Machine Learning Engineer Nanodegree

Capstone Project

Traffic Sign Classifier

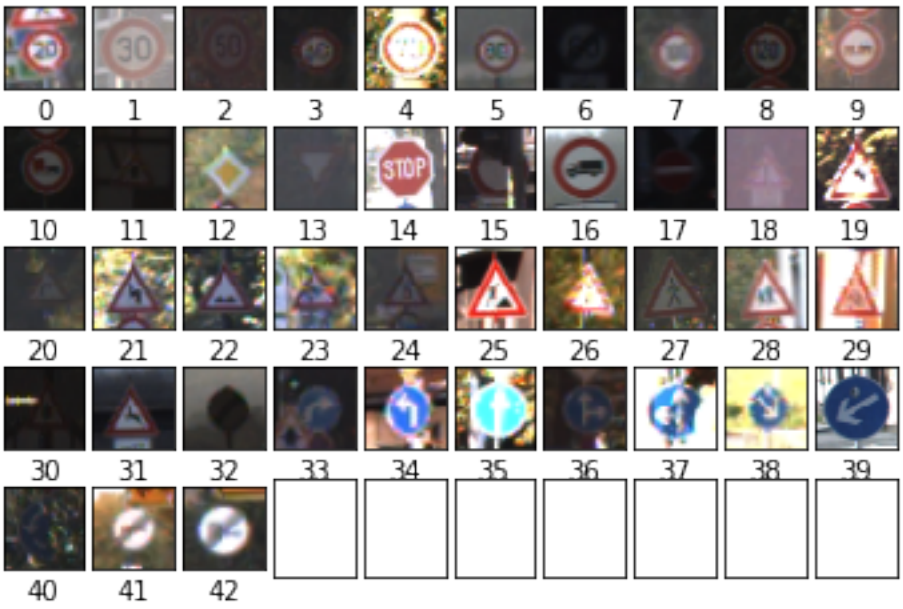
Jingxian Lin

October 8th, 2020

I. Definition

Project Overview

The task of recognizing traffic signs is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between images of traffic signs found in life.



Recalling that convolutional neural networks can be used to classify images with high accuracy and at scale, the goal of this capstone project is to apply deep learning techniques to the classification of traffic signs.

Problem Statement

The main objective of this project will be to build a pipeline to process real-world images. Given an image of a traffic sign, your algorithm will identify an estimate of the traffic sign. This is in fact a classification problem under a supervised learning domain, with the goal to reach a high accuracy.

Metrics

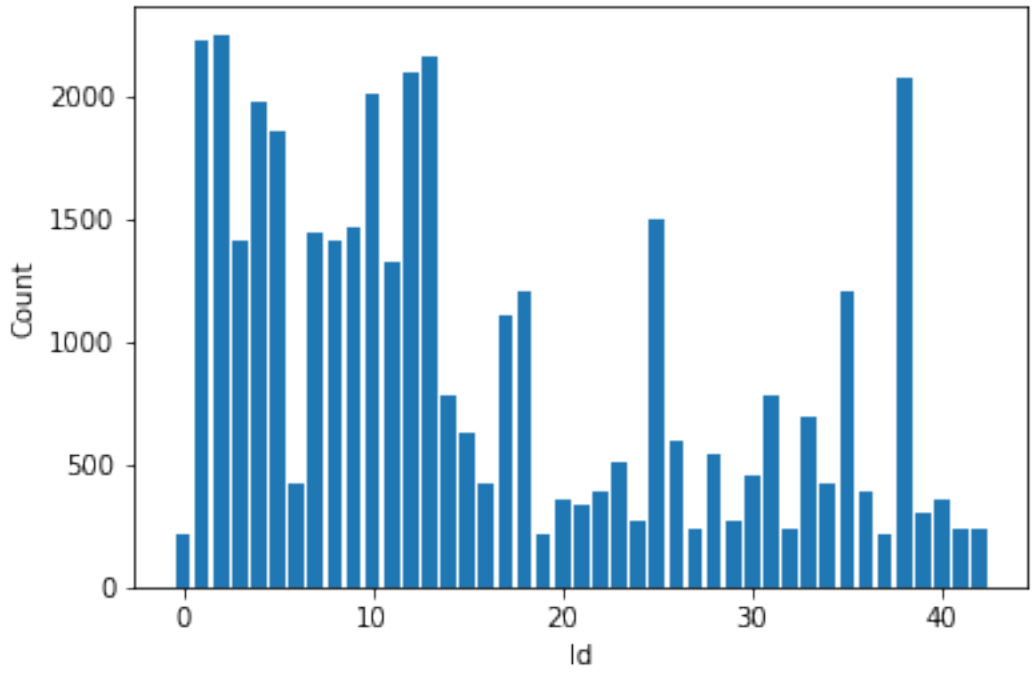
The evaluation metric that can be used to quantify the performance of both the benchmark model and the solution model is the accuracy score. Given the number of classes and the distribution of the labels, accuracy is good enough.

As a matter of justification, the count of each traffic sign is plotted in the next section.

II. Analysis

Data Exploration

German Traffic Sign Dataset is used. There are 39209 training images and 12630 testing images with 43 traffic sign categories. Label distribution (the count of each sign) is plotted below (the shown imbalance between different traffic signs influencing the metric choice and data sampling techniques, a must-have EDA step):



Exploratory Visualization

Sample images from the downloaded traffic sign dataset are presented below. Each category has a representative image shown here.

Algorithms and Techniques

Use conv_layer, flatten_layer, and fc_layer to create a CNN that can identify traffic sign from images. This way would reach higher accuracy than the following benchmark model, because the architecture is based on the extended model as a feature extractor, and a pooling layer and a fully connected layer are added.

The layers in CNNs are organized into three dimensions, width x height x depth, and the nodes in one layer do not necessarily connect to all nodes in the subsequent layer, but often just a sub region of it. This allows the CNN to perform two critical stages: One is the feature extraction—a filter window slides over the input and extracts a sum of the convolution at each location then stored in the feature map, a pooling process is included between CNN layers where typically the max value in each window is taken, decreasing the feature map size but retaining the significant data, this reduces the dimensionality of the network, the training time, and the likelihood of overfitting; the other is the classification, where the 3D data within the network is flattened into a 1D vector to be output.

Benchmark

Setting aside the fact that the classes are slightly imbalanced, a random guess will provide a correct answer roughly 1 in 43 times, which corresponds to an accuracy of less than 3%. A better benchmark model will be to create a LeNet to classify traffic signs from scratch, which should improve the accuracy.

III. Methodology

Data Preprocessing

Load the data set; Explore, summarize, and visualize the data set; Resize the images to (32, 32, 3) and rescale the images by dividing every pixel in every image by 255; Generate fake data by applying random rotation, random translation, and brightness augmentation; Cache augmented images, and split the data into train, test, and validation sets.

```
### Preprocess the data here.
def preprocessing(image):
    for i in range(3):
        image[:,:,i] = cv2.equalizeHist(image[:,:,i])

    return image / 255.

### Generate additional data.
def transforming(image, trans_range, angle_range):
    # Random rotation
    row,col,_ = image.shape
    ang_rot = np.random.uniform(angle_range)-angle_range/2
    Rot_M = cv2.getRotationMatrix2D((col/2,row/2),ang_rot,1)
    reflect = cv2.copyMakeBorder(image,10,10,10,10,cv2.BORDER_REFLECT)
    image = cv2.warpAffine(reflect,Rot_M,(col+20,row+20))[10:col+10,10:row
+10,:]

    # Random translation
    trans_x = trans_range*np.random.uniform()-trans_range/2
    trans_y = trans_range*np.random.uniform()-trans_range/2
    Trans_M = np.float32([[1,0,trans_x],[0,1,trans_y]])
    reflect = cv2.copyMakeBorder(image,10,10,10,10,cv2.BORDER_REFLECT)
    image = cv2.warpAffine(reflect,Trans_M,(col+20,row+20))[10:col+10,10:r
ow+10,:]

    # Brightness augmentation
    dst = cv2.cvtColor(image,cv2.COLOR_RGB2HSV)
    dst[:,:,2] = dst[:,:,2]*(0.1+np.random.uniform())
    image = cv2.cvtColor(dst,cv2.COLOR_HSV2RGB)

    return preprocessing(image)

X_trans = np.array([transforming(image, 3, 20) for image in X_train], dtyp
e=np.float32)
```

Implementation

For architecture constructing, define the CNN structure, specify loss function and optimizer. Three sets of convolutional and max pooling layers are used to extract complex features to distinguish different

traffic signs. A flatten layer and a fully connected layer are added, then an output layer with SoftMax activation.

```
### Define architecture here.
def conv_layer(input, num_in_channels, filter_size, num_filters, use_pooli
ng=True):
    shape = [filter_size, filter_size, num_in_channels, num_filters]
    weights = init_weights(shape)
    biases = init_biases(num_filters)
    layer = tf.add(tf.nn.conv2d(input=input, filter = weights, strides=[1,
1,1,1], padding='SAME'), biases)

    if use_pooling:
        layer = tf.nn.max_pool(value=layer, ksize=[1,2,2,1], strides=[1,2,
2,1], padding='SAME')
    return tf.nn.relu(layer), weights

def flatten_layer(layer):
    layer_shape = layer.get_shape()
    num_features = layer_shape[1:4].num_elements()
    return tf.reshape(layer, [-1,num_features]), num_features

def fc_layer(input, num_inputs, num_outputs, use_relu=True):
    weights = init_weights([num_inputs, num_outputs])
    biases = init_biases(num_outputs)
    layer = tf.add(tf.matmul(input, weights), biases)
    return tf.nn.relu(layer) if use_relu else layer, weights

def init_weights(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.05))

def init_biases(length):
    return tf.Variable(tf.constant(0.05, shape=[length]))
```

Refinement

Creating a CNN to predict traffic sign from scratch achieves a test accuracy score: ~97%; to train the model, Adam Optimizer was used; several batch size numbers were tried, and an acceptable compromise between speed and accuracy was found for a batch size of 512. Better result could be possible with more epochs and further hyperparameter tuning.

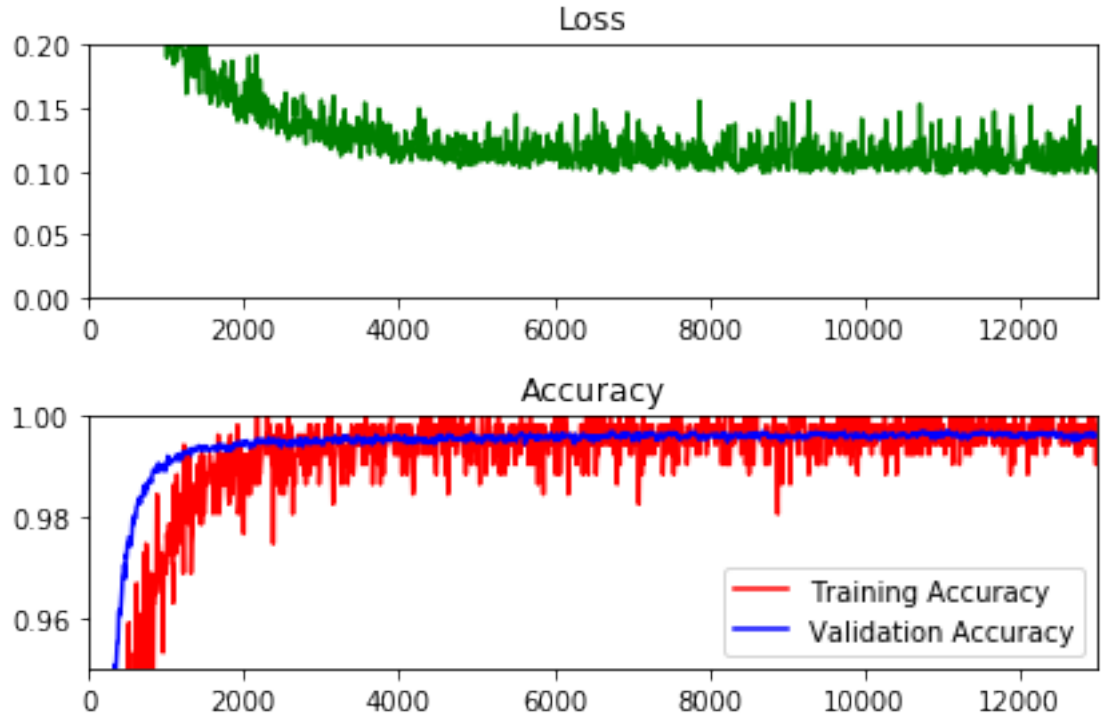
```
import math

session = tf.Session()
session.run(init)
```

```
batch_count = int(math.ceil(len(X_train)/batch_size))

for epoch in range(epoches):
    batches_pbar = tqdm(range(batch_count), desc='Epoch {:>2}/{}'.format(e
poch+1, epoches), unit='batches')

    for batch_i in batches_pbar:
        batch_start = batch_i*batch_size
        batch_features = X_train[batch_start:batch_start+batch_size]
        batch_labels = y_train[batch_start:batch_start+batch_size]

        feed_dict_batch = {x_image: batch_features, y_true: batch_labels,
keep_prob: 0.5}
        _, l = session.run([optimizer, loss], feed_dict = feed_dict_batch)

        if not batch_i % log_batch_step:
            training_accuracy = session.run(accuracy, feed_dict = feed_dic
t_batch)
            validation_accuracy = session.run(accuracy, feed_dict = valid_
feed_dict)
            previous_batch = batches[-1] if batches else 0
            batches.append(log_batch_step + previous_batch)
            loss_batch.append(l)
            train_acc_batch.append(training_accuracy)
            valid_acc_batch.append(validation_accuracy)

    validation_accuracy = session.run(accuracy, feed_dict = valid_feed_dic
t)
    print('Validation accuracy at {}'.format(validation_accuracy))
    save_path = saver.save(session, "./models/traffic_sign.ckpt")
    print("Model saved in file: %s" % save_path)
    test_accuracy = session.run(accuracy, feed_dict = test_feed_dict)
    print(epoch, 'Test accuracy at {}'.format(test_accuracy))
```

IV. Results

Model Evaluation and Validation

Monitor the accuracy and loss during the training process of the chosen CNN model: They change rapidly at first, then steadily, and finally a test accuracy is at 0.9707.



Justification

The final model achieves a classification accuracy of 97% on the test set, much higher than the benchmark. Use the model to make predictions on new images below taken from the web, generated by the Google map. Traffic pictures captured through a moving vehicle leading to motion blur and perspective distortions make classification more challenging. Especially, the 'Pedestrians' sign doesn't show up in the training set in this form.

The model predicted 4 out of 5 signs correctly, so it's 80% accurate. Compared to testing on the dataset, this classifier is not able to perform equally well on captured pictures. The 'Pedestrians' sign

didn't get recognized, but for others that appear in the training set, it gives out correct predictions.

V. Conclusion

Free-Form Visualization

Use the model's softmax probabilities to visualize the certainty of its predictions. For correct predictions, the model is at least more than 50% certain; for the traffic sign not appearing in the training set, it tends to predict as more general class. According to the count of each kind, balancing data maybe improve the result.

Reflection

Several steps are taken in this project:

- Import Datasets
- Exploratory Data Analysis
- Data Preprocessing
- Create a CNN to Classify Traffic Signs from Scratch
- Test a Model on New Images

Improvement

Finally, provide 3 possible points of improvement. The output is better than I expected. First, there is imbalance between different traffic signs, more training data will help; Second, try more transfer learning with VGG19, InceptionV3 or Xception; Third, model fusion can further improve.