

Udacity Self-Driving Car Engineer

Report

- **Project:** Traffic Sign Classifier
- **Name:** Shao Hongxu

Let me just use some brief sentence to explain the thing I did, basically these are 3 big parts here:

Part 1: import necessary lib

Part 2: identify Main function

Part 3: calculate model prediction accuracy

To be specific as below, you will find some note on Script, key take away and Area could be improved:

Part 1: import necessary lib:

```
In [1]: # -*- coding: utf-8 -*-
        """
        Created on Mon May  7 21:25:40 2018

        @author: UX390-GS048T
        """
        import pickle
        import numpy as np
        import matplotlib.pyplot as plt
        from skimage.color import rgb2gray
        import os
        os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
```

Part 2: identify Main function:

```
def Main():

    train_data_sizes = []
    train_data_coords = []
    train_data_features = []
    train_data_labels = []
    try:
        with open('F:/Udacity/Project 2/traffic-signs-data/train.p','rb') as snapFile:
            savedSnaps = pickle.load(snapFile)
    except IOError as e:
        return False
    train_data_sizes=savedSnaps['sizes']
    train_data_coords = savedSnaps['coords']
    train_data_features = savedSnaps['features']
    train_data_labels = savedSnaps['labels']
    train_images = rgb2gray(train_data_features)
```

To archive the goal of deep learning Model, below is what I have done under Main function:

2. 1: Import data of training, validation, testing set. Original data is within '.p' format, here we need to pull them and save into list format for further usage:

```

def Main():

    train_data_sizes = []
    train_data_coords = []
    train_data_features = []
    train_data_labels = []
    try:
        with open('F:/Udacity/Project 2/traffic-signs-data/train.p','rb') as snapFile:
            savedSnaps = pickle.load(snapFile)
    except IOError as e:
        return False
    train_data_sizes=savedSnaps['sizes']
    train_data_coords = savedSnaps['coords']
    train_data_features = savedSnaps['features']
    train_data_labels = savedSnaps['labels']
    train_images = rgb2gray(train_data_features)

    valid_data_sizes = []
    valid_data_coords = []
    valid_data_features = []
    valid_data_labels = []
    try:
        with open('F:/Udacity/Project 2/traffic-signs-data/valid.p','rb') as snapFile:
            savedSnaps = pickle.load(snapFile)
    except IOError as e:
        return False
    valid_data_sizes=savedSnaps['sizes']
    valid_data_coords = savedSnaps['coords']
    valid_data_features = savedSnaps['features']
    valid_data_labels = savedSnaps['labels']
    valid_images = rgb2gray(valid_data_features)

    test_data_sizes = []
    test_data_coords = []
    test_data_features = []
    test_data_labels = []
    try:
        with open('F:/Udacity/Project 2/traffic-signs-data/test.p','rb') as snapFile:
            savedSnaps = pickle.load(snapFile)
    except IOError as e:
        return False
    test_data_sizes=savedSnaps['sizes']
    test_data_coords = savedSnaps['coords']
    test_data_features = savedSnaps['features']
    test_data_labels = savedSnaps['labels']
    test_images = rgb2gray(test_data_features)




```

-> **Key take away:** while pulling original image data, it is necessary to convert colour image into grey image before using it for deep learning model. It will reduce the dimensions of input data. In the end, the benefit would be faster training time.

-> **Area could be improved:** not all the provided used in this code in the end, for those data not being used, related code could be commented to save computing Memory. More importantly, if we could enlarge original data, it would be very helpful, like project spec recommended, rotation, translation, zoom, flips, and/or colour perturbation etc. will decide later whether to do it or not depends on the time & accuracy.

-> **Result:**

Original data:

 test.p	7/5/2018 9:17 PM	P File	37,977 KB
 train.p	7/5/2018 9:17 PM	P File	104,636 KB
 valid.p	7/5/2018 9:17 PM	P File	13,261 KB

```
{'sizes': array([[53, 54],
                [42, 45],
                [48, 52],
                ...,
                [29, 29],
                [48, 49],
                [32, 31]], dtype=uint8), 'coords': array([[ 6,  5, 48, 49],
                [ 5,  5, 36, 40],
                [ 6,  6, 43, 47],
                ...,
                [ 6,  6, 24, 24],
                [ 5,  6, 43, 44],
                [ 6,  5, 27, 26]], dtype=uint8), 'features': array([[[[116, 139, 174],
                [116, 137, 171],
                [118, 138, 172],
                ...,
                [ 98, 114, 143],
                [ 97, 121, 147],
                [ 85, 105, 130]]],
```

Data after pulling into python:

```
[[ 67,  49,  44],
 [ 65,  50,  45],
 [ 65,  49,  45],
 ...,
 [ 57,  42,  39],
 [ 55,  42,  40],
 [ 56,  43,  40]],

...,

[[ 28,  24,  26],
 [ 29,  25,  27],
 [ 29,  27,  30],
 ...,
 [ 27,  24,  24],
 [ 28,  25,  24],
 [ 29,  27,  27]],
```

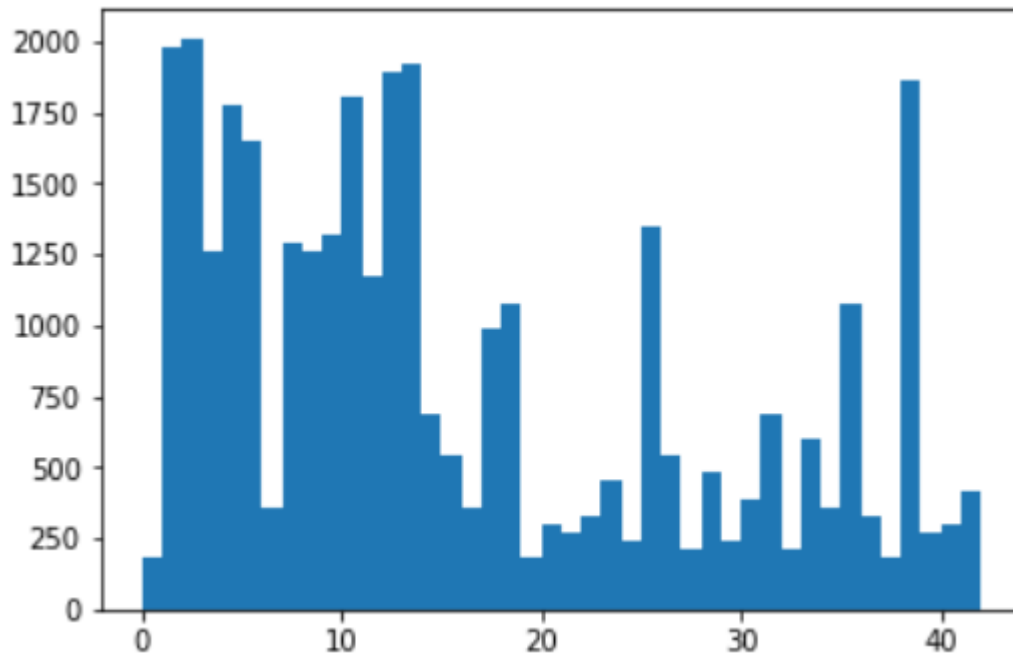
2. 2: Visualize the training data – image label

```
unique_labels = set(train_data_labels)
print(unique_labels)
plt.hist(train_data_labels, 42)
plt.show()
```

-> Result:

Summarise the training label, realized in total there are 42 different label sets, so that 43 output classifiers needed for the classification model:

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
34, 35, 36, 37, 38, 39, 40, 41, 42}
```



2. 3: choose some random images and list down shape and min/max value from image data:

```
# Import the `pyplot` module of `matplotlib`
# Determine the (random) indexes of the train_images that you want to see
traffic_signs = [260, 74, 3515, 2004]

# Fill out the subplots with the random train_images that you defined
for i in range(len(traffic_signs)):
    plt.subplot(1, 4, i+1)
    plt.axis('off')
    plt.imshow(train_images[traffic_signs[i]])
    plt.subplots_adjust(wspace=0.5)
    plt.show()
    print("shape: {0}, min: {1}, max: {2}".format(train_images[traffic_signs[i]].shape,
                                                train_images[traffic_signs[i]].min(),
                                                train_images[traffic_signs[i]].max()))
```

-> **Key take away:** as mentioned from project Spec, all images data provided have already been converted into 32*32 shape.

Still I just put a checking point here in case any other similar model has different shape of images, we can have a basic sense of how various the images' shape are.

-> **Area could be improved:** here we can also draw a chart to show how many image there are in total etc.. Just from better understanding data perspective.

-> **Result:**

4 sample Images:



shape: (32, 32), min: 0.061658823529411766, max: 0.4687956862745098



shape: (32, 32), min: 0.07532823529411764, max: 0.6334121568627451



shape: (32, 32), min: 0.04930588235294118, max: 0.1556862745098039



shape: (32, 32), min: 0.09835176470588235, max: 0.4997356862745098

2. 4: to initiate a training model using TensorFlow:

```
# Import tensorflow
import tensorflow as tf
# Initialize placeholders
x = tf.placeholder(dtype = tf.float32, shape = [None, 32, 32])
y = tf.placeholder(dtype = tf.int32, shape = [None])
# Flatten the input data
train_images_flat = tf.contrib.layers.flatten(x)
# Fully connected layer

# Dense Layer
#pool2_flat = tf.reshape(train_images_flat, [-1, 7 * 7 * 64])
dense1 = tf.layers.dense(inputs=train_images_flat, units=1024, activation=tf.nn.relu)
drop_out1 = tf.nn.dropout(dense1, 0.5)

dense2 = tf.layers.dense(inputs=drop_out1, units=128, activation=tf.nn.relu)
drop_out2 = tf.nn.dropout(dense2, 0.5)

dense3 = tf.layers.dense(inputs=drop_out2, units=64, activation=tf.nn.relu)
drop_out3 = tf.nn.dropout(dense3, 0.5)

logits = tf.contrib.layers.fully_connected(drop_out3, 43, tf.nn.relu)
# Define a loss function
loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels = y,
                                                                    logits = logits))

# Define an optimizer
train_op = tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)
# Convert logits to label indexes
correct_pred = tf.argmax(logits, 1)
# Define an accuracy metric
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

-> Key take away:

This is the most important area apparently, architect here obviously will impact the final accuracy in the end. Let's look at the model overview:

- Input layer: 32*32
- 1st layer: flatten layer
- 2nd layer: 1024 units, with 0.5 dropout
- 3rd layer: 128 unit, with 0.5 dropout

- 4th layer: 64 unit, with 0.5 dropout
- Output layer: 43

From 32*32, we use 5 layer levels to reduce level units into 43 classifiers in the end.

-> **Area could be improved:** here is the key area can be improved, as all other machine learning/ deep learning model, a bad model will not come out with any high accuracy prediction rate:

- how many levels of layer?
- how many units for each level of layer?
- dropout rate? here to be 0.5
- etc.

This part is the core of Machine learning/ deep learning. Also GPU, TPU play an important role of training out of the better model accuracy.

First time I was using laptop with 16GB CPU (no NAVIA GPU), 98% CPU usage stopped me doing that again since it may destroy the machine.

Second time I was using my desktop with NAVIA GPU 1080. It takes only 5min to complete 6000 echoes. And with not bad accuracy in the end, so the model architect mentioned above can work to a certain extend.

2. 5: Initiate model weight variables, and to set up run Epoch into 6000 times.

```
print("train_images_flat: ", train_images_flat)
print("logits: ", logits)
print("loss: ", loss)
print("predicted_labels: ", correct_pred)

tf.set_random_seed(1234)
sess = tf.Session()
sess.run(tf.global_variables_initializer())

for i in range(6001):
    print('EPOCH', i)
    _, accuracy_val = sess.run([train_op, accuracy], feed_dict={x: train_images, y: train_data_labels})
    if i % 10 == 0:
        print("Loss: ", loss)
    print('DONE WITH EPOCH')
```

2. 6: once model training completed, to show the prediction of 10 random images

```

import random

# Pick 10 random train_images
sample_indexes = random.sample(range(len(train_images)), 10)
sample_train_images = [train_images[i] for i in sample_indexes]
sample_labels = [train_data_labels[i] for i in sample_indexes]

# Run the "correct_pred" operation
predicted = sess.run([correct_pred], feed_dict={x: sample_train_images})[0]

# Print the real and predicted labels
print(sample_labels)
print(predicted)

# Display the predictions and the ground truth visually.
#fig = plt.figure(figsize=(10, 10))
for i in range(len(sample_train_images)):
    truth = sample_labels[i]
    prediction = predicted[i]
    plt.subplot(5, 2, 1+i)
    plt.axis('off')
    color='green' if truth == prediction else 'red'
    plt.text(40, 10, "Truth: {0}\nPrediction: {1}".format(truth, prediction),
            fontsize=12, color=color)
    plt.imshow(sample_train_images[i], cmap="gray")

plt.show()

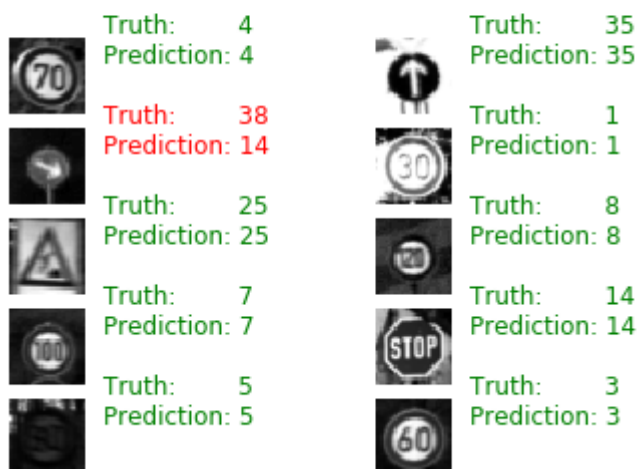
```

-> Result:

```

DONE WITH EPOCH
EPOCH 6000
Loss: Tensor("Mean_6:0", shape=(), dtype=float32)
DONE WITH EPOCH
[4, 35, 38, 1, 25, 8, 7, 14, 5, 3]
[ 4 35 14  1 25  8  7 14  5  3]

```



Part 3: calculate model prediction accuracy:


```

# Run predictions against the full test set.
predicted_train = sess.run([correct_pred], feed_dict={x: train_images})[0]
predicted_valid = sess.run([correct_pred], feed_dict={x: valid_images})[0]
predicted_test = sess.run([correct_pred], feed_dict={x: test_images})[0]

# Calculate correct matches
match_count_train = sum([int(y == y_) for y, y_ in zip(train_data_labels, predicted_train)])
# Calculate the accuracy
accuracy_train = match_count_train / len(train_data_labels)
# Print the accuracy
print("Accuracy_train: {:.3f}".format(accuracy_train))

# Calculate correct matches
match_count_valid = sum([int(y == y_) for y, y_ in zip(valid_data_labels, predicted_valid)])
# Calculate the accuracy
accuracy_valid = match_count_valid / len(valid_data_labels)
# Print the accuracy
print("Accuracy_valid: {:.3f}".format(accuracy_valid))

# Calculate correct matches
match_count_test = sum([int(y == y_) for y, y_ in zip(test_data_labels, predicted_test)])
# Calculate the accuracy
accuracy_test = match_count_test / len(test_data_labels)
# Print the accuracy
print("Accuracy_test: {:.3f}".format(accuracy_test))

```

Result:

After 6000 echoes:

```

Accuracy_train: 0.934
Accuracy_valid: 0.827
Accuracy_test: 0.812

```

After 12000 echoes:

```

Accuracy_train: 0.924
Accuracy_valid: 0.819
Accuracy_test: 0.802

```

Summary:

These are the steps I could think of for project 2, as a base line. Still a lot of improvement needed to achieve a better accuracy.

The key area could be improved is:

1. Original data expansion

if we could enlarge original data, it would be very helpful, like project spec recommended, rotation, translation, zoom, flips, and/or colour perturbation etc.

2. Model architecture enhancement

- how many levels of layer?
- how many units for each level of layer?
- dropout rate? here to be 0.5
- etc.

3. After 12000 echoes, my computer GPU temperature become 81, should couldn't try with more echoes with model. Think will think about the way to enhance model instead of running more echoes. Now training accuracy could reached 93%, but validation set still below 90% (target is 93%), will keep thinking about the way to improve and update here whenever there is a new idea!!