Trash Classification Using Convolutional Network Upon Colored Image:
Model Training, Testing and Comparing

Yisi Zou 1001826321
Yizhou Liu 1001139822
Lei Hua Huang 1001538204
Tianyi Xie 1000570679

1. Introduction

An enormous amount of trashes has been generated world-widely every day, especially in cities. Those trashes are in different materials that are required to be classified and processed accordingly. Improper classification and recycling of trashes could lead to environmental and economic losses. Therefore, segregation of trashes has been an important aspect of trash management for a sustainable society. The current trash classification process relies on sorting manually by human hand, which is fairly inefficient. In this case, we are proposing a system of trash segregation that intends to automatically receive images of a single trash and classify them into recycling material types.

In our design, a Machine Learning algorithm of convolutional neural network (CNN) is implemented to segregate the input images of trashes into six categories: metal, paper, plastic, cardboard, glasses and trash. The input image dataset is enlarged to a functional size by using data augmentation. Residual network (ResNet) will be used to proceed with the classification process.

2. Background

2.1 Convolutional Neural Network Architecture

We found five commonly used architectures of convolutional neural networks according to our research. The three classic networks of them are LeNet-5, AlexNet, and VGG, and the two modern networks are ResNet and Inception NN.

LeNet-5 is the simplest model among these five networks and is the most comfortable one for handling, however, it can only recognize the grey scaled images and is designed to classify handwritten digits [1]. *AlexNet* is similar to LeNet, but it is much larger in model scale. It contains 60 million parameters, and most importantly, it is capable of working with color images [2]. Thus, this CNN architecture would be one of the candidates for trash sorting. *VGG* applies smaller filters and it is deeper, which makes it more accurate than AlexNet[3]. *ResNet* allows people to train very deep networks, even those have more than 100 layers. In theory, very deep networks could represent very complex functions, which are useful in real life cases. However, they are hard to train because of vanishing and exploding gradient types of problems. In ResNet process, by skipping connections, it takes the activation from one layer and suddenly feed it to another layer even much deeper in the neural networks [4]. *Inception NN* would provide higher classification accuracy than all the four networks mentioned above, nevertheless, it is much deeper and requires multiple receptive field sizes for convolution, which will cause extremely long training time [5]. In this sense, Inception is not a good choice for our project. In order to balance between classification accuracy and length of training time, we chose ResNet neural network.

2.2 Related Work

In the past few years, there has been many neural network research projects focusing on the image classification, however, only a few of them concentrated on the trash classification.

A previous project on waste sorting we have discovered was a work done by Mindy Yang and Gary Thung from Stanford University. They used a support vector machine (SVM) with SIFT features and an 11-layer convolutional neural network (CNN), which is similar to AlexNet but smaller. The objective of their experiment is to use SVM to classify the trash into recycling categories first, and then use CNN to

do the six-category classification. Each image data contains exactly one object of waste with white background. According to the report, they achieved a classification accuracy of 63% for the trained SVM and only 22% for the trained CNN, while in their corresponding GitHub repository, the accuracy for CNN becomes 75% [6]. Since the repository might updated after their paper was dated, we assume their final accuracy was 75% and our intention is to train new network with higher accuracy.

2.3 Transfer Learning

In our project, although we planned to use the same trash image dataset as the existing project, we decided not to apply a transfer learning. The reason is that the classification accuracy for CNN in existing project is too low for a trained neural network. When we investigated their resizing process, we found out that they just simply modified all the images into the same size regardless of the position of object in each image and also did not enlarge the number of images. This careless resizing resulted in an invalid dataset, which would have a very negative influence on the model training and the recognition accuracy.

Figure2.1 examples of invalid cropped image in previous work [6]

In our case, after enlarging the original dataset, we would receive a valid dataset that contains more than 40,000 identical images, which is not regarded as large dataset for the neural network training. Rather than reuse the parameters of a trained model that has much simpler architecture than ResNet and has lower accuracy, training our own model would be a better choice.

3. Dataset and Data Enlargement

The original dataset is trashnet.zip and is obtained from a repository of Gary Thung, who did the previous project, for research purposes. The dataset contains 2527 images of different sizes that span six classes: metal, paper, plastic, cardboard, glasses and trash [6]. Since the size of the original dataset is not large enough for training, we decided to enlarge and resize the dataset by cropping 9 times and flipping 1 time for each image. Thus, the new dataset will consist 2527*18 = 45486 images of the same size (64*64 pixels). To illustrate, one of the original input images is provided below in *Figure 3.1*:

Figure 3.1 Original image of a can

The given image is a can of pizza sauce with size of 654*874. After cropping for 9 times, we obtain the following images:

Figure 3.2 Cropped images

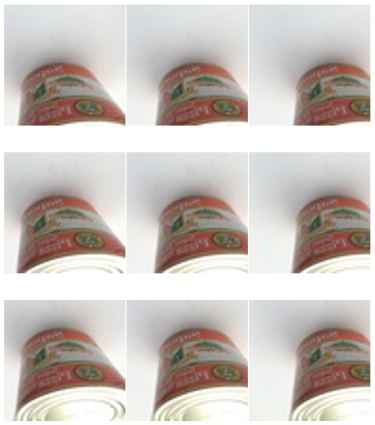

Figure 3.3 Flipped images

Then we may flip the cropped images to get a new set of 9 images of the can shown in *Figure 3.3*. Together with the images in *Figure3.2*, we now have 18 available images of the can in total.

The resulting images are all in a perfect state to be trained, with white background and the object clearly shown.

After using data augmentation techniques on our entire original dataset, we obtained a new dataset consisting 45,486 images of the same size, 64*64 pixels, which is more capable for a deep neural network training problem.

4. Model and Experiments

4.1 Model

In our work, we used ResNet50 to construct our CNN. ResNet is an architecture that can deal with vanishing gradient issue once and for all. The main idea behind ResNet is shortcut connection, which skips one or more layers. This shortcut enables us to train very deep networks efficiently [4].

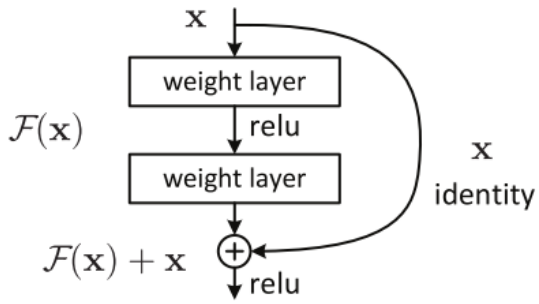

Figure 4.1 A residual block [4]

Instead of learning through a main path that directly map from x to y with a function H(x), we defined a residual mapping function, which is denoted by F(x) = H(x) – x. After reframing, the residual mapping function would become H(x) = F(x) + x, where F(x) represents the stacked non-linear layers and x represents the identity function. The hypothesis for this model is that optimizing the residual function is much easier than optimizing the original underlying mapping [4].

The implementation of the popular ResNet50 full architecture shows below [4]:
CONV2D -> BATCHNORM -> RELU -> MAXPOOL -> CONVBLOCK -> IDBLOCK*2 -> CONVBLOCK -> IDBLOCK*3 -> CONVBLOCK -> IDBLOCK*5 -> CONVBLOCK -> IDBLOCK*2 -> AVGPOOL -> TOPLAYER

4.2 Experiment

The ResNet CNN was trained with 40,500 training samples and 4,986 test samples. The dimension of each color image is 64x64x3. In order to avoid an underfitting or overfitting, we used 20 epochs and the batch size is 405. It takes 99673 seconds to achieve our final testing results.

5. Testing Result

After training our ResNet50 network with 20 epochs, we got the following results:

epoch	training loss	training accuracy (%)	test loss	test accuracy (%)	time spent (s)
1	1.467	49.76	1.382	51.89	4926
2	1.088	58.04	1.001	64.16	4981
3	0.806	71.55	0.760	72.42	4936
4	0.605	79.02	0.639	76.19	4969
5	0.399	85.75	0.588	80.08	4968
6	0.263	91.03	0.531	84.08	4976
7	0.162	94.52	0.461	86.44	4996
8	0.103	96.36	0.435	88.33	4971
9	0.0907	96.84	0.417	89.45	4952
10	0.0691	97.62	0.399	90.05	4956
11	0.0545	98.25	0.321	91.67	4998
12	0.0452	98.50	0.314	92.44	5073
13	0.0377	98.76	0.427	90.25	5168
14	0.0348	98.85	0.330	92.38	5007
15	0.0333	98.90	0.320	92.36	4935
16	0.0326	98.87	0.298	92.68	4970
17	0.0323	98.84	0.267	93.30	4987
18	0.0377	98.65	0.304	92.35	4974
19	0.0231	99.21	0.236	93.98	4996
20	0.0339	98.87	0.235	94.05	4934
					99673

Figure 5.1 Table of loss, accuracy and time for each epoch

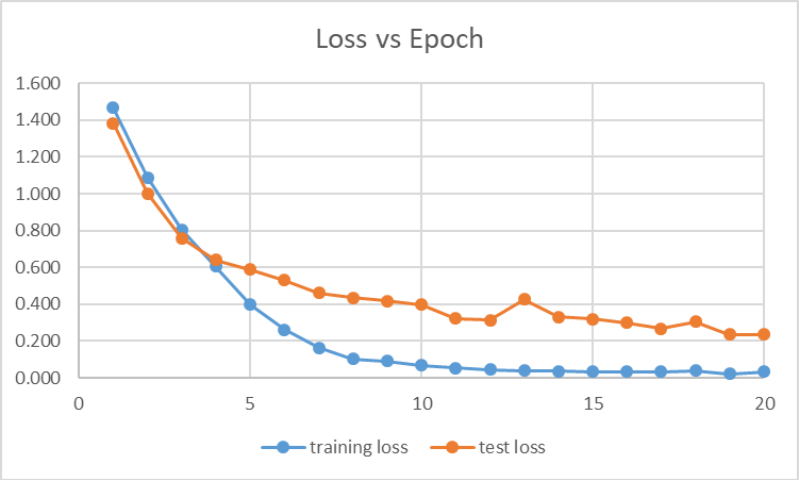

Figure 5.2 Training and test loss vs. epoch

As the number of epochs increases, the training loss and test loss decline rapidly at first and then converge after 15 epochs. After 20 epochs, the training loss becomes 0.0339 and the test loss becomes 0.235. Since both training and test losses are small and the gap between them is also small, we may conclude that our model is well fitted.

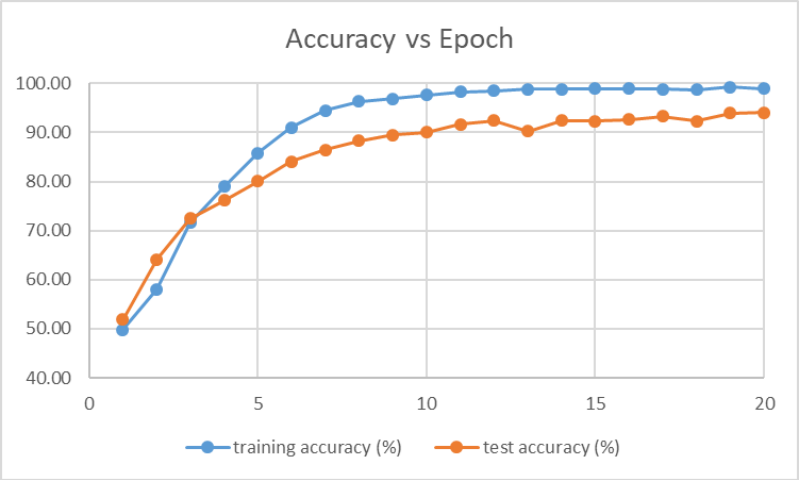

Figure 5.3 Training and test accuracy vs. epoch

As the number of epochs increases, the training accuracy and testing accuracy increase correspondingly, and converge after 15 epochs. Finally, with 20 epochs, the training accuracy is 98.87% and test accuracy is 94.05%.

6. Conclusion:

Supported by our enlarged data, the training set we used, known as ResNet50, is better than the original model given by the dataset provider. We finally get around 94% test accuracy, which is much higher than 75% test accuracy given by the dataset provider, and the model itself can be regarded as well fitted.

For further improvement, more datasets are recommended to be trained and validated for better results. Another way is to develop the network with a deeper Inception NN, which will be considered under more efficient running environment and more sufficient time.

References:

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
[2] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In NIPS, 2012.
[3] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015.
[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.
[6] Garythung, "garythung/trashnet," *GitHub*, 10-Apr-2017. [Online]. Available: https://github.com/garythung/trashnet. [Accessed: 18-Dec-2018].

Appendix:

1.resize.py

```
import os
import glob
from wand.image import Image

load_root = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'dataset')
save_root = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'resized_dataset')
categories = ['cardboard', 'glass', 'metal', 'paper', 'plastic', 'trash']
full_width = 80
full_height = 80
factors = [(0, 0.8), (0.1, 0.9), (0.2, 1)]
flop = [False, True]

for category in categories :
   counter = 0
   load_path = os.path.join(load_root, category, '*g')
   files = glob.glob(load_path)

   for file in files :
      with Image(filename=file) as img:
         img.resize(full_width, full_height)

         for is_flop in flop :
            if is_flop :
               img.flop()

            for width_start, width_end in factors :
               for height_start, height_end in factors :
                  counter += 1
                  save_name = category + str(counter) + '.jpg'
                  save_path = os.path.join(save_root, category, save_name)

                  with img.clone() as next_img :
                     next_img.crop(int(full_width*width_start), int(full_height*height_start),
                              int(full_width*width_end), int(full_height*height_end))
                     next_img.format = 'jpg'
                     next_img.save(filename=save_path)
```

2.data.py

```
import os
import glob
from PIL import Image
from keras.preprocessing import image
import numpy as np
```

```
import time

def convert_to_one_hot(Y, C):
    Y = np.eye(C)[Y.reshape(-1)].T
    return Y

load_root = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'resized_dataset')
save_root = os.path.dirname(os.path.abspath(__file__))
categories = ['cardboard', 'glass', 'metal', 'paper', 'plastic', 'trash']
Y_label = 0
image_height = 64
image_width  = 64

for category in categories :
    load_path = os.path.join(load_root, category, '*g')
    files = glob.glob(load_path)

    dataset_X = []
    dataset_Y = []
    counter = 0

    for file in files :
        img = Image.open(file)
        temp = image.img_to_array(img)
        temp = np.reshape(temp, (-1,1)).tolist()
        dataset_X.append(temp)
        dataset_Y.append(Y_label)
        counter += 1

    print(categories[Y_label], "finished!")
    Y_label          += 1

    dataset_X = np.array(dataset_X)
    dataset_X = np.reshape(dataset_X, (-1,image_height, image_width, 3))
    dataset_X = dataset_X / 255
    dataset_Y = np.array(dataset_Y)
    dataset_Y = np.reshape(dataset_Y, (1, -1))
    dataset_Y = convert_to_one_hot(dataset_Y, 6).T

    print ("dataset_X shape: " + str(dataset_X.shape))
    print ("dataset_Y shape: " + str(dataset_Y.shape))

    save_path = os.path.join(save_root, category + "_dataset_X.npy")
    np.save(save_path, dataset_X)
    save_path = os.path.join(save_root, category + "_dataset_Y.npy")
    np.save(save_path, dataset_Y)
```

3. ResNet.py

```
import numpy as np
from keras import layers
from keras.layers import Input, Add, Dense, Activation, ZeroPadding2D, BatchNormalization, Flatten,
Conv2D, AveragePooling2D, MaxPooling2D, GlobalMaxPooling2D
from keras.models import Model, load_model
from keras.utils import layer_utils
from keras.utils.data_utils import get_file
from keras.applications.imagenet_utils import preprocess_input
from keras.utils.vis_utils import model_to_dot
from keras.utils import plot_model
from keras.initializers import glorot_uniform

import os

import keras.backend as K
K.set_image_data_format('channels_last')
K.set_learning_phase(1)

load_root = os.path.dirname(os.path.abspath(__file__))

# GRADED FUNCTION: identity_block

def identity_block(X, f, filters, stage, block):
    """
    Implementation of the identity block as defined in Figure 3

    Arguments:
    X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
    f -- integer, specifying the shape of the middle CONV's window for the main path
    filters -- python list of integers, defining the number of filters in the CONV layers of the main path
    stage -- integer, used to name the layers, depending on their position in the network
    block -- string/character, used to name the layers, depending on their position in the network

    Returns:
    X -- output of the identity block, tensor of shape (n_H, n_W, n_C)
    """

    # defining name basis
    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    # Retrieve Filters
    F1, F2, F3 = filters

    # Save the input value. You'll need this later to add back to the main path.
    X_shortcut = X
```

```
    # First component of main path
    X = Conv2D(filters = F1, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name = conv_name_base
+ '2a', kernel_initializer = glorot_uniform())(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2a')(X)
    X = Activation('relu')(X)

    # Second component of main path
    X = Conv2D(filters = F2, kernel_size = (f, f), strides = (1,1), padding = 'same', name = conv_name_base
+ '2b', kernel_initializer = glorot_uniform())(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
    X = Activation('relu')(X)

    # Third component of main path
    X = Conv2D(filters = F3, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name = conv_name_base
+ '2c', kernel_initializer = glorot_uniform())(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)

    # Final step: Add shortcut value to main path, and pass it through a RELU activation
    X = Add()([X_shortcut, X])
    X = Activation('relu')(X)

    return X

# GRADED FUNCTION: convolutional_block

def convolutional_block(X, f, filters, stage, block, s = 2):
    """
    Implementation of the convolutional block as defined in Figure 4

    Arguments:
    X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
    f -- integer, specifying the shape of the middle CONV's window for the main path
    filters -- python list of integers, defining the number of filters in the CONV layers of the main path
    stage -- integer, used to name the layers, depending on their position in the network
    block -- string/character, used to name the layers, depending on their position in the network
    s -- Integer, specifying the stride to be used

    Returns:
    X -- output of the convolutional block, tensor of shape (n_H, n_W, n_C)
    """

    # defining name basis
    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    # Retrieve Filters
    F1, F2, F3 = filters
```

```
    # Save the input value
    X_shortcut = X

    ##### MAIN PATH #####
    # First component of main path
    X = Conv2D(F1, (1, 1), strides = (s,s), name = conv_name_base + '2a', kernel_initializer =
glorot_uniform())(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2a')(X)
    X = Activation('relu')(X)

    # Second component of main path
    X = Conv2D(F2, (f, f), strides = (1,1), padding = 'same', name = conv_name_base + '2b',
kernel_initializer = glorot_uniform())(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
    X = Activation('relu')(X)

    # Third component of main path
    X = Conv2D(F3, (1, 1), strides = (1,1), padding = 'valid', name = conv_name_base + '2c',
kernel_initializer = glorot_uniform())(X)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)

    ##### SHORTCUT PATH ####
    X_shortcut = Conv2D(F3, (1, 1), strides = (s,s), name = conv_name_base + '1', kernel_initializer =
glorot_uniform())(X_shortcut)
    X_shortcut = BatchNormalization(axis = 3, name = bn_name_base + '1')(X_shortcut)

    # Final step: Add shortcut value to main path, and pass it through a RELU activation
    X = Add()([X_shortcut, X])
    X = Activation('relu')(X)

    return X

# GRADED FUNCTION: ResNet50

def ResNet50(input_shape = (64, 64, 3), classes = 6):
    """
    Implementation of the popular ResNet50 the following architecture:
    CONV2D -> BATCHNORM -> RELU -> MAXPOOL -> CONVBLOCK -> IDBLOCK*2 -> CONVBLOCK ->
IDBLOCK*3
    -> CONVBLOCK -> IDBLOCK*5 -> CONVBLOCK -> IDBLOCK*2 -> AVGPOOL -> TOPLAYER

    Arguments:
    input_shape -- shape of the images of the dataset
    classes -- integer, number of classes

    Returns:
    model -- a Model() instance in Keras
```

```
    """

    # Define the input as a tensor with shape input_shape
    X_input = Input(input_shape)

    # Zero-Padding
    X = ZeroPadding2D((3, 3))(X_input)

    # Stage 1
    X = Conv2D(64, (7, 7), strides = (2, 2), name = 'conv1', kernel_initializer = glorot_uniform())(X)
    X = BatchNormalization(axis = 3, name = 'bn_conv1')(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((3, 3), strides=(2, 2))(X)

    # Stage 2
    X = convolutional_block(X, f = 3, filters = [64, 64, 256], stage = 2, block='a', s = 1)
    X = identity_block(X, 3, [64, 64, 256], stage=2, block='b')
    X = identity_block(X, 3, [64, 64, 256], stage=2, block='c')

    # Stage 3
    X = convolutional_block(X, f = 3, filters = [128, 128, 512], stage = 3, block='a', s = 2)
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='b')
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='c')
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='d')

    # Stage 4
    X = convolutional_block(X, f = 3, filters = [256, 256, 1024], stage = 4, block='a', s = 2)
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='b')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='c')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='d')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='e')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='f')

    # Stage 5
    X = convolutional_block(X, f = 3, filters = [512, 512, 2048], stage = 5, block='a', s = 2)
    X = identity_block(X, 3, [512, 512, 2048], stage=5, block='b')
    X = identity_block(X, 3, [512, 512, 2048], stage=5, block='c')

    # AVGPOOL
    X = AveragePooling2D(pool_size=(2, 2), name = "avg_pool")(X)

    # output layer
    X = Flatten()(X)
    X = Dense(classes, activation='softmax', name='fc' + str(classes), kernel_initializer =
glorot_uniform())(X)


    # Create model
```

```
    model = Model(inputs = X_input, outputs = X, name='ResNet50')

    return model

################################################################################
# construct the model for the first time
model = ResNet50(input_shape = (64, 64, 3), classes = 6)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# load the model after resume
#model = load_model('A3_model_last.h5')

# load image data
load_path = os.path.join(load_root, "cardboard_dataset_X.npy")
dataset_X = np.load(load_path)
load_path = os.path.join(load_root, "cardboard_dataset_Y.npy")
dataset_Y = np.load(load_path)

rest_categories = ['glass', 'metal', 'paper', 'plastic', 'trash']

for category in rest_categories :
    load_path = os.path.join(load_root, category + "_dataset_X.npy")
    dataset_X = np.vstack((dataset_X, np.load(load_path)))
    load_path = os.path.join(load_root, category + "_dataset_Y.npy")
    dataset_Y = np.vstack((dataset_Y, np.load(load_path)))

# random shuffle the data
randIndx = np.arange(dataset_X.shape[0])
# make sure the taining set and test set does not change after resuming
np.random.seed(0)
np.random.shuffle(randIndx)
dataset_X, dataset_Y = dataset_X[randIndx], dataset_Y[randIndx]

# divide the dataset into training and test set
X_train = dataset_X[:40500]
Y_train = dataset_Y[:40500]
X_test = dataset_X[40500:]
Y_test = dataset_Y[40500:]

epoch_num = 20

print ("number of training examples = " + str(X_train.shape[0]))
print ("number of test examples = " + str(X_test.shape[0]))
print ("X_train shape: " + str(X_train.shape))
print ("Y_train shape: " + str(Y_train.shape))
print ("X_test shape: " + str(X_test.shape))
print ("Y_test shape: " + str(Y_test.shape))
```

```
for i in range(epoch_num) :
    # train each epoch, evaluate, and save the model
    model.fit(X_train, Y_train, epochs = 1, batch_size = 405)

    preds = model.evaluate(X_test, Y_test)
    print ("Test Loss = " + str(preds[0]))
    print ("Test Accuracy = " + str(preds[1]))

    save_name = 'A3_model'+ str(i) + '.h5'

    model.save(save_name)
```