Machine Learning Engineer Nanodegree

Capstone Project: Traffic Sign Recognition

Lang Su

August 22nd, 2018

I. Definition

We recognize images and patterns at every moment in our life. When individuals detect an object, it seems like our brain doesn't take a lot of efforts to tell what we are looking at. However, it did take a long time for each person to learn what is what since we were born, not only because there are so many different categories in the world for us to learn, but also due to the fact that our intelligence was growing gradually to support our learning process.

Object recognition[1] is a technology in the field of computer vision for finding and identifying objects in an image or video sequence. There are many well-researched domains of object detection include face detection and pedestrian detection. Many of them have been successfully achieved using artificial neural network[2].

Figure1: Face detection. Retrieved from https://towardsdatascience.com/face-detection-for-beginners-e58e8f21aad9

Artificial neural networks are computing systems vaguely inspired by the biological neural networks that constitute animal brains. Such systems mimic the learning process of living creatures, which do not need to be programmed with any task-specific rules. They "learn" to perform tasks from examples, e.g. learn to classify dog breeds with pictures of different dog breeds and their names (labels). Neural networks were deployed on a large scale, particularly in image and visual recognition problems, which are known as "deep learning[3]".

Figure 2: Traffic sign detection. Retrieved from http://www.mdpi.com/1424-8220/17/6/1207

The problem that this project is focusing on is one of the key tasks in autonomous vehicles areas -- Traffic Sign Recognition[4]. Traffic-sign recognition (TSR) is a technology by which a vehicle is able to recognize the traffic signs put on the road, such as "speed limit" or "turn ahead". Modern traffic-sign recognition systems are being developed using convolutional neural networks[5], which is a class of deep, feedforward artificial neural networks. The trained neural net can then be used in real life driving to detect traffic signs in real time. The predictions outputted by such systems will immediately be processed by other systems in self-driving car to make further decisions, such as whether it should perform a full-stop or drop its speed below 50 km/h.

Project Overview

This project is designed to find a solution to the traffic sign recognition problem in real life scenario. In specific, I will be mainly focusing on training artificial neural networks to classify traffic signs within the range of the given dataset.

Moreover, implemented solutions will be evaluated according to its efficiency, accuracy and utility. For example, how long does it take for the trained neural network to make one prediction, what is the accuracy score evaluated by defined metrics, how useful is the implementation in real life, etc. Adjustments to the methods or parameters may also be made to maximize the model strength.

Problem Statement

The problem to be solved is how to teach computers recognize traffic sign when it receives an image of a traffic sign. For instance, we want the machine to output “40 km/h speed limit” when receiving a given photo of such a traffic sign. Furthermore, we expect our solution still does a good job when the given traffic sign image isn’t in good quality, such as an image with a relatively small traffic sign, or an unclear picture.

The main method we are going to use is training a Convolutional Neural Network using traffic sign datasets. The main steps of the project can be divided into the following:

1. **Data exploration:** Discover and download traffic sign images in decent quality.
2. **Data preparation:** Reshape them into input data that can be feed into our neural network. Split them into training dataset, validation dataset and test dataset.
3. **Pre-trained model exploration:** We are going to use a pre-trained model as the “base” of our model. Thus we need to look for such a model that fits our problem.
4. **Fine-tone[6] the chosen model:** Fine-tone the chosen pre-trained model.
5. **Training:** Train our model using training dataset from step 2
6. **Testng:** Test our trained CNN using testing dataset from step 2
7. **Improve:** Adjust parameters to improve test score
8. **Evaluate:** Evaluate its comprehensive performance and explore potential improvements

By the end of this project, we are expected to see a well-trained traffic sign classifier that is capable of recognizing the traffic sign in any given related pictures.

Metrics

The evaluation metric that is used to quantify the performance of both the benchmark model and the solution model is given by the following formula:

$$Accuray\ score\ =\ \frac{\#\ Correct\ predictions}{\#\ tests} *\ 100\%$$

For example, if the model made 70 correct predictions given 100 tests, whereas the remaining 30 predictions don’t match the labels, the accuracy score is calculated as:

$$Accuracy\ score\ =\ \frac{70}{100} *\ 100\%\ =\ 70\%$$

II. Analysis

Data Exploration

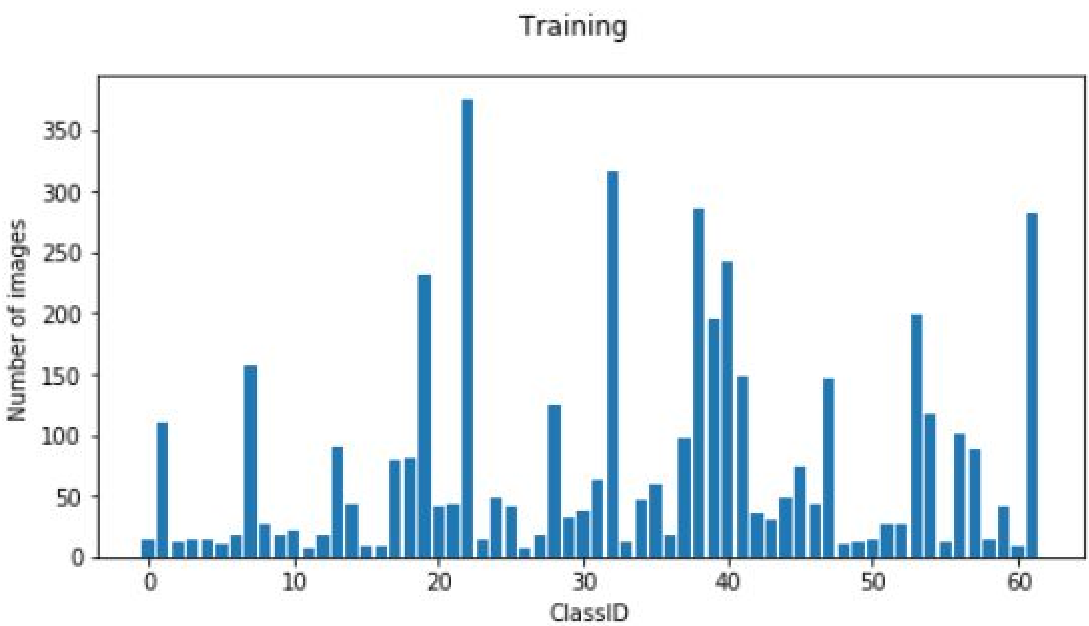
The dataset I used is a reduced subset of the Belgium Traffic Signs database. There are 62 classes in total, and there is one directory for each class. Each directory contains the corresponding images of the traffic sign in .ppm format (RGB color), as well as a csv file of annotations. Each image is already a cropped version of the actual photo that was taken in real life. There are training dataset and testing dataset already separated in advance.

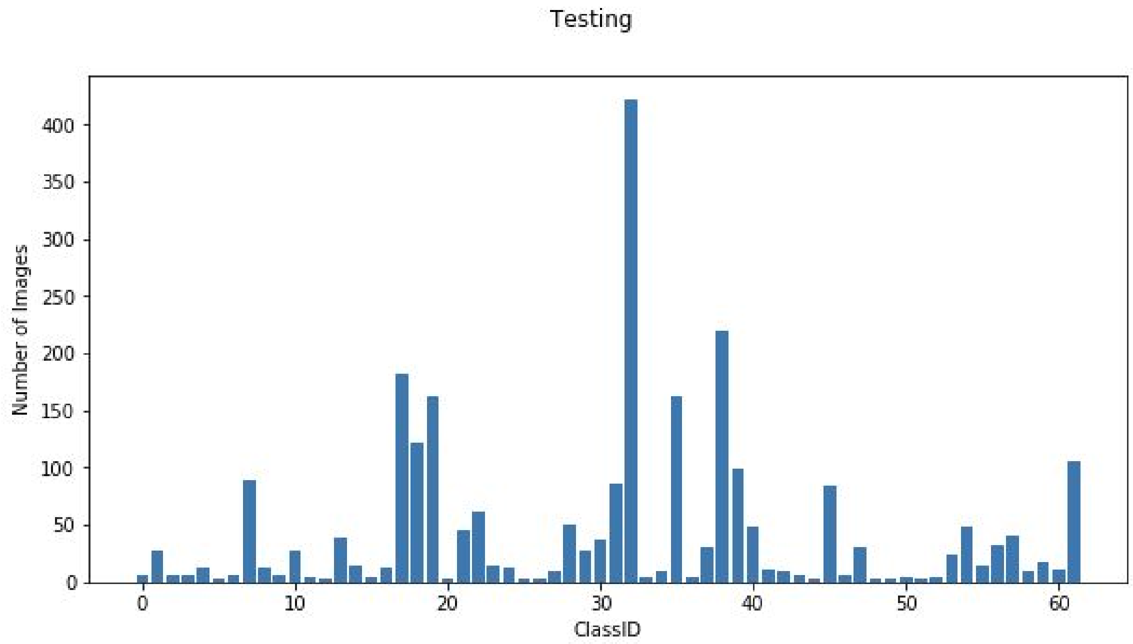
Note: the delimiter of the csv file is semicolon (;).
The csv file contains the following details,

Filename -- Image file name the following information applies to
Width, Height -- Dimensions of the image
Roi.x1,Roi.y1, Roi.x2,Roi.y2 -- Location of the sign within the image
ClassId -- The class of the traffic sign. **Note: ClassID starts from 0.**

Reference:
Radu Timofte*, Markus Mathias*, Rodrigo Benenson, and Luc Van Gool, Traffic Sign Recognition - How far are we from the solution?, International Joint Conference on Neural Networks (IJCNN 2013), August 2013, Dallas, USA.

Below are plots that shows the numbers of images of each class.





From these two plots, we realize that the data we got is not very evenly distributed. For instance, Class 22 has about 350 training images whereas class 0 only has about 5 images.

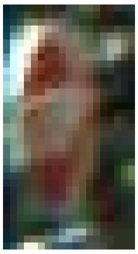
Statistics for datasets:

```
print('There are %d total traffic sign categories.' %
len(set(all_training_targets)))
print('There are %d total traffic sign images.' % len(traffic_sign_files_total))
print('There are %d training traffic sign images.' % len(all_training_targets))
print('There are %d validation traffic sign images.' %
len(all_validation_targets))
print('There are %d test traffic sign images.'% len(all_test_targets))


There are 62 total traffic sign categories.
There are 7125 total traffic sign images.
There are 4591 training traffic sign images.
There are 1267 validation traffic sign images.
There are 1267 test traffic sign images.
```

For display and future use, I convert all .ppm images into .png images and store them in another folder under the same structure.
The screenshot below shows several sample traffic sign images for class 00021.

Our dataset indeed has abnormal images such as dark and unclear images.




In my opinions, I don't think they need to be eliminated or fixed from the dataset. Our machine can still extract features (lines, curves...) from such dark or unclear images. It is expected to tell the traffic sign it's looking at under such conditions. Therefore, pictures like these won't hurt if the proportion is relatively small.

During training, we need all pictures to be converted into the same ratio (224 x 224). This means some images will be stretched vertically or horizontally. I don't think this is an issue because the features in the picture won't get distorted or ruined as the whole picture doesn't change significantly.

Algorithms and Techniques

Fine-tuning:

The technique that I am going to use to solve our problem is fine-tuning[6] a pre-trained deep learning model in Keras[7]. Fine tuning is a process to take a network model that has already been trained for a given task, train it on another similar task. The principle of fine-tuning is taking advantage of the feature extraction that happens in the front layers of the network without developing that feature extraction network from scratch.

The first thing we need to do, is to find a pre-trained model that fits our problem. We want to find such an model with pre-trained weights on a common dataset like the

ImageNet. I have finally chosen ResNet-50 Pre-trained Model for Keras, trained on ImageNet dataset.

In general, the technique of fine-tuning is replacing the output layer originally trained to recognize (in the case of imagenet models) 1,000 classes, with a layer that recognizes the number of classes we require (in our case 62 classes). After attaching the new output layer, the model is then trained to take the lower level features from the front of the network and map them to the desired output classes, using SGD (Stochastic Gradient Descent [8]).

Just like training other Neural Networks, we need to define the following parameters before training:

```
# row number of pixels for input_shape
img_row = 224
# col number of pixels for input_shape
img_col = 224
# three for RGB
channel = 3
# number of classes, in our case 62
num_class = len(set(all_training_targets))
# the number of training examples in one forward/backward pass.
Note: the higher the batch size, the more memory needed
batch_size = 16
# One Epoch is when an entire dataset is passed forward and
backward through the neural network only once.
epochs = 10
# Learning rate:
Lr = 0.01
# Decay: Learning rate decay over each update.
Decay = 1e-6
# momentum: Parameter that accelerates SGD in the relevant
direction and dampens oscillations.
momentum=0.9
```

Benchmark

The benchmark model for this project is **60%** accuracy score achieved on the test dataset. The score is calculated by the metric we defined previously.

III. Methodology

Data Preprocessing

In this section, I will show all the preparations I made before feeding training inputs into the model.

My first step is to convert every traffic sign picture as well as its information into a python object defined by this Traffic_Sign class.

```
class Traffic_Sign:
    """ A Single Traffic sign model """

    def __init__(self, filePath, fileName, width, height, x1, y1, x2,
y2, classTxt, classID):
        self.filePath = filePath
        self.fileName = fileName
        self.width = width
        self.height = height
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.classTxt = classTxt
        self.classID = classID
```

To load dataset into a list of such objects, I iterated through every annotation csv files. After shuffling the list of Traffic_Sign objects, I then extract filePath and classID out as “training files” and “targets”.

```
# a list of training traffic_sign_collection
all_training_objs = load_dataset(training_dataset_directory)

# a list of testing traffic_sign_collection
all_testing_and_validation_objs =
load_dataset(testing_dataset_directory)


all_training_files, all_training_targets =
extract_X_and_Y(all_training_objs)
```

Even though our dataset has already been divided into training dataset and testing dataset, we still need to manually take validation dataset. Compare to the number of training images, we seem to have an abundant number of test files. Therefore, I split the test dataset into validation dataset and testing dataset.

```
# extract testing files and targets and split into validation and test
X_temp, Y_temp = extract_X_and_Y(all_testing_and_validation_objs)

# merge X_temp and all_training_files to get total files
traffic_sign_files_total = X_temp + all_training_files
traffic_sign_targets_total = Y_temp + all_training_targets
# get all validation field and all test files
all_validation_files, all_test_files, all_validation_targets, all_test_targets = train_test_split(X_temp, Y_temp, test_size=0.5, random_state=42)
```

Next, we need to define a function to convert the path to the image (string) into tensor to fit the input shape -- a 3D array in (224, 224, 3). Any pictures in different ratio will be converted into 224 x 224.

```
def path_to_tensor(img_path):
    # loads RGB image as PIL.Image.Image type
    img = image.load_img(img_path, target_size=(224, 224))
    # convert PIL.Image.Image type to 3D tensor with shape (224, 224, 3)
    x = image.img_to_array(img)
    # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) and return 4D tensor
    return np.expand_dims(x, axis=0)

def paths_to_tensor(img_paths):
    list_of_tensors = [path_to_tensor(img_path) for img_path in tqdm(img_paths)]
    return np.vstack(list_of_tensors)
```

Now our training files are ready to be feeded in the model. However, we still need to oneHot-encode the targets. That is, we need a *num_of_targets x num_of_classes* 2-D array (In our training target case: 4591 x 62) , each row has the i-th element set to 1, and others set to 0, where i is the classID of that traffic sign.

```
def oneHotEncodeTargets(list_of_targets):
    onehot_encoded = list()
    for value in list_of_targets:
        letter = [0 for _ in range(62)]
        letter[int(value)] = 1
        onehot_encoded.append(letter)
    return np.array(onehot_encoded)

all_training_targets = oneHotEncodeTargets(all_training_targets)
all_test_targets = oneHotEncodeTargets(all_test_targets)
all_validation_targets =
oneHotEncodeTargets(all_validation_targets)
```

Implementation

Fine-Tuning ResNet50:

The following source code is from [ResNet50 implementation](#).

```
    x = ZeroPadding2D((3, 3))(img_input)
    x = Conv2D(64, (7, 7), name="conv1", strides=(2, 2))(x)
    x = BatchNormalization(axis=bn_axis, name='bn_conv1')(x)
    x = Activation('relu')(x)
    x = MaxPooling2D((3, 3), strides=(2, 2))(x)

    x = conv_block(x, 3, [64, 64, 256], stage=2, block='a',
strides=(1, 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')

    x = conv_block(x, 3, [128, 128, 512], stage=3, block='a')
    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')
```

```
    x = conv_block(x, 3, [256, 256, 1024], stage=4, block='a')
    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')

    x = conv_block(x, 3, [512, 512, 2048], stage=5, block='a')
    x = identity_block(x, 3, [512, 512, 2048], stage=5, block='b')
    x = identity_block(x, 3, [512, 512, 2048], stage=5, block='c')

    # original fully connected layer
    x_orig = AveragePooling2D((7, 7), name='avg_pool')(x)
    x_orig = Flatten()(x_orig)
    x_orig = Dense(1000, activation='softmax',
name='fcImageNet')(x_orig)

    # Create model
    model = Model(img_input, x_orig)
```

This fine-tune method doesn't use a "pop" method to remove the last layer. Instead, it connects to the final layer we want after loading the ImageNet weights.

```
    # load image net weight
    model.load_weights(imagenet_weights_path)

    # replace softmax layer for transfer learning
    x_new = AveragePooling2D((7, 7), name='avg_pool')(x)
    x_new = Flatten()(x_new)
    x_new = Dense(num_classes, activation='softmax',
name='fcTS')(x_new)

    model = Model(img_input, x_new)
```

Train:

```
TS_model.fit(train_tensors, all_training_targets,
batch_size=batch_size, epochs=epochs, shuffle=True, verbose=1,
validation_data=(valid_tensors, all_validation_targets))
```

Metric:

```
test_accuracy =
100*np.sum(np.array(ts_prediction)==np.argmax(all_test_targets,
axis=1))/len(ts_prediction)
print('Test accuracy: %.4f%%' % test_accuracy)
```

Refinement

Using the parameters I defined in “Algorithms and Techniques” section, the model achieved a **90%** accuracy score, which is pretty decent. The only refinement I made is changing number of epoths from **10** to **25**. The final accuracy score on testing dataset achieved **99.1%**.

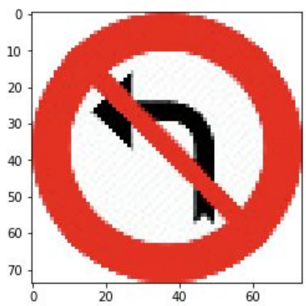
IV. Results

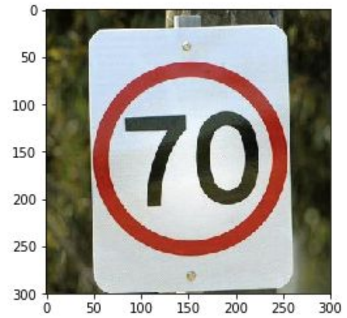
Model Evaluation and Validation

Below is some sample images downloaded from the Internet used for testing. I have classified them into different kinds of testing, so that we can see if the program is consistent:

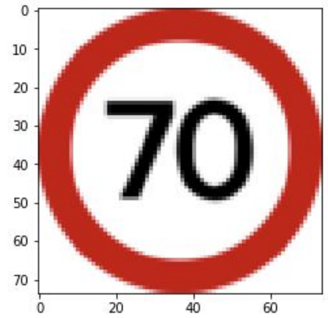
Easy recognition test: traffic sign occupies most of the space of images

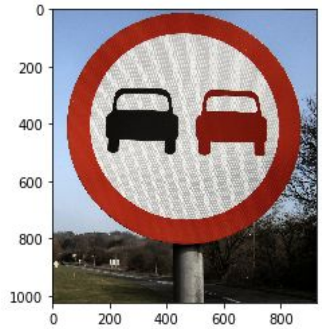

```
Confidence: 0.9993894. The classID for the traffic sign in this picture is: 29
Its class name in the database is: C31LEFT
Searching in the image database... It's referening to this one!
```



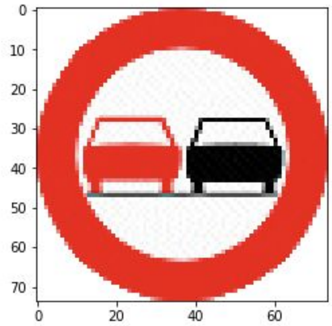



```
Confidence: 0.99999654. The classID for the traffic sign in this picture is: 32
Its class name in the database is: C43
Searching in the image database... It's referening to this one!
```

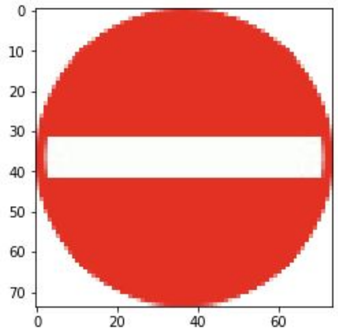




```
Confidence: 0.7758715. The classID for the traffic sign in this picture is: 31
Its class name in the database is: C35
Searching in the image database... It's referening to this one!
```



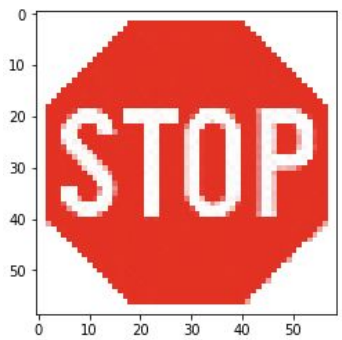
```
Confidence: 0.9469988. The classID for the traffic sign in this picture is: 22
Its class name in the database is: C1
Searching in the image database... It's referening to this one!
```

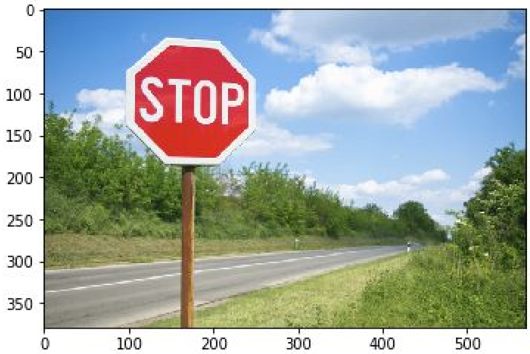


Our model is pretty solid when the target occupies more than half of a image. The confidence of prediction is generally greater than 90% for the samples I posted here as well as the tests I did locally. Some exceptions such as 3rd sample image have confidence lower than 80%. However, it's more because of the ambiguous color for the two cars in the picture. Overall, our model did a great job in this testing category.

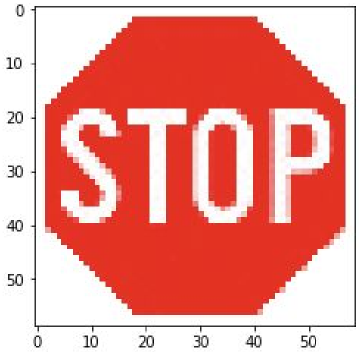
Small proportion test: traffic sign occupies less than half of the space of images

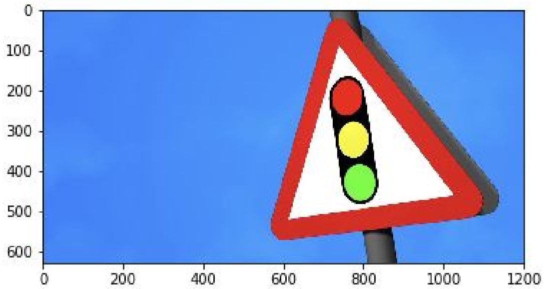

```
Confidence: 0.9919761. The classID for the traffic sign in this picture is: 21
Its class name in the database is: B5
Searching in the image database... It's referening to this one!
```



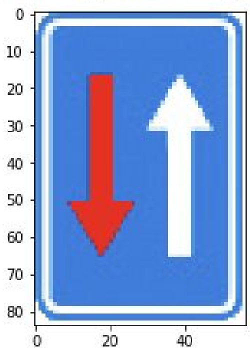



```
Confidence: 0.58831125. The classID for the traffic sign in this picture is: 21
Its class name in the database is: B5
Searching in the image database... It's referening to this one!
```



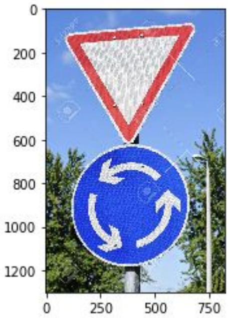



```
Confidence: 0.16885376. The classID for the traffic sign in this picture is: 44
Its class name in the database is: B21
Searching in the image database... It's referening to this one!
```

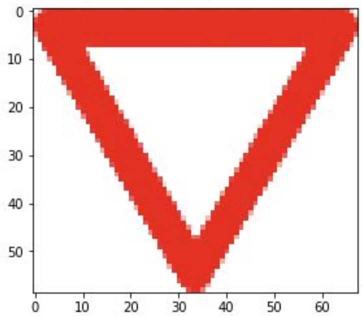


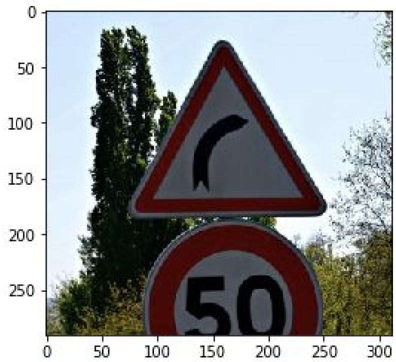
The model did a decent job most of the time, but with a lower confidence. Some wrong predictions can be found, such as the last sample. The main reason for this is that, the model takes the whole picture as input, and try to tell which traffic sign this picture looks like. For a model that have only “seen” cropped traffic sign images, tests from this section is a little bit hard for it, but still it performs not bad.

Multiple traffic signs test: more than one traffic signs in the image

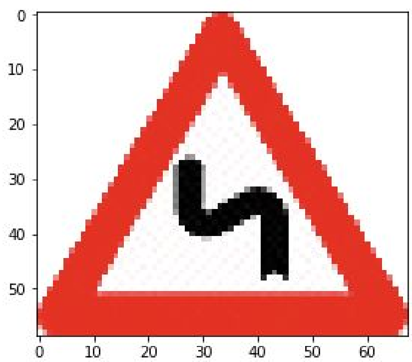


```
Confidence: 0.4284406. The classID for the traffic sign in this picture is: 19
Its class name in the database is: B1
Searching in the image database... It's referening to this one!
```






```
Confidence: 0.2042899. The classID for the traffic sign in this picture is: 5
Its class name in the database is: A1C
Searching in the image database... It's referening to this one!
```



Just like the previous test, the prediction comes with lower confidence and wrong predictions are made some time.

```
time_used = []
for eachfile in os.listdir(test_folder):
    start_time = time.time()
    if (eachfile[0]!='.'):
        predict_img(os.path.join(test_folder, eachfile))
    elapsed_time = time.time() - start_time
    time_used.append(elapsed_time)
print("Average time spend for 1 prediction:
{}".format(str(np.mean(time_used))))

# Output:
# Average time spend for 1 prediction: 0.5287921245281513
```

In terms of efficiency, the model takes an average of **0.53 second** to make one prediction, which is a reasonable time.

Justification

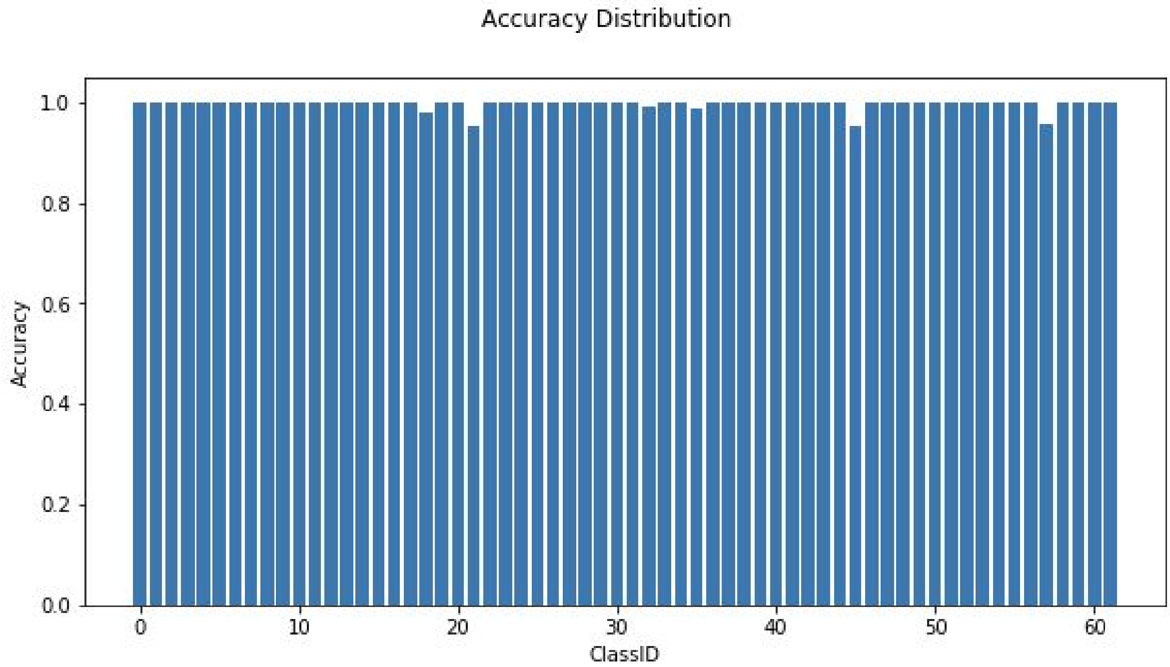
Compare to the benchmark we defined previously (60% accuracy score on testing dataset), our model performs way more better (99.1%).

The final model made expected prediction most of the time, especially when I tested using images that have traffic sign part taking most of the spaces. However, low confidence and wrong predictions were made when traffic sign are relatively smaller in the image, or there are multiple traffic signs in one image (the reasons are actually the same). Thus our model is robust to handle the job most of the cases when traffic sign occupies most of the image frame, but it is not reliable enough to be used in real time situation.

V. Conclusion

Free-Form Visualization

As we showed previously, the test score is 99.1%. I have made a accuracy distribution chart for all 62 classes.



As we can see, the model did a wonderful job on most of the classes.

Reflection

This capstone project not only helps me review my machine learning knowledges, but also offers me a great chance to gain hands-on experiences on a formal machine learning project.

Through this project, I found many interesting parts that I really enjoy. For example, manipulating dataset to fit in model, fine-tuning pre-trained model, analyzing and plotting data to see the statistics, etc…

The difficult part I found in this project, is searching the most correct and efficient way of doing what I want to do. For example, I took a long time to figure out what is the best way to fine-tune a ResNet50 model. It is not hard to make it work, but it takes time to figure out why it is implemented that way.

The final model and solution fits my expectations. However, it definitely needs improvements. I am confident to say I build a successful traffic sign classifier, but it needs more work to become a real-time traffic sign detector and classifier. This will be the next step I am working on.

Improvement

As I discussed in the previous section, the most important improvement is real-time detection support. Through my searches, it is difficult to make such an improvement on the model that we already trained. There are some mature object detection implementations such as object detection API of Tensorflow, YOLO etc… Therefore, this improvement may require changing the deep learning network we had.

Another improvement I could have made, is increasing the number of training dataset of classes that have less images. From the statistics listed in the report, I realized that the dataset is unbalanced. I could either replace our dataset with a better comprehensive dataset, or manually edit and add images to the dataset.

Reference

[1]: https://en.wikipedia.org/wiki/Outline_of_object_recognition
[2]: https://en.wikipedia.org/wiki/Artificial_neural_network
[3]: https://en.wikipedia.org/wiki/Deep_learning
[4]: https://en.wikipedia.org/wiki/Traffic-sign_recognition
[5]: https://en.wikipedia.org/wiki/Convolutional_neural_network
[6]: http://wiki.fast.ai/index.php/Fine_tuning
[7]: https://en.wikipedia.org/wiki/Keras
[8]: https://en.wikipedia.org/wiki/Stochastic_gradient_descent