PARCIAL III DE OPTATIVA 3

CRISTIAN DANIEL GARCIA BARRERA, RUBEN DARIO JORDAN SAENZ

I. INTRODUCTION

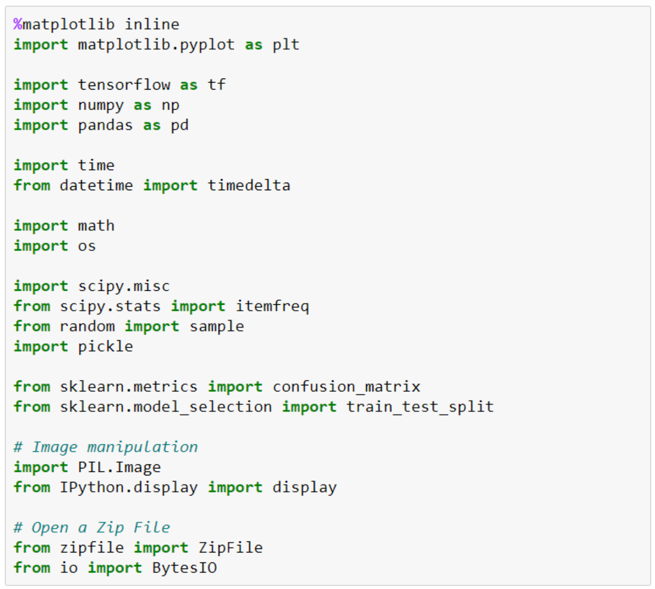
Through the application of deep neural networks where it is intended to classify images of different breeds, which have access to the neural network which will have the ability to determine the breed of a dog from a photo using the set of breed data of dog, receiving the input signal and giving an output signal for the classifications of the category showing the results for each dog breed.

II. METHODS

We proceed to provide the code with many examples of each breed class, then proceed to develop learning algorithms that look at these examples and learn about the appearance of each image of each dog breed that is being put in the learning code .
That is, first a set of training data of tagged images of different dog breeds is accumulated and then sent so that the neural network becomes familiar with the data obtained.
Given this fact, the image classification data line is formalized as follows:
• The entry is a set of training data consisting of N images, each with a K tag of different classes.
• The training set is used to train a coach to learn how are the characteristics of each of the classes
• The quality of the coach is evaluated, asking him to predict labels for a new group of images that he has not had access to and has not been able to see. Then the true labels of the images are compared with those predicted by the selector
In order to develop the code with good performance, two methods are used, which are the use of image classification with machine learning and the use of convolutional neural networks.

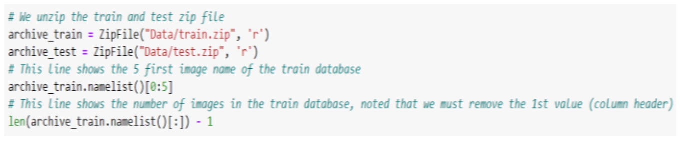
III. DATA PROCESSING PROCEDURE

1. The importation of all the packages or libraries necessary to carry out the program is carried out



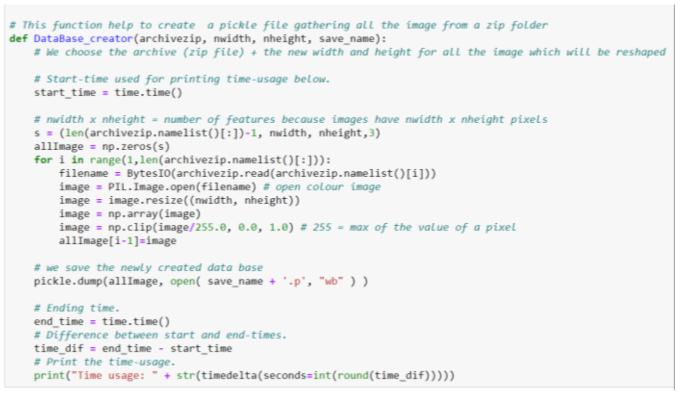
```python
%matplotlib inline
import matplotlib.pyplot as plt

import tensorflow as tf
import numpy as np
import pandas as pd

import time
from datetime import timedelta

import math
import os

import scipy.misc
from scipy.stats import itemfreq
from random import sample
import pickle

from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

# Image manipulation
import PIL.Image
from IPython.display import display

# Open a Zip File
from zipfile import ZipFile
from io import BytesIO
```

2. After importing the libraries, the files are decompressed so that the program has access to them



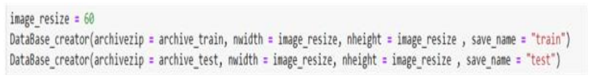
```python
# We unzip the train and test zip file
archive_train = ZipFile("Data/train.zip", 'r')
archive_test = ZipFile("Data/test.zip", 'r')
# This line shows the 5 first image name of the train database
archive_train.namelist()[0:5]
# This line shows the number of images in the train database, noted that we must remove the 1st value (column header)
len(archive_train.namelist()[:]) - 1
```

3. The data is resized and normalized, a function is used which generates a pickle file where it saves all the decompressed images



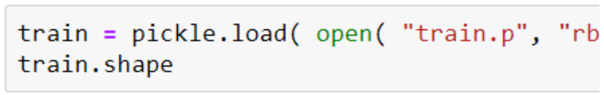
```python
# This function help to create  a pickle file gathering all the image from a zip folder
def DataBase_creator(archivezip, nwidth, nheight, save_name):
    # We choose the archive (zip file) + the new width and height for all the image which will be reshaped

    # Start-time used for printing time-usage below.
    start_time = time.time()

    # nwidth x nheight = number of features because images have nwidth x nheight pixels
    s = (len(archivezip.namelist()[:])-1, nwidth, nheight,3)
    allImage = np.zeros(s)
    for i in range(1,len(archivezip.namelist()[:])):
        filename = BytesIO(archivezip.read(archivezip.namelist()[i]))
        image = PIL.Image.open(filename) # open colour image
        image = image.resize((nwidth, nheight))
        image = np.array(image)
        image = np.clip(image/255.0, 0.0, 1.0) # 255 = max of the value of a pixel
        allImage[i-1]=image

    # we save the newly created data base
    pickle.dump(allImage, open( save_name + '.p', "wb" ) )

    # Ending time.
    end_time = time.time()
    # Difference between start and end-times.
    time_dif = end_time - start_time
    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_dif)))))
```

Then define the new image size applied to all images



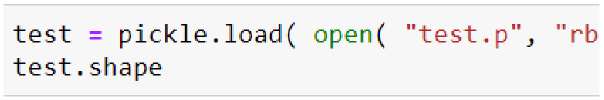
```
image_resize = 60
DataBase_creator(archivezip = archive_train, nwidth = image_resize, nheight = image_resize , save_name = "train")
DataBase_creator(archivezip = archive_test, nwidth = image_resize, nheight = image_resize , save_name = "test")
```

Training data is loaded



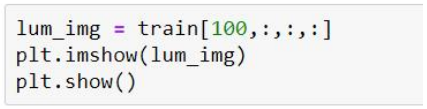
```
train = pickle.load( open( "train.p", "rb
train.shape
```

The test data is loaded

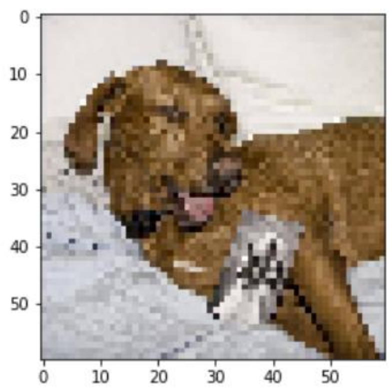


```
test = pickle.load( open( "test.p", "rb
test.shape
```

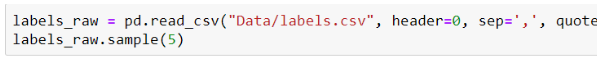
We proceed to show an image as a reference with which the program will be tested



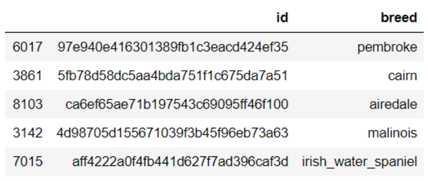
```
lum_img = train[100,:,:,:]
plt.imshow(lum_img)
plt.show()
```



4. The tag file is reviewed, the CSV file of the data tag is approached

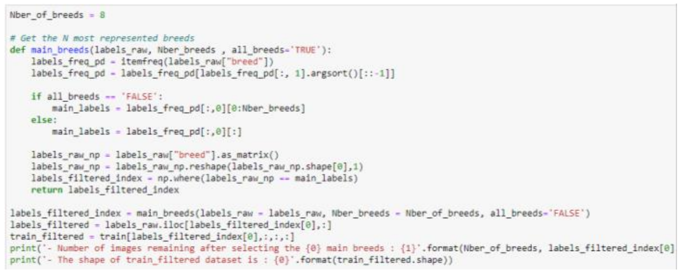


```
labels_raw = pd.read_csv("Data/labels.csv", header=0, sep=',', quote
labels_raw.sample(5)
```



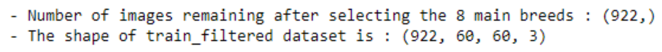
	id	breed
6017	97e940e416301389fb1c3eacd424ef35	pembroke
3861	5fb78d58dc5aa4bda751f1c675da7a51	cairn
8103	ca6ef65ae71b197543c69095ff46f100	airedale
3142	4d98705d155671039f3b45f96eb73a63	malinois
7015	aff4222a0f4fb441d627f7ad396caf3d	irish_water_spaniel

5. Extraction of the best known breeds, a reduction of data is made so that the difficulty of the program decreases in the data analytics apart from this will allow to be able to perform the calculations in less time.



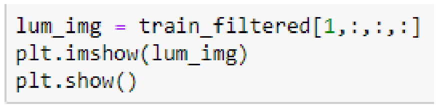
```
Nber_of_breeds = 8

# Get the N most represented breeds
def main_breeds(labels_raw, Nber_breeds , all_breeds='TRUE'):
    labels_freq_pd = itemfreq(labels_raw["breed"])
    labels_freq_pd = labels_freq_pd[labels_freq_pd[:, 1].argsort()[::-1]]

    if all_breeds == 'FALSE':
        main_labels = labels_freq_pd[:,0][0:Nber_breeds]
    else:
        main_labels = labels_freq_pd[:,0][:]

    labels_raw_np = labels_raw["breed"].as_matrix()
    labels_raw_np = labels_raw_np.reshape(labels_raw_np.shape[0],1)
    labels_filtered_index = np.where(labels_raw_np == main_labels)
    return labels_filtered_index

labels_filtered_index = main_breeds(labels_raw = labels_raw, Nber_breeds = Nber_of_breeds, all_breeds='FALSE')
labels_filtered = labels_raw.iloc[labels_filtered_index[0],:]
train_filtered = train[labels_filtered_index[0],:,:,:]
print('- Number of images remaining after selecting the {0} main breeds : {1}'.format(Nber_of_breeds, labels_filtered_index[0]
print('- The shape of train_filtered dataset is : {0}'.format(train_filtered.shape))
```

The following output is obtained

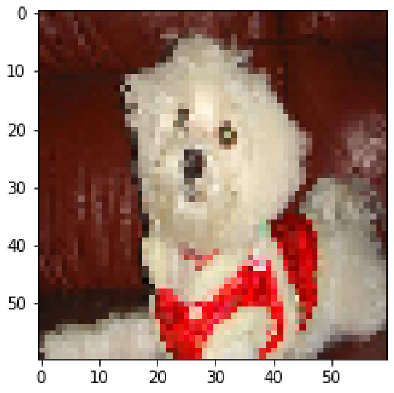


```
- Number of images remaining after selecting the 8 main breeds : (922,)
- The shape of train_filtered dataset is : (922, 60, 60, 3)
```

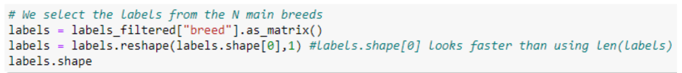
An image is displayed



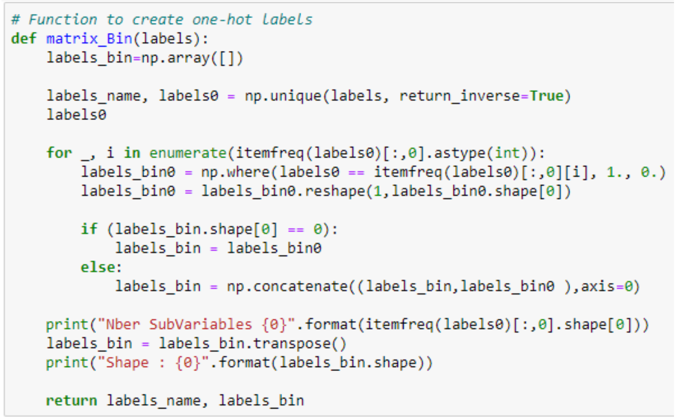
```
lum_img = train_filtered[1,:,:,:]
plt.imshow(lum_img)
plt.show()
```



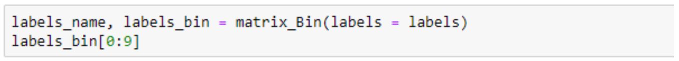
6. One-Hot tags are made, unique decoding is done for tag data



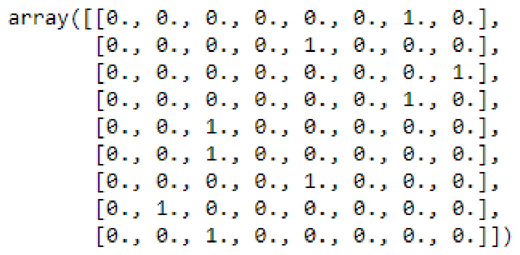
```
# We select the labels from the N main breeds
labels = labels_filtered["breed"].as_matrix()
labels = labels.reshape(labels.shape[0],1) #labels.shape[0] looks faster than using len(labels)
labels.shape
```



```
# Function to create one-hot labels
def matrix_Bin(labels):
    labels_bin=np.array([])

    labels_name, labels0 = np.unique(labels, return_inverse=True)
    labels0

    for _, i in enumerate(itemfreq(labels0)[:,0].astype(int)):
        labels_bin0 = np.where(labels0 == itemfreq(labels0)[:,0][i], 1., 0.)
        labels_bin0 = labels_bin0.reshape(1,labels_bin0.shape[0])

        if (labels_bin.shape[0] == 0):
            labels_bin = labels_bin0
        else:
            labels_bin = np.concatenate((labels_bin,labels_bin0 ),axis=0)

    print("Nber SubVariables {0}".format(itemfreq(labels0)[:,0].shape[0]))
    labels_bin = labels_bin.transpose()
    print("Shape : {0}".format(labels_bin.shape))

    return labels_name, labels_bin
```

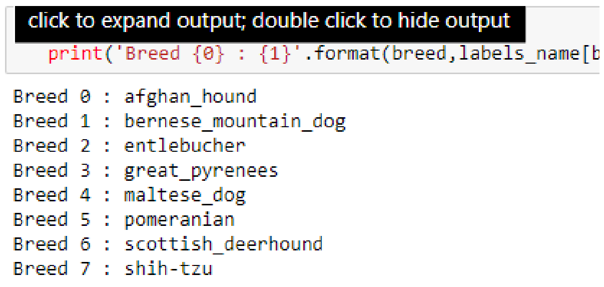


```
labels_name, labels_bin = matrix_Bin(labels = labels)
labels_bin[0:9]
```



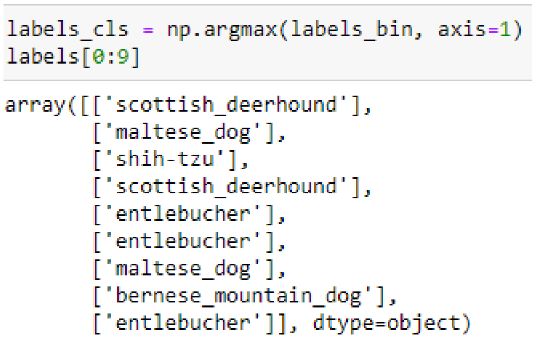
```
array([[0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0.]])
```

7. Quick check on tags
You look at the N tags that were saved and with the One-Hot tags you can find which breed corresponds to



```
print('Breed {0} : {1}'.format(breed,labels_name[b
```

```
Breed 0 : afghan_hound
Breed 1 : bernese_mountain_dog
Breed 2 : entlebucher
Breed 3 : great_pyrenees
Breed 4 : maltese_dog
Breed 5 : pomeranian
Breed 6 : scottish_deerhound
Breed 7 : shih-tzu
```

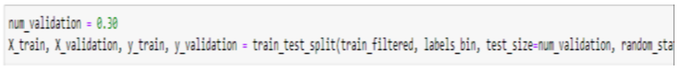


```
labels_cls = np.argmax(labels_bin, axis=1)
labels[0:9]
```

```
array([['scottish_deerhound'],
       ['maltese_dog'],
       ['shih-tzu'],
       ['scottish_deerhound'],
       ['entlebucher'],
       ['entlebucher'],
       ['maltese_dog'],
       ['bernese_mountain_dog'],
       ['entlebucher']], dtype=object)
```

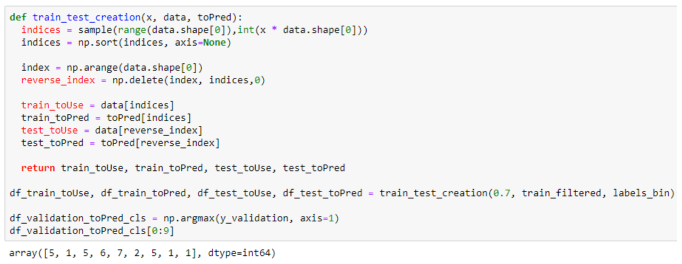
IV. Convolutional Neural Networks

1. Data creation and validation
   The data obtained from the data stream is divided into two of which one is a training set and the other is a validation set due to this the accuracy of the program in the train can be verified from the set of validation set training



```
num_validation = 0.30
X_train, X_validation, y_train, y_validation = train_test_split(train_filtered, labels_bin, test_size=num_validation, random_sta
```

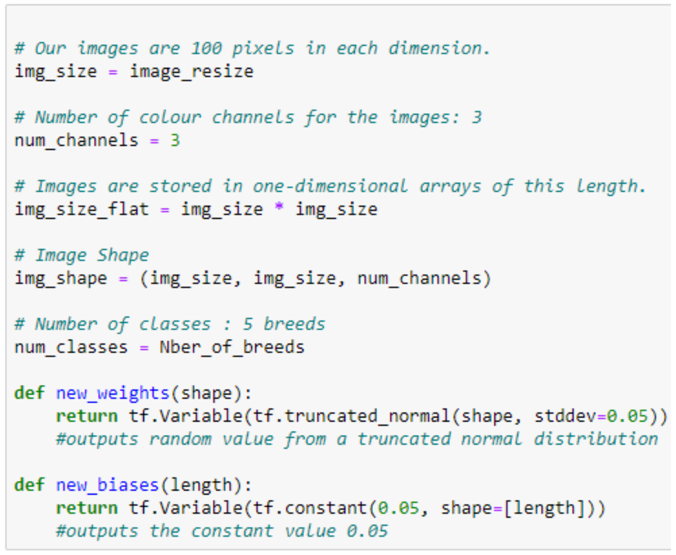
2. The creation of a test data stream is performed where the original data is divided to train and test sets



```
def train_test_creation(x, data, toPred):
  indices = sample(range(data.shape[0]),int(x * data.shape[0]))
  indices = np.sort(indices, axis=None)

  index = np.arange(data.shape[0])
  reverse_index = np.delete(index, indices,0)

  train_toUse = data[indices]
  train_toPred = toPred[indices]
  test_toUse = data[reverse_index]
  test_toPred = toPred[reverse_index]

  return train_toUse, train_toPred, test_toUse, test_toPred

df_train_toUse, df_train_toPred, df_test_toUse, df_test_toPred = train_test_creation(0.7, train_filtered, labels_bin)

df_validation_toPred_cls = np.argmax(y_validation, axis=1)
df_validation_toPred_cls[0:9]
```

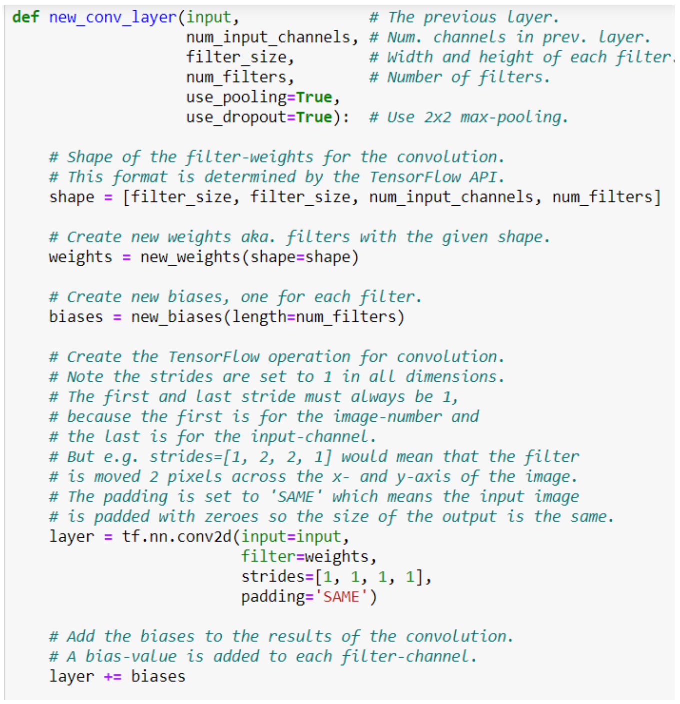
```
array([5, 1, 5, 6, 7, 2, 5, 1, 1], dtype=int64)
```

3. The definition of layers is carried out
Where we first define our weights and other constants

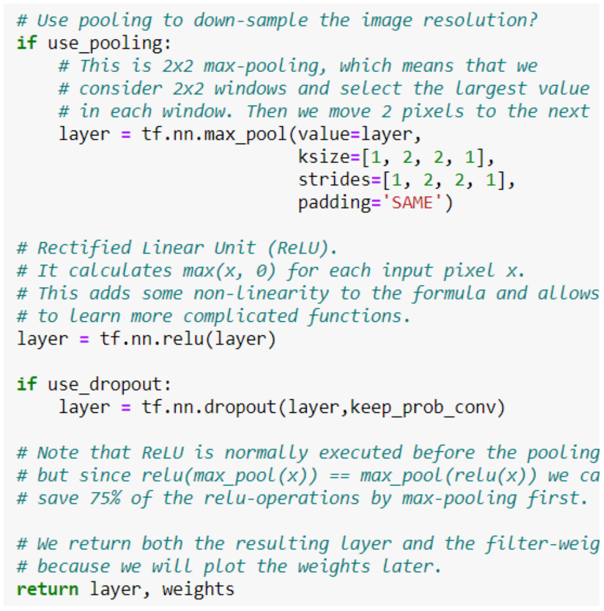


```
# Our images are 100 pixels in each dimension.
img_size = image_resize

# Number of colour channels for the images: 3
num_channels = 3

# Images are stored in one-dimensional arrays of this length.
img_size_flat = img_size * img_size

# Image Shape
img_shape = (img_size, img_size, num_channels)

# Number of classes : 5 breeds
num_classes = Nber_of_breeds

def new_weights(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.05))
    #outputs random value from a truncated normal distribution

def new_biases(length):
    return tf.Variable(tf.constant(0.05, shape=[length]))
    #outputs the constant value 0.05
```

Then the convolution layer is defined

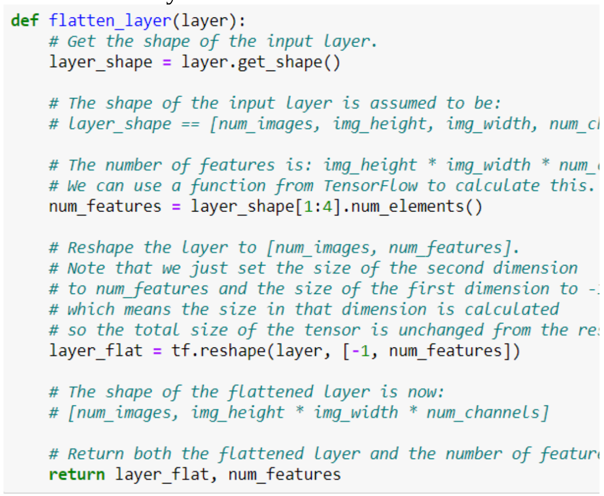


```
def new_conv_layer(input,              # The previous layer.
                   num_input_channels, # Num. channels in prev. layer.
                   filter_size,        # Width and height of each filter.
                   num_filters,        # Number of filters.
                   use_pooling=True,
                   use_dropout=True):  # Use 2x2 max-pooling.

    # Shape of the filter-weights for the convolution.
    # This format is determined by the TensorFlow API.
    shape = [filter_size, filter_size, num_input_channels, num_filters]

    # Create new weights aka. filters with the given shape.
    weights = new_weights(shape=shape)

    # Create new biases, one for each filter.
    biases = new_biases(length=num_filters)

    # Create the TensorFlow operation for convolution.
    # Note the strides are set to 1 in all dimensions.
    # The first and last stride must always be 1,
    # because the first is for the image-number and
    # the last is for the input-channel.
    # But e.g. strides=[1, 2, 2, 1] would mean that the filter
    # is moved 2 pixels across the x- and y-axis of the image.
    # The padding is set to 'SAME' which means the input image
    # is padded with zeroes so the size of the output is the same.
    layer = tf.nn.conv2d(input=input,
                         filter=weights,
                         strides=[1, 1, 1, 1],
                         padding='SAME')

    # Add the biases to the results of the convolution.
    # A bias-value is added to each filter-channel.
    layer += biases
```



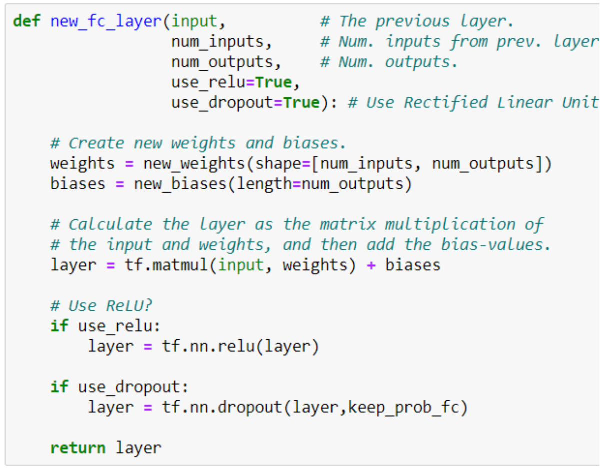
```
# Use pooling to down-sample the image resolution?
if use_pooling:
    # This is 2x2 max-pooling, which means that we
    # consider 2x2 windows and select the largest value
    # in each window. Then we move 2 pixels to the next
    layer = tf.nn.max_pool(value=layer,
                           ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1],
                           padding='SAME')

# Rectified Linear Unit (ReLU).
# It calculates max(x, 0) for each input pixel x.
# This adds some non-linearity to the formula and allows
# to learn more complicated functions.
layer = tf.nn.relu(layer)

if use_dropout:
    layer = tf.nn.dropout(layer,keep_prob_conv)

# Note that ReLU is normally executed before the pooling
# but since relu(max_pool(x)) == max_pool(relu(x)) we ca
# save 75% of the relu-operations by max-pooling first.

# We return both the resulting layer and the filter-weig
# because we will plot the weights later.
return layer, weights
```

Then the flat layer is defined



```
def flatten_layer(layer):
    # Get the shape of the input layer.
    layer_shape = layer.get_shape()

    # The shape of the input layer is assumed to be:
    # layer_shape == [num_images, img_height, img_width, num_ch

    # The number of features is: img_height * img_width * num_
    # We can use a function from TensorFlow to calculate this.
    num_features = layer_shape[1:4].num_elements()

    # Reshape the layer to [num_images, num_features].
    # Note that we just set the size of the second dimension
    # to num_features and the size of the first dimension to -
    # which means the size in that dimension is calculated
    # so the total size of the tensor is unchanged from the re
    layer_flat = tf.reshape(layer, [-1, num_features])

    # The shape of the flattened layer is now:
    # [num_images, img_height * img_width * num_channels]

    # Return both the flattened layer and the number of featur
    return layer_flat, num_features
```

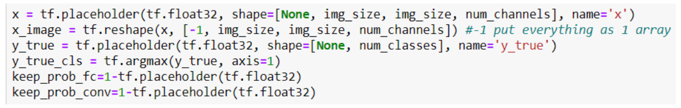
And finally you define the layer already completely finished and connected



```
def new_fc_layer(input,          # The previous layer.
                 num_inputs,     # Num. inputs from prev. layer
                 num_outputs,    # Num. outputs.
                 use_relu=True,
                 use_dropout=True): # Use Rectified Linear Unit

    # Create new weights and biases.
    weights = new_weights(shape=[num_inputs, num_outputs])
    biases = new_biases(length=num_outputs)

    # Calculate the layer as the matrix multiplication of
    # the input and weights, and then add the bias-values.
    layer = tf.matmul(input, weights) + biases

    # Use ReLU?
    if use_relu:
        layer = tf.nn.relu(layer)

    if use_dropout:
        layer = tf.nn.dropout(layer,keep_prob_fc)

    return layer
```

4. CNN with TensorFlow

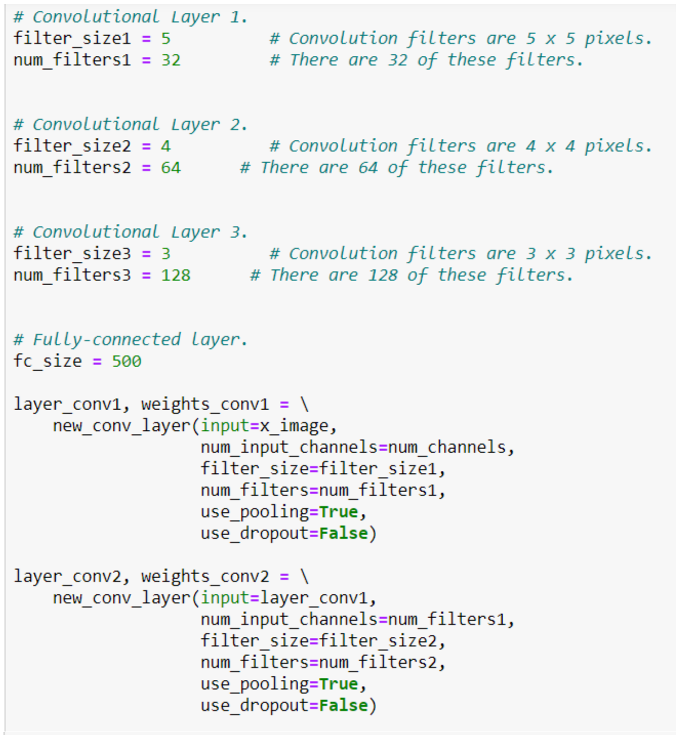
At this point a position marker will be set for the tensioner



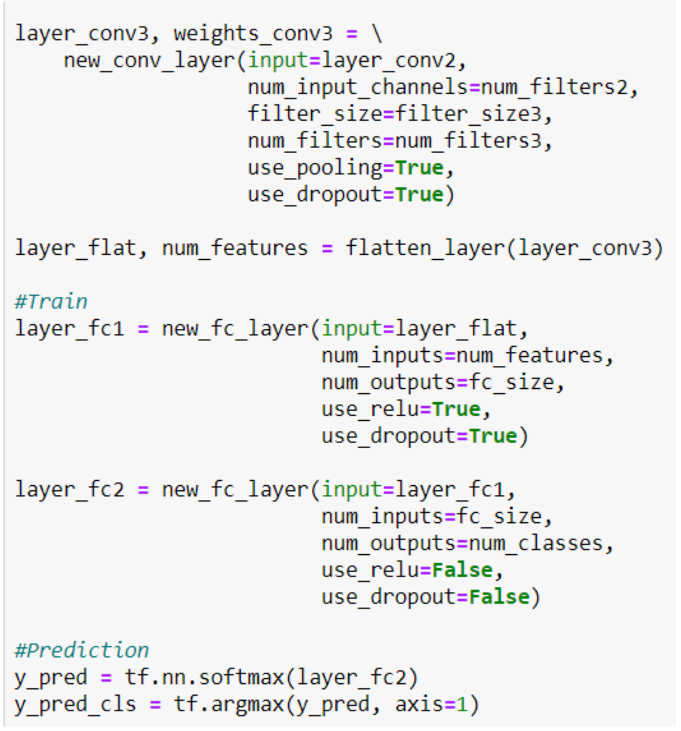
```
x = tf.placeholder(tf.float32, shape=[None, img_size, img_size, num_channels], name='x')
x_image = tf.reshape(x, [-1, img_size, img_size, num_channels]) #-1 put everything as 1 array
y_true = tf.placeholder(tf.float32, shape=[None, num_classes], name='y_true')
y_true_cls = tf.argmax(y_true, axis=1)
keep_prob_fc=1-tf.placeholder(tf.float32)
keep_prob_conv=1-tf.placeholder(tf.float32)
```

5. CNN layer design with TensorFlow

In this part you can vary with the filter sizes and the number of filters.



```
# Convolutional Layer 1.
filter_size1 = 5          # Convolution filters are 5 x 5 pixels.
num_filters1 = 32         # There are 32 of these filters.


# Convolutional Layer 2.
filter_size2 = 4          # Convolution filters are 4 x 4 pixels.
num_filters2 = 64      # There are 64 of these filters.


# Convolutional Layer 3.
filter_size3 = 3          # Convolution filters are 3 x 3 pixels.
num_filters3 = 128      # There are 128 of these filters.


# Fully-connected layer.
fc_size = 500

layer_conv1, weights_conv1 = \
    new_conv_layer(input=x_image,
                   num_input_channels=num_channels,
                   filter_size=filter_size1,
                   num_filters=num_filters1,
                   use_pooling=True,
                   use_dropout=False)

layer_conv2, weights_conv2 = \
    new_conv_layer(input=layer_conv1,
                   num_input_channels=num_filters1,
                   filter_size=filter_size2,
                   num_filters=num_filters2,
                   use_pooling=True,
                   use_dropout=False)
```



```
layer_conv3, weights_conv3 = \
    new_conv_layer(input=layer_conv2,
                   num_input_channels=num_filters2,
                   filter_size=filter_size3,
                   num_filters=num_filters3,
                   use_pooling=True,
                   use_dropout=True)

layer_flat, num_features = flatten_layer(layer_conv3)

#Train
layer_fc1 = new_fc_layer(input=layer_flat,
                         num_inputs=num_features,
                         num_outputs=fc_size,
                         use_relu=True,
                         use_dropout=True)

layer_fc2 = new_fc_layer(input=layer_fc1,
                         num_inputs=fc_size,
                         num_outputs=num_classes,
                         use_relu=False,
                         use_dropout=False)

#Prediction
y_pred = tf.nn.softmax(layer_fc2)
y_pred_cls = tf.argmax(y_pred, axis=1)
```

6. Loss of cross entropy with CNN with TensorFLow

The loss function is defined to train the program

```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=layer_fc2, labels=y_
cost = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(cost)
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

7. Training the model
The neural network is trained

```
session = tf.Session()
def init_variables():
    session.run(tf.global_variables_initializ
```

With the following function a batch is created from a data set we use a batch to train our model or program

```
batch_size = 50

#function next_batch
def next_batch(num, data, labels):
    '''
    Return a total of `num` random samples and labels.
    '''
    idx = np.arange(0 , len(data))
    np.random.shuffle(idx)
    idx = idx[:num]
    data_shuffle = [data[i] for i in idx]
    labels_shuffle = [labels[i] for i in idx]


    return np.asarray(data_shuffle), np.asarray(labels_sh
```

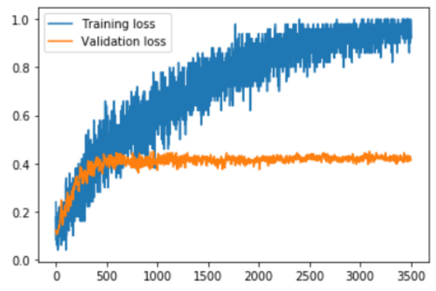
```
def optimize(num_iterations, X):
    global total_iterations

    start_time = time.time()

    #array to plot
    losses = {'train':[], 'validation':[]}

    for i in range(num_iterations):
            total_iterations += 1
            # Get a batch of training examples.
            # x_batch now holds a batch of images and
            # y_true_batch are the true labels for those images.
            x_batch, y_true_batch = next_batch(batch_size, X_train, y_trai


            # Put the batch into a dict with the proper names
            # for placeholder variables in the TensorFlow graph.
            feed_dict_train = {x: x_batch,
                               y_true: y_true_batch,
                               keep_prob_conv : 0.3,
                               keep_prob_fc : 0.4}
            feed_dict_validation = {x: X_validation,
                               y_true: y_validation,
                               keep_prob_conv : 1,
                               keep_prob_fc : 1}

            # Run the optimizer using this batch of training data.
            # TensorFlow assigns the variables in feed_dict_train
            # to the placeholder variables and then runs the optimizer.
            session.run(optimizer, feed_dict=feed_dict_train)

            acc_train = session.run(accuracy, feed_dict=feed_dict_train)
            acc_validation = session.run(accuracy, feed_dict=feed_dict_val
            losses['train'].append(acc_train)
            losses['validation'].append(acc_validation)
```

```
batch_size = 50

#function next_batch
def next_batch(num, data, labels):
    '''
    Return a total of `num` random samples and labels.
    '''
    idx = np.arange(0 , len(data))
    np.random.shuffle(idx)
    idx = idx[:num]
    data_shuffle = [data[i] for i in idx]
    labels_shuffle = [labels[i] for i in idx]


    return np.asarray(data_shuffle), np.asarray(labels_sh
```

```
            # Print status every X iterations.
            if (total_iterations % X == 0) or (i ==(num_iterations -1)):
            # Calculate the accuracy on the training-set.

                msg = "Iteration: {0:>6}, Training Accuracy: {1:>6.1%}, Validation Accuracy: {2:>6.1%}"
                print(msg.format(total_iterations, acc_train, acc_validation))


    # Ending time.
    end_time = time.time()


    # Difference between start and end-times.
    time_dif = end_time - start_time


    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_dif)))))

    plt.plot(losses['train'], label='Training loss')
    plt.plot(losses['validation'], label='Validation loss')
    plt.legend()
    _ = plt.ylim()
```

```
init_variables()
total_iterations = 0
optimize(num_iterations=3500, X=250)
```

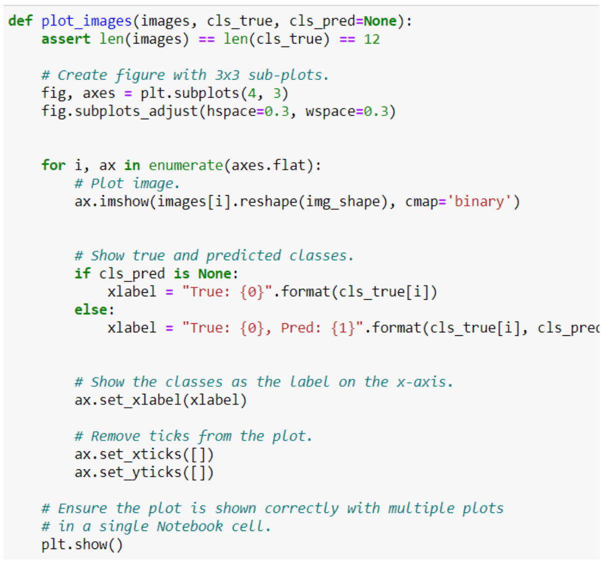
```
Iteration:    250, Training Accuracy:  30.0%, Validation Accuracy:  37.5%
Iteration:    500, Training Accuracy:  40.0%, Validation Accuracy:  39.0%
Iteration:    750, Training Accuracy:  62.0%, Validation Accuracy:  38.6%
Iteration:   1000, Training Accuracy:  56.0%, Validation Accuracy:  37.5%
Iteration:   1250, Training Accuracy:  64.0%, Validation Accuracy:  41.2%
Iteration:   1500, Training Accuracy:  78.0%, Validation Accuracy:  42.2%
Iteration:   1750, Training Accuracy:  74.0%, Validation Accuracy:  41.5%
Iteration:   2000, Training Accuracy:  94.0%, Validation Accuracy:  41.5%
Iteration:   2250, Training Accuracy:  88.0%, Validation Accuracy:  41.5%
Iteration:   2500, Training Accuracy:  90.0%, Validation Accuracy:  42.2%
Iteration:   2750, Training Accuracy:  98.0%, Validation Accuracy:  42.6%
Iteration:   3000, Training Accuracy:  96.0%, Validation Accuracy:  42.2%
Iteration:   3250, Training Accuracy:  96.0%, Validation Accuracy:  43.3%
Iteration:   3500, Training Accuracy:  92.0%, Validation Accuracy:  41.5%
Time usage: 0:08:54
```



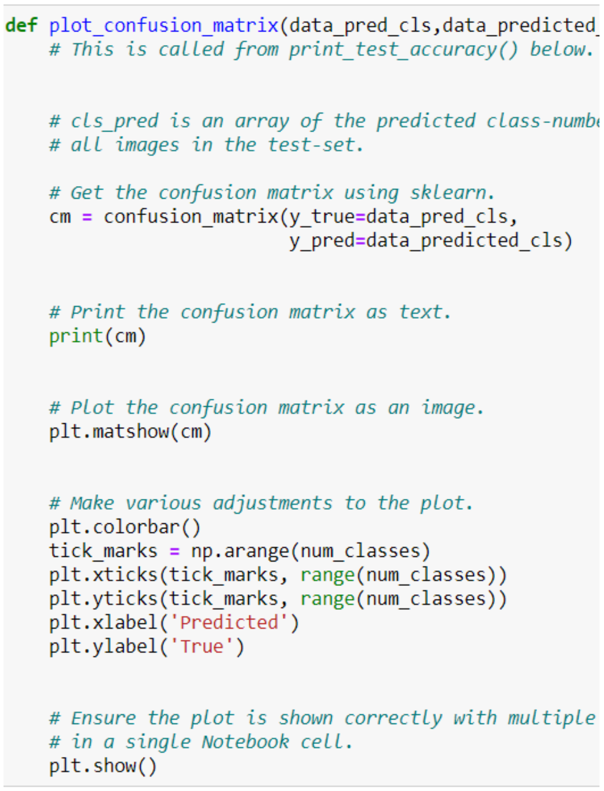
As can be seen in the graph given by the program, it can be seen that the model tends to over adjust which is not very good.

8. CNN with TensroFLow
Unfortunately, the results obtained are not as good as the accuracy percentage is 44% but this result can be improved if more images of the breeds of dogs with which the program is working are provided.

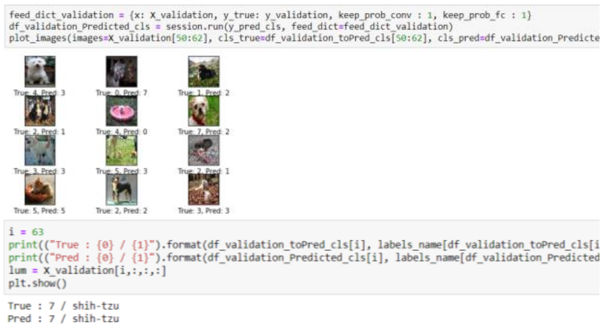
Below are some of the images of the new test data with the corresponding races and the predicted races.



```
def plot_images(images, cls_true, cls_pred=None):
    assert len(images) == len(cls_true) == 12

    # Create figure with 3x3 sub-plots.
    fig, axes = plt.subplots(4, 3)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)


    for i, ax in enumerate(axes.flat):
        # Plot image.
        ax.imshow(images[i].reshape(img_shape), cmap='binary')


        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i], cls_prec


        # Show the classes as the label on the x-axis.
        ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```



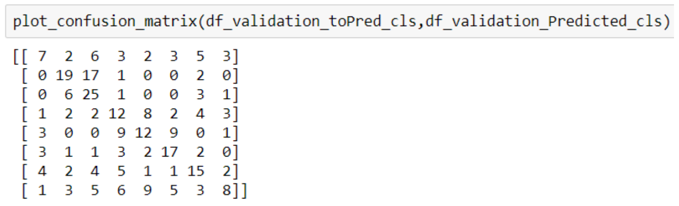
```
def plot_confusion_matrix(data_pred_cls,data_predicted_
    # This is called from print_test_accuracy() below.


    # cls_pred is an array of the predicted class-numbe
    # all images in the test-set.

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=data_pred_cls,
                          y_pred=data_predicted_cls)


    # Print the confusion matrix as text.
    print(cm)


    # Plot the confusion matrix as an image.
    plt.matshow(cm)


    # Make various adjustments to the plot.
    plt.colorbar()
    tick_marks = np.arange(num_classes)
    plt.xticks(tick_marks, range(num_classes))
    plt.yticks(tick_marks, range(num_classes))
    plt.xlabel('Predicted')
    plt.ylabel('True')


    # Ensure the plot is shown correctly with multiple
    # in a single Notebook cell.
    plt.show()
```



```
feed_dict_validation = {x: X_validation, y_true: y_validation, keep_prob_conv : 1, keep_prob_fc : 1}
df_validation_Predicted_cls = session.run(y_pred_cls, feed_dict=feed_dict_validation)
plot_images(images=X_validation[50:62], cls_true=df_validation_toPred_cls[50:62], cls_pred=df_validation_Predicte
```

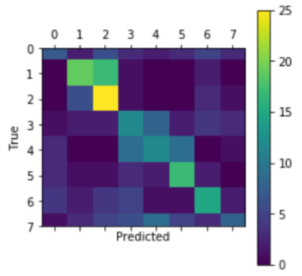
```
i = 63
print(("True : {0} / {1}").format(df_validation_toPred_cls[i], labels_name[df_validation_toPred_cls[i
print(("Pred : {0} / {1}").format(df_validation_Predicted_cls[i], labels_name[df_validation_Predicted
lum = X_validation[i,:,:,:]
plt.show()
```

```
True : 7 / shih-tzu
Pred : 7 / shih-tzu
```



```
plot_confusion_matrix(df_validation_toPred_cls,df_validation_Predicted_cls)
```

```
[[ 7  2  6  3  2  3  5  3]
 [ 0 19 17  1  0  0  2  0]
 [ 0  6 25  1  0  0  3  1]
 [ 1  2  2 12  8  2  4  3]
 [ 3  0  0  9 12  9  0  1]
 [ 3  1  1  3  2 17  2  0]
 [ 4  2  4  5  1  1 15  2]
 [ 1  3  5  6  9  5  3  8]]
```



V. CONCLUSIONS

it can be concluded that the neural networks in their operating capacities allow us to make programs that can facilitate us to recognize the recognition of data without knowing what it is thanks to the training