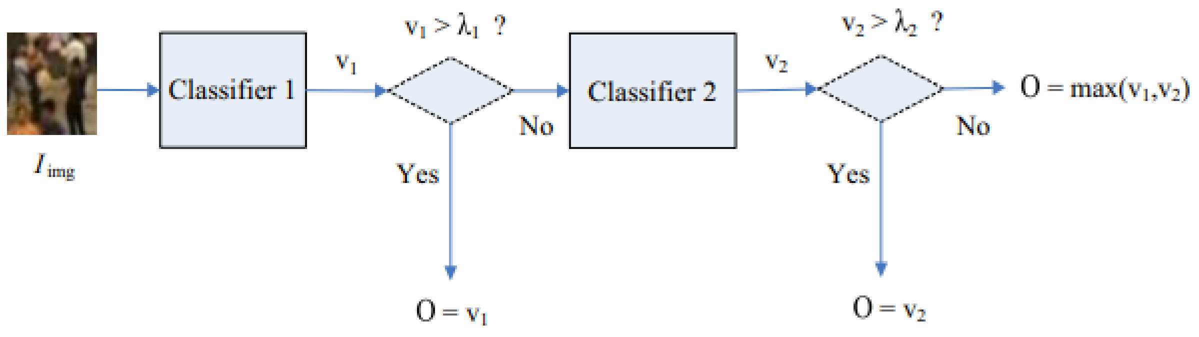


Fig 3. The test classification architecture

Hard sample is picked out by classifier 1 with the condition that the max output component is below α or the class label does not match its labeled category. Because the hard sample among which the max output component is below α is more close to the classi fication boundary, it will be more likely to interfere with the training. After separating hard samples and regular samples, both of the classifiers are trained individually until the network errors are below the thresholds. Experimentation show that this strategy of training enhances the performance. For detection, the crowd density level of an image is discriminated first by classifier 1. If the max component of the output does not satisfy a threshold, this sample could be considered as a hard sample. So it should be sent to classifier 2 for reclassification.

3. Design Analysis

The following is the code for the given project with sufficient comments to explain specific things

Code:

```
#################################################################################
import numpy as np
import cv2
import pandas as pd
import random
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt

from google.colab import drive
drive.mount('/content/gdrive')

!unzip /content/gdrive/MyDrive/Resized.zip
df = pd.read_csv('/content/gdrive/MyDrive/Category.csv')

coding = {'VL' : 0,
          'L' : 1,
          'M' : 2,
          'H' : 3,
          'VH' : 4}

img_data = []
label = []

for i in range(221):
  img_data.append(cv2.imread('/content/Resized/image_' + str(i)+'.jpg', cv
2.IMREAD_COLOR).astype('float32')/255)
  label.append(coding[df.iloc[i,1]])


temp = list(zip(img_data, label))
random.shuffle(temp)
img_data, label = zip(*temp)

x_train = img_data[:110]
y_train = label[:110]
```

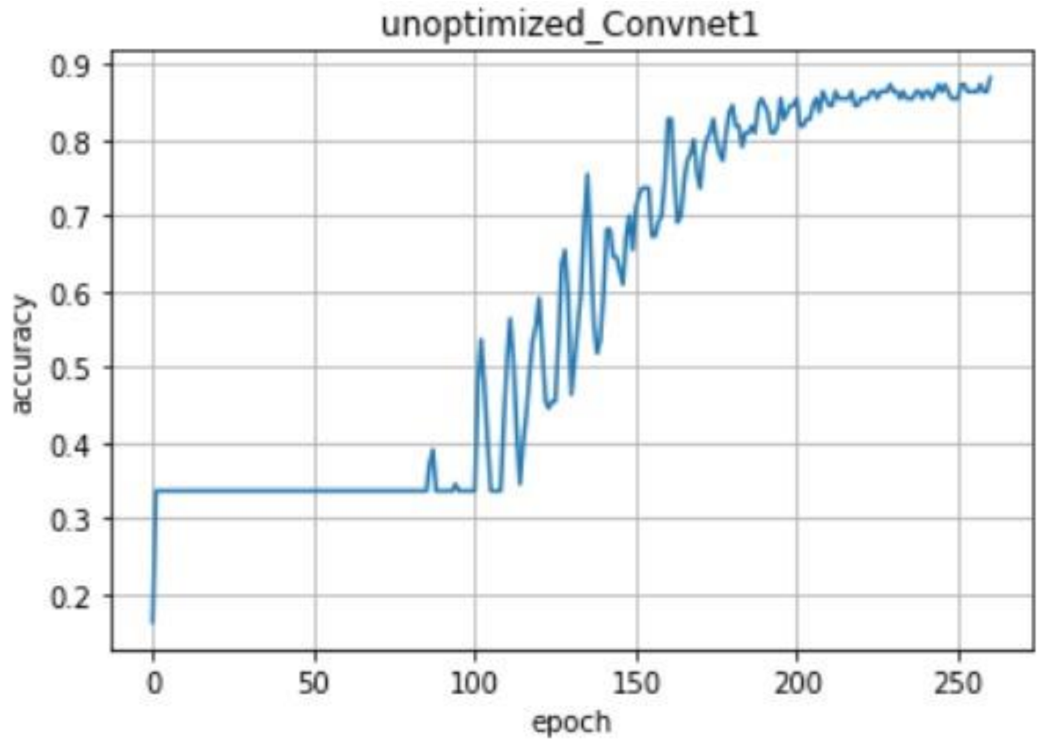
```
x_validate = img_data[110:165]
y_validate = label[110:165]
x_test = img_data[165:]
y_test = label[165:]


#x_train = np.array(x_train).astype('float32')/255.0
#x_validate = np.array(x_validate).astype('float32')/255.0
#x_test = np.array(x_test).astype('float32')/255.0


def my_model(input_shape,
             filter1,
             kernel1_shape,
             pool1_shape,
             filter2,
             kernel2_shape,
             pool2_shape):

"""Function to return unoptimized convnet model as described in paper

  Args:

  input_shape : tuple of size 3
                image dimension with channel at last

  filter1 : integer
            number of filter for Conv2D layer

  kernel1_shape : tuple of size 2

  pool1_shape : tuple of size 2

  similarly for others

  Returns:

  model object. Need to be compiled.
"""


  inputs = keras.Input(shape=input_shape,name='input')
  layer1 = layers.Conv2D(filter1,kernel1_shape,padding='valid',activation=
'sigmoid',name='layer1')(inputs)
  layer2 =layers.MaxPool2D(pool_size=pool1_shape,name='layer2')(layer1)
  layer3 = layers.Conv2D(filter2,kernel_size=kernel2_shape,padding='valid'
,activation='sigmoid',name='layer3')(layer2)
  layer4 = layers.MaxPool2D(pool_size=pool2_shape,name='layer4')(layer3)
  layer5_1 = layers.Flatten()(layer4)
  layer5_2 = layers.Flatten()(layer2)
  outputs = layers.Dense(5,activation='softmax',name='output')(layers.Conc
atenate()([layer5_1,layer5_2]))
  model = keras.Model(inputs=inputs,outputs=outputs)

  return model


class CustomCallBack1(keras.callbacks.Callback):
  def on_epoch_end(self,epochs,logs=None):
    if logs.get("accuracy")>0.88:
      self.model.stop_training = True

class CustomCallBack2(keras.callbacks.Callback):
  def on_epoch_end(self,epochs,logs=None):
    if logs.get("accuracy")>0.95:
      self.model.stop_training = True


def chart(model_history,title):
  plt.plot(model_history.history['accuracy'])
  plt.grid()
  plt.title(title)
  plt.ylabel('accuracy')
  plt.xlabel('epoch')
  plt.show()

model1 = my_model((42,40,3),
                  5,
                  (7,5),
                  (4,4),
                  30,
                  (3,3),
                  (2,2))
print(model1.summary())

model1.compile(
    loss = keras.losses.SparseCategoricalCrossentropy(),
    optimizer = keras.optimizers.Adam(lr=0.0008),
    metrics=['accuracy']
)
```

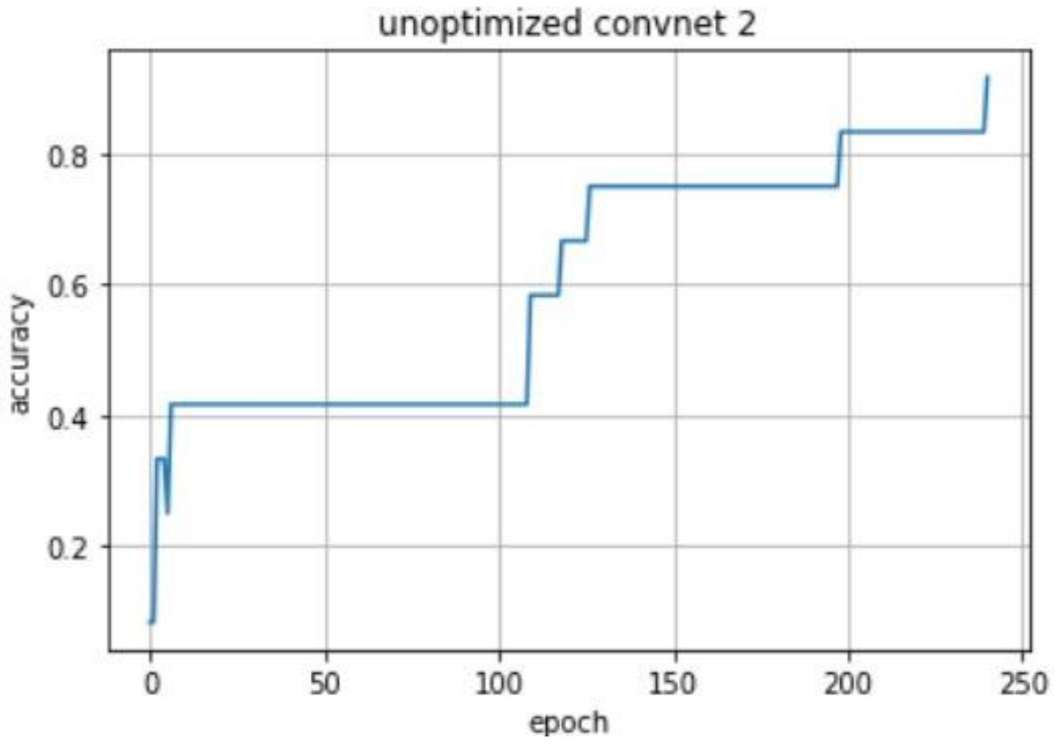
```
history = model1.fit(np.array(x_train),np.array(y_train),batch_size=64,epo
chs=300,verbose=1,callbacks=[CustomCallBack1()])

model1.evaluate(np.array(x_validate),np.array(y_validate),batch_size = 64,
verbose=1)
chart(history,'unoptimized_Convnet1')


def optimize(mod,filter1,filter2,kernel,sigma):
  #kernel shape of second conv2D layer

  intermediate_layer_model = keras.Model(inputs=mod.inputs,
                                    outputs=mod.get_layer('layer4').outp
ut)

  #Creation of similarity matrix
  M = np.zeros(shape=(filter2,filter2))
  for image in x_train:
    Mk = np.zeros(shape=(filter2,filter2))
    #Getting the feature maps after second sub sampling
    intermediate_output = intermediate_layer_model.predict(image[np.newaxi
s,:])
    for i in range(filter2-1):
      Ai = intermediate_output[0,:,:,i]
      normAi = np.linalg.norm(Ai)
      for j in range(i+1,filter2):
        Aj = intermediate_output[0,:,:,j]
        Mk[i,j] = np.sum(Ai*Aj)/(normAi * np.linalg.norm(Aj))
    M += Mk
  M = M/len(x_train)

  #getting weights and biases of layers that needs to be changed.
  #In this case, the second Conv2D layer and the last dense layer
  weights = mod.layers[8].get_weights()[0]
  bias = mod.layers[8].get_weights()[1]
  weights_conv = mod.layers[3].get_weights()[0]
  bias_conv = mod.layers[3].get_weights()[1]

  element = kernel[0]*kernel[1]


#Calculating new weights and biases
  removed_weight_index = []
  for i in range(filter2-1):
    for j in range(i+1,filter2):
      if(j not in removed_weight_index and M[i,j] >= sigma):
        weights[i*element:i*element + element,:] += weights[j*element:j*el
ement+element,:]
        weights[j*element:j*element+element,:] = 0
        M[j,:] = 0
        removed_weight_index.append(j)
  modified_filter_count = filter2 - len(removed_weight_index)


modified_weights_conv = np.zeros(shape=(kernel[0],kernel[1],filter1,modifi
ed_filter_count))
  modified_biases_conv = np.zeros(shape=(modified_filter_count,))
  modified_weights_dense = np.zeros(shape=(weights.shape[0] - element*len(
removed_weight_index),5))
  modified_bias_dense = np.zeros(shape=(5,))

  #Storing new weights and biases in proper arrays for further usage
  count = 0
  for i in range(filter2):
    if i not in removed_weight_index:
      modified_weights_conv[:,:,:,count] = weights_conv[:,:,:,i]
      modified_biases_conv[count] = bias_conv[i]
      modified_weights_dense[count*element:count*element+element,:] = weig
hts[i*element:i*element+element,:]
      last = count*element + element
      count += 1

  modified_weights_dense[last:,:] = weights[kernel[0]*kernel[1]*filter2:,:
]
  modified_bias_dense[:] = bias[:]

  return (modified_filter_count, [modified_weights_conv,modified_biases_co
nv], [modified_weights_dense,modified_bias_dense])


modified_filter_count, conv_weights, dense_weights = optimize(model1,5,30,
(3,3),0.999)
modified_model1 = my_model((42,40,3),
                  5,
                  (7,5),
                  (4,4),
                  modified_filter_count,
                  (3,3),
                  (2,2))
```

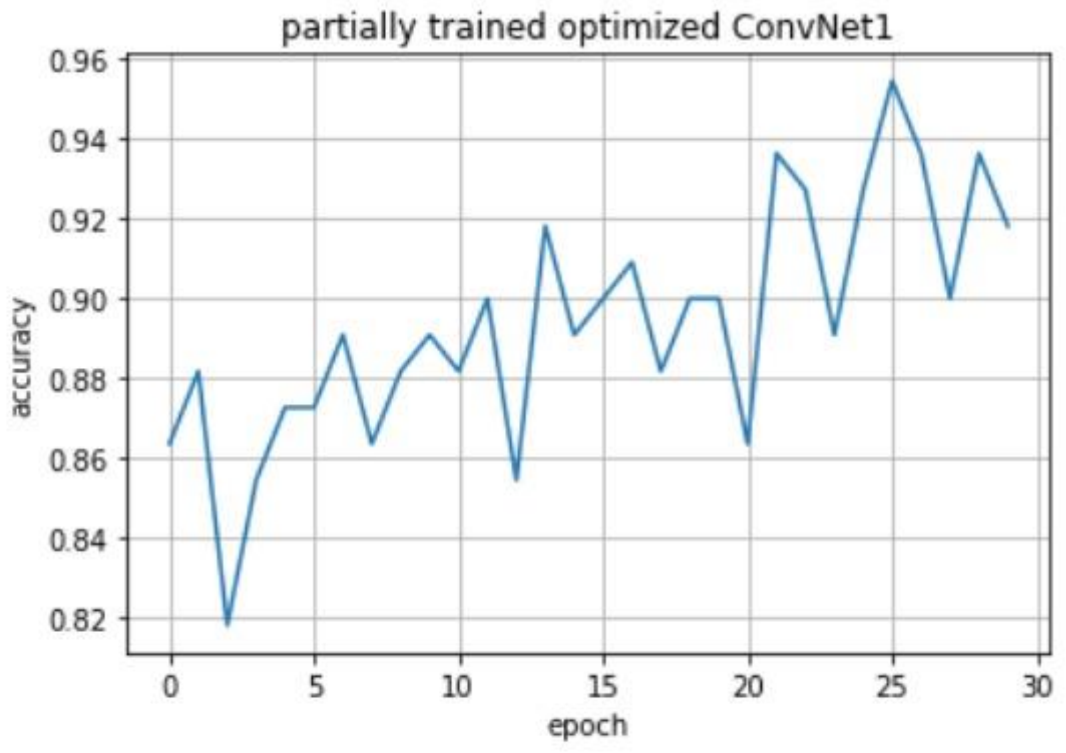
```
modified_model1.layers[1].set_weights([model1.layers[1].get_weights()[0],m
odel1.layers[1].get_weights()[1]])
modified_model1.layers[3].set_weights(conv_weights)
modified_model1.layers[8].set_weights(dense_weights)

print(modified_model1.summary())

modified_model1.compile(
    loss = keras.losses.SparseCategoricalCrossentropy(),
    optimizer = keras.optimizers.Adam(lr=0.0008),
    metrics=['accuracy'])
history = modified_model1.fit(np.array(x_train),np.array(y_train),batch_si
ze=10,epochs=30,verbose=1)

modified_model1.evaluate(np.array(x_validate),np.array(y_validate),batch_s
ize = 10,verbose=1)

chart(history,'partially trained optimized ConvNet1')

regular_set = []
regular_label = []
hard_set = []
hard_label = []
for img,label in zip(x_train,y_train):
  if np.max(modified_model1.predict(img[np.newaxis,:])) > 0.5 :
    regular_set.append(img)
    regular_label.append(label)
  else:
    hard_set.append(img)
    hard_label.append(label)

model2 = my_model((42,40,3),
                  6,
                  (7,5),
                  (4,4),
                  12,
                  (3,3),
                  (2,2))


print(model2.summary())


model2.compile(
    loss = keras.losses.SparseCategoricalCrossentropy(),
    optimizer = keras.optimizers.Adam(lr=0.0008),
    metrics=['accuracy']
)

history = model2.fit(np.array(hard_set),np.array(hard_label),batch_size=64
,epochs=300,verbose=1,callbacks=[CustomCallBack1()])


chart(history,'unoptimized convnet 2')

modified_filter_count, conv_weights, dense_weights = optimize(model2,6,12,
(3,3),0.999)

modified_model2 = my_model((42,40,3),
                  6,
                  (7,5),
                  (4,4),
                  modified_filter_count,
                  (3,3),
                  (2,2))

modified_model2.layers[1].set_weights([model2.layers[1].get_weights()[0],m
odel2.layers[1].get_weights()[1]])
modified_model2.layers[3].set_weights(conv_weights)
modified_model2.layers[8].set_weights(dense_weights)

modified_model2.compile(
    loss = keras.losses.SparseCategoricalCrossentropy(),
    optimizer = keras.optimizers.Adam(lr=0.0008),
    metrics=['accuracy']
)

history1 = modified_model2.fit(np.array(hard_set),np.array(hard_label),bat
ch_size=64,epochs=5,verbose=1)
history2 = modified_model1.fit(np.array(regular_set),np.array(regular_labe
l),batch_size=64,epochs=80,verbose=1)

chart(history1,'trained convnet 2')
chart(history2,'trained convnet 1')



def pred(img):
  if np.max(modified_model1.predict(img)) > 0.7:
    return np.argmax(modified_model1.predict(img))
  elif np.max(modified_model2.predict(img)) > 0.5:
    return np.argmax(modified_model2.predict(img))
  elif np.max(modified_model1.predict(img)) > np.max(modified_model2.predi
ct(img)) > 0.5:
    return np.argmax(modified_model1.predict(img))
  else:
    return np.argmax(modified_model2.predict(img))

correct = []
incorrect = []
for img,label in zip(x_test,y_test):
  output = pred(img[np.newaxis,:])
  if output == label:
    correct.append(1)
  else:
    incorrect.append(1)

a = len(correct)
b = len(incorrect)

#paper accuracy = 96.04%
print("accuracy = ",a/(a+b))
print('error rate = ',b/(b+a))

#########################################################################
```

4. Results

The graphs of the results are as follows:







The results of the implementation of the project are:
(The procedure used for classifying test images used is given below with $\lambda_1 = 0.7$ and $\lambda_2 = 0.5$.)

The model produced an accuracy of 94.60% as compared to 96.04% mentioned in the paper. With quite fast processing performance, thus putting very small load on the computational platform.

5. Conclusion

We were successfully able to implement the minimalistic crowd density recognition with 94.60% accuracy on the test data. As per the aim we were able to achieve these result with very light and fast 2 layer convolutional and single layer of artificial neural network, thus improving performance and speed of the crowd density recognition drastically. Thus we can conclude that we were successful at achieving the goal of building a crowd density recognition algorithm, which works with very good accuracy and has low complexity and fast performance.