DriverdistractionML

January 13, 2022

1 Detection of distracted driver using Convolutional Neural Networks

1.0.1 Introduction

In this project, the detection of distracted driver with Machine Learning using Convolutional Neural Networks was analysed and predicted.

1.0.2 Dataset description

The dataset had set of training and test images. The training data splitted into ten classes from c0 to c9. The 10 classes to predict are:

```
c0: normal driving
c1: texting - right
c2: talking on the phone - right
c3: texting - left
c4: talking on the phone - left
c5: operating the radio
c6: drinking
c7: reaching behind
c8: hair and makeup
c9: talking to passenger
```

Along with the set of images, two csv files were presented to assist our project. One with the details about the name of the images along with the class and another sample csv to show the submission format of the project.

1.0.3 Libraries

The libraries were imported to support our project. The assistance of tensorflow and keras is vital to proceed ahead. With matplot to plot charts and pandas to perform csv read and write operations.

[2]:
```
import os
from os.path import join
import tensorflow as tf
import keras_preprocessing
from keras_preprocessing import image
from keras_preprocessing.image import ImageDataGenerator
```

```python
import matplotlib.pyplot as plt
import pandas as pd
```

1.0.4 Model

The Convolutional Neural Network was constructed with input size of (100,100) with the '3' represents 'rgb' format of the image. With Batch normalization, we can standardize the data in between convolutional layers. Maxpooling is to find out the maximum value from the region covered by filter and the data will be converted to one dimensional array using flatten and dropout will help us to prevent overfitting. The hidden dense layers were added to improve efficiency and with the final dense layer represents output with 10 classes. The optimizer 'adam' was used to compile the model.

[3]:
```python
cnnmodel = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(100, 100,␣
↪3)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu', padding = 'same'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu', padding = 'same'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(128, (3,3), activation='relu', padding = 'same'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
    ])
cnnmodel.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics␣
↪= ['accuracy'])
cnnmodel.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 98, 98, 32)        896
_________________________________________________________________
batch_normalization (BatchNo (None, 98, 98, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 49, 49, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 49, 49, 64)        18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 49, 49, 64)        256
```

```
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 49, 49, 64)        36928
_________________________________________________________________
batch_normalization_2 (Batch (None, 49, 49, 64)        256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 24, 24, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 128)       73856
_________________________________________________________________
batch_normalization_3 (Batch (None, 24, 24, 128)       512
_________________________________________________________________
flatten (Flatten)            (None, 73728)             0
_________________________________________________________________
dropout (Dropout)            (None, 73728)             0
_________________________________________________________________
dense (Dense)                (None, 1024)              75498496
_________________________________________________________________
dense_1 (Dense)              (None, 512)               524800
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 76,159,754
Trainable params: 76,159,178
Non-trainable params: 576
_________________________________________________________________
```

1.0.5 Data preprocessing

The working directory was set to acccess the folder contains training images.

[4]:
```
workingdir = os.path.abspath('')
trainingdirectory = os.path.join(workingdir + '/
 ↪state-farm-distracted-driver-detection/imgs/train/')
```

1.0.6 Train and validation dataset split

Image generator was built to get access of images from the training folder. The data augumentation was used to generalize the model with horizontal flip, width and heigth shift range and rotation range. To improve the model, the training data were splitted into training and validation data in the ratio of 80:20. From the generator, the training and validation set can be accessed in the batch size of 64.

[5]:
```
trainingdataimage = ImageDataGenerator(rescale = 1./255, height_shift_range = 0.
 ↪2,
                                        width_shift_range = 0.2, shear_range =␣
 ↪0.2, rotation_range = 40, zoom_range = 0.2,
                                        fill_mode = 'nearest', horizontal_flip␣
 ↪= True, validation_split = 0.2)
```

```
trainingset = trainingdataimage.flow_from_directory(trainingdirectory,
                                                        target_size = (100,␣
 ↪100), batch_size = 64,
                                                        class_mode =␣
 ↪'categorical', subset = 'training', shuffle = True)
validationset = trainingdataimage.flow_from_directory(trainingdirectory,
                                                        target_size = (100,␣
 ↪100), batch_size = 64,
                                                        class_mode =␣
 ↪'categorical', subset = 'validation', shuffle = True)
```

```
Found 17943 images belonging to 10 classes.
Found 4481 images belonging to 10 classes.
```

1.0.7 Model fit

The training and validation images made to fit with the CNN model on 60 iterations with steps per iteration will be length of the image generator of training set and length of image generator of validation set was denoted as validation steps.

[6]:
```
history =  cnnmodel.fit(trainingset, epochs = 60, steps_per_epoch =␣
 ↪len(trainingset),
                             validation_data = validationset, verbose = 1,␣
 ↪validation_steps = len(validationset))
```

```
Epoch 1/60
281/281 [==============================] - 376s 1s/step - loss: 2.5116 -
accuracy: 0.1720 - val_loss: 3.2027 - val_accuracy: 0.1035
Epoch 2/60
281/281 [==============================] - 323s 1s/step - loss: 1.8083 -
accuracy: 0.3328 - val_loss: 2.3313 - val_accuracy: 0.2508
Epoch 3/60
281/281 [==============================] - 318s 1s/step - loss: 1.4136 -
accuracy: 0.4869 - val_loss: 2.3826 - val_accuracy: 0.3113
Epoch 4/60
281/281 [==============================] - 319s 1s/step - loss: 1.1155 -
accuracy: 0.6021 - val_loss: 1.1466 - val_accuracy: 0.6072
Epoch 5/60
281/281 [==============================] - 321s 1s/step - loss: 0.9086 -
accuracy: 0.6864 - val_loss: 1.3155 - val_accuracy: 0.5418
Epoch 6/60
281/281 [==============================] - 323s 1s/step - loss: 0.7383 -
accuracy: 0.7448 - val_loss: 1.2715 - val_accuracy: 0.5952
Epoch 7/60
281/281 [==============================] - 319s 1s/step - loss: 0.6287 -
accuracy: 0.7870 - val_loss: 0.8880 - val_accuracy: 0.7048
Epoch 8/60
281/281 [==============================] - 319s 1s/step - loss: 0.5582 -
```

```
accuracy: 0.8103 - val_loss: 1.0156 - val_accuracy: 0.6809
Epoch 9/60
281/281 [==============================] - 318s 1s/step - loss: 0.5107 -
accuracy: 0.8295 - val_loss: 1.1969 - val_accuracy: 0.6255
Epoch 10/60
281/281 [==============================] - 330s 1s/step - loss: 0.4548 -
accuracy: 0.8499 - val_loss: 0.6042 - val_accuracy: 0.8027
Epoch 11/60
281/281 [==============================] - 338s 1s/step - loss: 0.4264 -
accuracy: 0.8586 - val_loss: 0.5852 - val_accuracy: 0.8134
Epoch 12/60
281/281 [==============================] - 317s 1s/step - loss: 0.4007 -
accuracy: 0.8697 - val_loss: 0.9471 - val_accuracy: 0.7119
Epoch 13/60
281/281 [==============================] - 325s 1s/step - loss: 0.3753 -
accuracy: 0.8781 - val_loss: 0.3377 - val_accuracy: 0.8884
Epoch 14/60
281/281 [==============================] - 348s 1s/step - loss: 0.3502 -
accuracy: 0.8889 - val_loss: 0.4997 - val_accuracy: 0.8424
Epoch 15/60
281/281 [==============================] - 349s 1s/step - loss: 0.3306 -
accuracy: 0.8908 - val_loss: 0.6327 - val_accuracy: 0.8087
Epoch 16/60
281/281 [==============================] - 353s 1s/step - loss: 0.3241 -
accuracy: 0.8922 - val_loss: 0.4044 - val_accuracy: 0.8746
Epoch 17/60
281/281 [==============================] - 350s 1s/step - loss: 0.3174 -
accuracy: 0.8966 - val_loss: 0.4143 - val_accuracy: 0.8639
Epoch 18/60
281/281 [==============================] - 347s 1s/step - loss: 0.3037 -
accuracy: 0.8990 - val_loss: 0.3509 - val_accuracy: 0.8848
Epoch 19/60
281/281 [==============================] - 351s 1s/step - loss: 0.2923 -
accuracy: 0.9056 - val_loss: 0.6673 - val_accuracy: 0.7891
Epoch 20/60
281/281 [==============================] - 342s 1s/step - loss: 0.2779 -
accuracy: 0.9098 - val_loss: 0.4353 - val_accuracy: 0.8699
Epoch 21/60
281/281 [==============================] - 344s 1s/step - loss: 0.2670 -
accuracy: 0.9128 - val_loss: 0.4784 - val_accuracy: 0.8465
Epoch 22/60
281/281 [==============================] - 341s 1s/step - loss: 0.2477 -
accuracy: 0.9206 - val_loss: 0.3648 - val_accuracy: 0.8846
Epoch 23/60
281/281 [==============================] - 347s 1s/step - loss: 0.2475 -
accuracy: 0.9217 - val_loss: 0.4804 - val_accuracy: 0.8552
Epoch 24/60
281/281 [==============================] - 340s 1s/step - loss: 0.2576 -
```

```
accuracy: 0.9172 - val_loss: 0.4012 - val_accuracy: 0.8819
Epoch 25/60
281/281 [==============================] - 343s 1s/step - loss: 0.2390 -
accuracy: 0.9268 - val_loss: 0.6704 - val_accuracy: 0.7900
Epoch 26/60
281/281 [==============================] - 333s 1s/step - loss: 0.2378 -
accuracy: 0.9234 - val_loss: 0.4291 - val_accuracy: 0.8777
Epoch 27/60
281/281 [==============================] - 333s 1s/step - loss: 0.2209 -
accuracy: 0.9307 - val_loss: 0.3868 - val_accuracy: 0.8748
Epoch 28/60
281/281 [==============================] - 321s 1s/step - loss: 0.2311 -
accuracy: 0.9289 - val_loss: 0.4000 - val_accuracy: 0.8790
Epoch 29/60
281/281 [==============================] - 334s 1s/step - loss: 0.2034 -
accuracy: 0.9336 - val_loss: 0.2391 - val_accuracy: 0.9277
Epoch 30/60
281/281 [==============================] - 322s 1s/step - loss: 0.2077 -
accuracy: 0.9350 - val_loss: 0.2802 - val_accuracy: 0.9197
Epoch 31/60
281/281 [==============================] - 316s 1s/step - loss: 0.2071 -
accuracy: 0.9374 - val_loss: 0.3917 - val_accuracy: 0.8777
Epoch 32/60
281/281 [==============================] - 317s 1s/step - loss: 0.2057 -
accuracy: 0.9360 - val_loss: 0.3143 - val_accuracy: 0.9092
Epoch 33/60
281/281 [==============================] - 315s 1s/step - loss: 0.1995 -
accuracy: 0.9377 - val_loss: 0.2639 - val_accuracy: 0.9170
Epoch 34/60
281/281 [==============================] - 316s 1s/step - loss: 0.1884 -
accuracy: 0.9391 - val_loss: 0.3887 - val_accuracy: 0.8931
Epoch 35/60
281/281 [==============================] - 315s 1s/step - loss: 0.1911 -
accuracy: 0.9409 - val_loss: 0.2730 - val_accuracy: 0.9159
Epoch 36/60
281/281 [==============================] - 313s 1s/step - loss: 0.1929 -
accuracy: 0.9392 - val_loss: 0.9694 - val_accuracy: 0.7710
Epoch 37/60
281/281 [==============================] - 312s 1s/step - loss: 0.1743 -
accuracy: 0.9453 - val_loss: 0.2871 - val_accuracy: 0.9110
Epoch 38/60
281/281 [==============================] - 313s 1s/step - loss: 0.1715 -
accuracy: 0.9458 - val_loss: 0.5447 - val_accuracy: 0.8757
Epoch 39/60
281/281 [==============================] - 311s 1s/step - loss: 0.1768 -
accuracy: 0.9468 - val_loss: 0.3030 - val_accuracy: 0.9134
Epoch 40/60
281/281 [==============================] - 313s 1s/step - loss: 0.1727 -
```

```
accuracy: 0.9469 - val_loss: 0.2971 - val_accuracy: 0.9096
Epoch 41/60
281/281 [==============================] - 313s 1s/step - loss: 0.1711 -
accuracy: 0.9468 - val_loss: 0.2146 - val_accuracy: 0.9317
Epoch 42/60
281/281 [==============================] - 313s 1s/step - loss: 0.1784 -
accuracy: 0.9454 - val_loss: 0.2956 - val_accuracy: 0.9177
Epoch 43/60
281/281 [==============================] - 312s 1s/step - loss: 0.1580 -
accuracy: 0.9510 - val_loss: 0.1983 - val_accuracy: 0.9400
Epoch 44/60
281/281 [==============================] - 312s 1s/step - loss: 0.1649 -
accuracy: 0.9489 - val_loss: 0.5169 - val_accuracy: 0.8505
Epoch 45/60
281/281 [==============================] - 312s 1s/step - loss: 0.1633 -
accuracy: 0.9498 - val_loss: 0.2482 - val_accuracy: 0.9250
Epoch 46/60
281/281 [==============================] - 312s 1s/step - loss: 0.1590 -
accuracy: 0.9518 - val_loss: 0.1737 - val_accuracy: 0.9476
Epoch 47/60
281/281 [==============================] - 312s 1s/step - loss: 0.1558 -
accuracy: 0.9526 - val_loss: 0.2266 - val_accuracy: 0.9362
Epoch 48/60
281/281 [==============================] - 312s 1s/step - loss: 0.1531 -
accuracy: 0.9549 - val_loss: 1.1021 - val_accuracy: 0.7414
Epoch 49/60
281/281 [==============================] - 312s 1s/step - loss: 0.1555 -
accuracy: 0.9535 - val_loss: 0.2648 - val_accuracy: 0.9214
Epoch 50/60
281/281 [==============================] - 312s 1s/step - loss: 0.1518 -
accuracy: 0.9523 - val_loss: 0.2495 - val_accuracy: 0.9297
Epoch 51/60
281/281 [==============================] - 314s 1s/step - loss: 0.1526 -
accuracy: 0.9544 - val_loss: 0.4012 - val_accuracy: 0.8947
Epoch 52/60
281/281 [==============================] - 312s 1s/step - loss: 0.1416 -
accuracy: 0.9567 - val_loss: 0.3095 - val_accuracy: 0.9112
Epoch 53/60
281/281 [==============================] - 312s 1s/step - loss: 0.1387 -
accuracy: 0.9559 - val_loss: 0.1920 - val_accuracy: 0.9389
Epoch 54/60
281/281 [==============================] - 312s 1s/step - loss: 0.1422 -
accuracy: 0.9569 - val_loss: 0.3476 - val_accuracy: 0.9078
Epoch 55/60
281/281 [==============================] - 314s 1s/step - loss: 0.1490 -
accuracy: 0.9567 - val_loss: 0.2508 - val_accuracy: 0.9333
Epoch 56/60
281/281 [==============================] - 315s 1s/step - loss: 0.1364 -
```

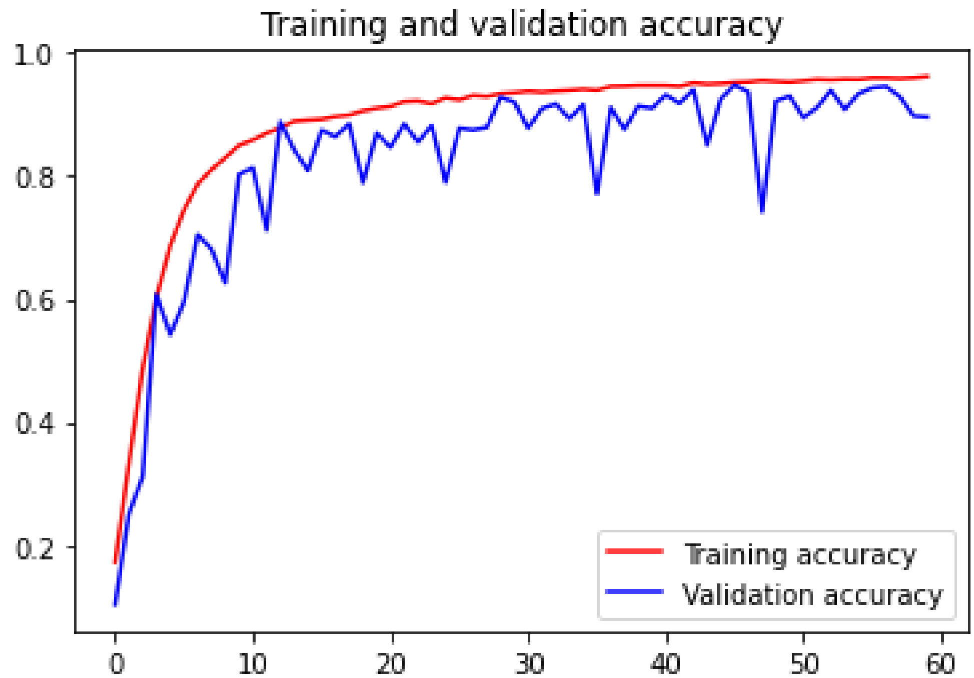
```
accuracy: 0.9589 - val_loss: 0.1951 - val_accuracy: 0.9431
Epoch 57/60
281/281 [==============================] - 313s 1s/step - loss: 0.1364 -
accuracy: 0.9587 - val_loss: 0.1701 - val_accuracy: 0.9453
Epoch 58/60
281/281 [==============================] - 312s 1s/step - loss: 0.1372 -
accuracy: 0.9579 - val_loss: 0.2215 - val_accuracy: 0.9288
Epoch 59/60
281/281 [==============================] - 313s 1s/step - loss: 0.1409 -
accuracy: 0.9591 - val_loss: 0.4382 - val_accuracy: 0.8976
Epoch 60/60
281/281 [==============================] - 313s 1s/step - loss: 0.1296 -
accuracy: 0.9613 - val_loss: 0.3573 - val_accuracy: 0.8958
```

1.0.8 Plot to show training accuracy vs validation accuracy

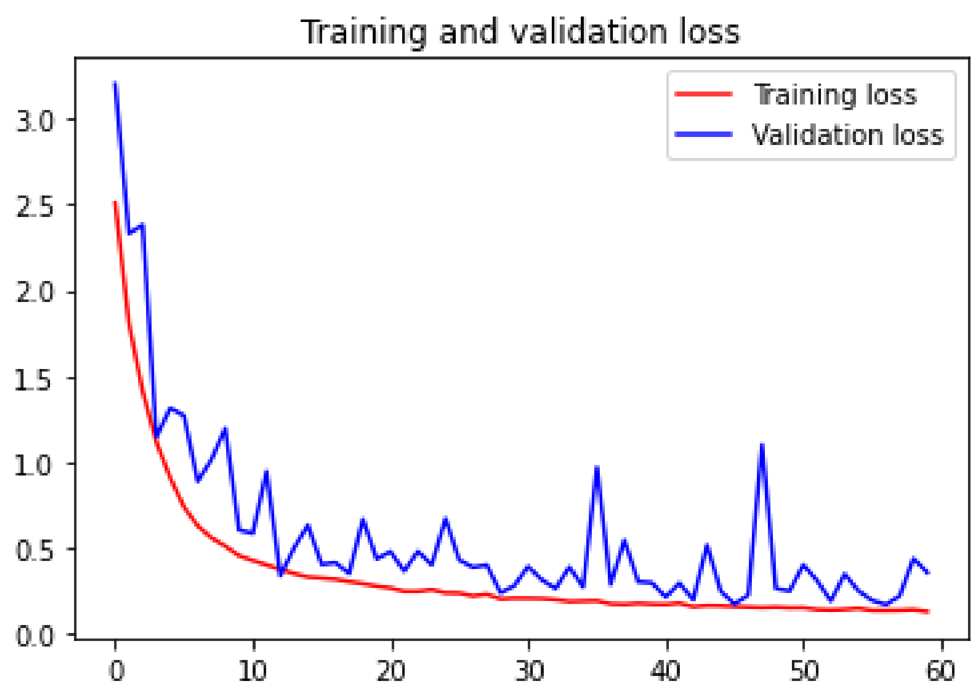
The two plots were designed to visualize the learning curve of the model. One plot concentrated on Training and validation accuracy over 60 iterations and next one concentrated on training and validation loss.

[8]:

```
accuracy = history.history['accuracy']
validation_accuracy = history.history['val_accuracy']
loss = history.history['loss']
validation_loss = history.history['val_loss']

epochs = range(len(accuracy))

plt.plot(epochs, accuracy, 'red', label='Training accuracy')
plt.plot(epochs, validation_accuracy, 'blue', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend(loc=0)
plt.figure()
plt.show()

plt.plot(epochs, loss, 'red', label='Training loss')
plt.plot(epochs, validation_loss, 'blue', label='Validation loss')
plt.title('Training and validation loss')
plt.legend(loc=0)
plt.figure()
plt.show()
```




```
<Figure size 432x288 with 0 Axes>
```



```
<Figure size 432x288 with 0 Axes>
```

1.0.9 Test data prediction

Like training data, image generator were built for test data along with its directory. The important step of the project is the prediction of test images with the learning the CNN model has undergone with training and validation images.

```
[9]: testparentdirectory = os.path.join(workingdir + '/
      ↪state-farm-distracted-driver-detection/imgs/')
     testdataimage = ImageDataGenerator(rescale = 1./255)
     testdata = testdataimage.flow_from_directory(testparentdirectory,␣
      ↪classes=['test'], target_size = (100,100))
     testoutput = cnnmodel.predict(testdata, verbose = 1)
```

```
Found 79726 images belonging to 1 classes.
2492/2492 [==============================] - 495s 199ms/step
```

1.0.10 Preparing output dataframe

The sample submission csv was read by pandas to prepare the format of output. With image and images name taken from the csv, the prediction values was replaced with the original value of csv in the same format and convert it to a dataframe to export it easily into a csv file.

```
[10]: specimencsv = pd.read_csv(os.path.join(workingdir + '/
       ↪state-farm-distracted-driver-detection/sample_submission.csv'))
      result = {'img':list(specimencsv.values[:,0]),}
      for value in range(0,10):
          result['c' + str(value)] = list(testoutput[:,value])
```

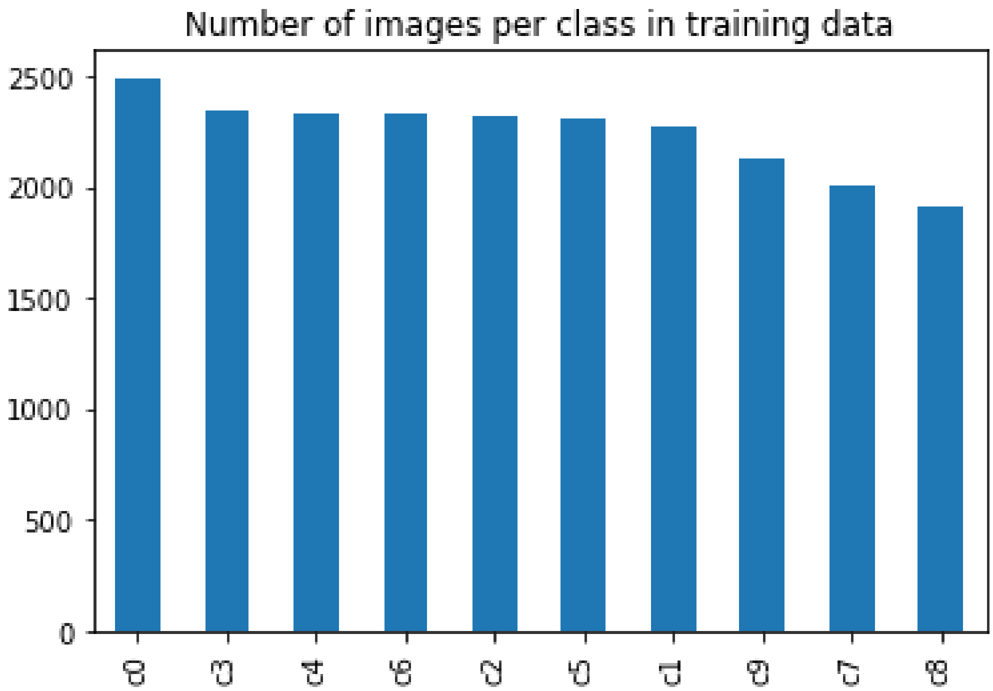
```
[11]: testoutput = pd.DataFrame(result)
```

1.0.11 Exploratory data analysis

The number of images in each class were depicted with a bar plot.

```
[12]: imagescsv = pd.read_csv(os.path.join(workingdir + '/
       ↪state-farm-distracted-driver-detection/driver_imgs_list.csv'))
      imagescsv.classname.value_counts().plot(kind = 'bar', label = 'index')
      plt.title('Number of images per class in training data')
```

```
[12]: Text(0.5, 1.0, 'Number of images per class in training data')
```



1.0.12 Preparing output file

Then, the csv had been written from the 'testoutput' dataframe.

[13]:
```
testoutput.to_csv('Testoutput.csv', index = False, encoding='utf-8')
```

1.0.13 Conclusion

Thus, the prediction of test images from the model with the learning of training and validation images was successfully exported as a csv file with over 96% accuracy.

1.0.14 References

1. https://www.tensorflow.org/tutorials/keras/classification
2. https://www.tensorflow.org/tutorials/images/cnn
3. https://charon.me/posts/keras/keras2/
4. https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator
5. https://www.tensorflow.org/guide/keras/train_and_evaluate