Traffic Sign Classifier

Eeshan Deosthale 05/08/2020

Introduction

The traffic-sign classifier project deals with classifying German traffic signs from their images using Deep Learning techniques. Building a deep Convolutional Neural Network model is essential to classify traffic signs for this project. Important steps in completing this projects are as follows.

1. Load the German traffic-sign data-set and identify training, validation and test sets
2. Explore, summarize and visualize the data-set
3. Pre-process the data-set (augment, normalize, grayscale etc.)
4. Design a deep learning model using tensorflow
5. Train and test the model architecture using training and validation data-set
6. Use the model to make predictions on test set as well as new images
7. Analyze softmax probabilities for new images
8. Summarize the results

As a first step to develop code for this project, it is important to import necessary packages in python.

Listing 1: Import Packages

```
import pickle
import numpy as np
import cv2
from sklearn.utils import shuffle
import tensorflow as tf
import random
import matplotlib.pyplot as plt
# Visualizations will be shown in the notebook.
%matplotlib inline
```

Dataset Exploration

The traffic signs images used in this project are from the German traffic sign dataset. The dataset is divided into 3 sets viz. training, validation and test sets. A quick summary of the dataset is given in Table 1. The training set has more than 30000 examples available and each image is color image with RGB channels.

Along with the overall summary, it is also important to look at the number of training examples available per class. A bar plot as shown in Fig. 1 is used to visualize the training examples for different classes. There is lot of variation between number of training images available per traffic sign. Some signs have upwards of 1500 images available while some have less than 500. The preprocessing of this dataset is described in the next section.

Number of training examples	34799
Number of validation examples	4410
Number of testing examples	12630
Image data shape	$(32, 32, 3)$
Number of classes	43

Table 1: Summary of the German Traffic Sign Dataset

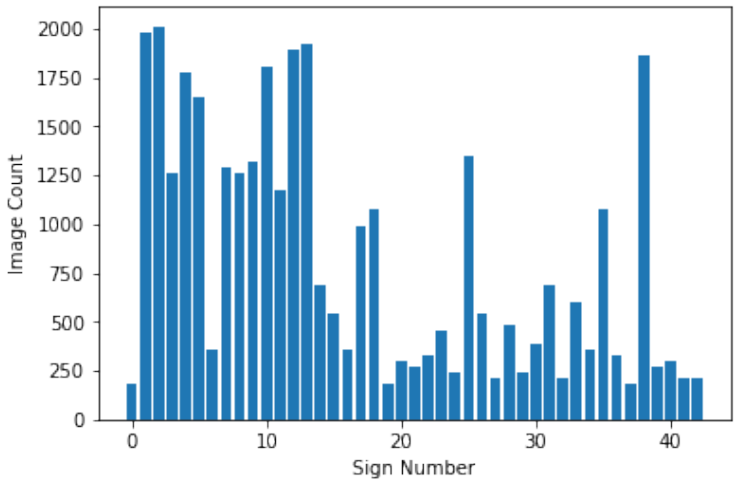

Figure 1: Number of Training Images Available Per Class

Preprocessing Training Data

The data preprocessing is done in 4 steps.

1. Level the number of training examples available per traffic sign

2. Grayscale the images to reduce datapoints
3. Augment the data set by rotating and translating images
4. Normalize the dataset which is suitable for training neural networks

Leveling the Dataset

As described in previous section, the dataset has high variability in terms of number of training examples available per class. To level the dataset, a threshold of 1500 images was selected and for the classes with examples less than that, images were added by randomly duplicating training images. Though this method works well for this dataset, an online article mentioned that it may not be a good idea as the variability in the original training dataset can be representative of the occurance of that particular class in real world. The dataset looks as shown in Fig. 2.

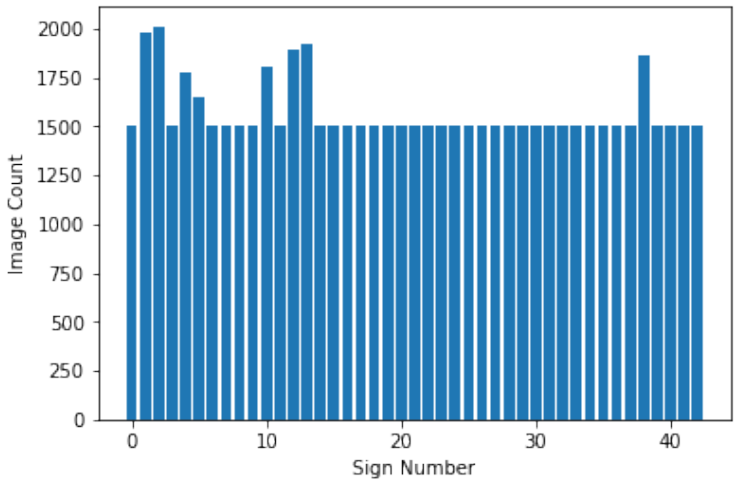

Figure 2: Number of Training Images Available Per Class

Listing 2: Dataset Leveling

```
## Bring the number of images per traffic sign above a certain threshold
threshold = 1500
for i in range(n_classes):
    if counts_per_label[i]<threshold:
        img_arr = X_train[y_train == i]
        diff_count = threshold - counts_per_label[i]
        rand_idx = (np.random.rand(diff_count)*(len(img_arr)-1)//1).astype(↩
           int)
        append_arr = img_arr[rand_idx]
        append_label = np.ones(diff_count)*i
```

```
        X_train = np.concatenate((X_train, append_arr))
        y_train = np.concatenate((y_train, append_label))
unique_labels, counts_per_label = np.unique(y_train, return_counts = True)
fig, ax = plt.subplots()
ax.bar(range(len(unique_labels)), counts_per_label)
ax.set(xlabel = 'Sign Number', ylabel = 'Image Count')
print('Training X Length', len(X_train), 'Training y Length', len(y_train))
```

Grayscaling Images

Once the dataset is levelled, the images are grayscaled to reduce feature maps and thereby reduce datapoints. Python OpenCV library is used to convert RGB images to grayscale images. Fig. 3 shows a 'Right Turn Ahead' sign in color and grayscale.

Listing 3: Grayscaling Dataset

```
X_train_gray = np.zeros((len(X_train), 32, 32, 1))
for i in range(len(X_train)):
    X_train_gray[i,:,:,0] = cv2.cvtColor(X_train[i], cv2.COLOR_RGB2GRAY)

X_valid_gray = np.zeros((len(X_valid), 32, 32, 1))
for i in range(len(X_valid)):
    X_valid_gray[i,:,:,0] = cv2.cvtColor(X_valid[i], cv2.COLOR_RGB2GRAY)

X_test_gray = np.zeros((len(X_test), 32, 32, 1))
for i in range(len(X_test)):
    X_test_gray[i,:,:,0] = cv2.cvtColor(X_test[i], cv2.COLOR_RGB2GRAY)
```

(a) Color Image (b) Grayscale Image

Figure 3: Grayscaling of Traffic Sign Images

Dataset Augmentation

The next step in pre-processing the data is to augment the dataset by rotating and translating existing images. Such kind of augmentation prevents the neural network model from over-fitting the data. For each image, the angle of rotation is randomly chosen between $-10°$ and $10°$. Also, the images are translated by 3 pixels in x and y direction, but the sign is chose in a sequential manner. The dataset is augmented by 3 times its original number by rotating, translating and rotating as well translating every image. Fig. 4 shows the 'Right Turn Ahead' sign after rotation and translation.

(a) Rotated Sign (b) Translated Sign

Figure 4: Rotation and Translation of Traffic Sign Images

Listing 4: Image Rotation

```
rot_arr = (np.random.rand(len(X_train_gray))*20-10)//1
new_img_arr, new_y_arr = [], []
X_train_gray_copy = np.copy(X_train_gray)
y_train_copy = np.copy(y_train)
for i, img in enumerate(X_train_gray_copy):
    M = cv2.getRotationMatrix2D((16,16),rot_arr[i],1)
    dst = cv2.warpAffine(img,M,(32,32))
    dst = np.expand_dims(dst, axis=2)
    new_img_arr += [dst]
    new_y_arr += [y_train_copy[i]]
new_img_arr = np.array(new_img_arr)
new_y_arr = np.array(new_y_arr)
X_train_gray = np.append(X_train_gray, new_img_arr, axis=0)
y_train = np.append(y_train, new_y_arr, axis=0)
print(len(X_train_gray), len(y_train))
```

Listing 5: Image Translation

```
trans_x = [3, 3, -3, -3]
trans_y = [3, -3, 3, -3]
# X_train_gray_copy = np.copy(X_train_gray)
new_trans_X, new_trans_y = [], []
for i, img in enumerate(X_train_gray_copy):
    M_t = np.float32([[1,0,i%4],[0,1,i%4]])
    dst = cv2.warpAffine(img,M_t,(32,32))
    dst = np.expand_dims(dst, axis=2)
    new_trans_X += [dst]
    new_trans_y += [y_train_copy[i]]
new_trans_X = np.array(new_trans_X)
new_trans_y = np.array(new_trans_y)
X_train_gray = np.append(X_train_gray, new_trans_X, axis=0)
y_train = np.append(y_train, new_trans_y, axis=0)
print(len(X_train_gray), len(y_train))
```

Listing 6: Image Rotation plus Translation

```
# X_train_gray_copy = np.copy(X_train_gray)
# y_train_copy = np.copy(y_train)
trans_x = [3, 3, -3, -3]
trans_y = [3, -3, 3, -3]
new_X, new_y = [], []
rot_arr = (np.random.rand(len(X_train_gray_copy))*20-10)//1
for i, img in enumerate(X_train_gray_copy):
    M = cv2.getRotationMatrix2D((16,16),rot_arr[i],1)
    M_t = np.float32([[1,0,i%4],[0,1,i%4]])
    dst = cv2.warpAffine(img,M,(32,32))
    dst = cv2.warpAffine(img,M_t,(32,32))
    dst = np.expand_dims(dst, axis=2)
    new_X += [dst]
    new_y += [y_train_copy[i]]
new_X = np.array(new_X)
new_y = np.array(new_y)
X_train_gray = np.append(X_train_gray, new_X, axis=0)
y_train = np.append(y_train, new_y, axis=0)
print(len(X_train_gray), len(y_train))
```

Normalizing Images

Finally, as a last step after data augmentation, the images are normalized to avoid extreme values during model training. Normalizing images also sets the values around zero which is useful in training neural networks. After normalization, all the pixel values lie between -1 and 1. The normalized image of 'Right Turn Ahead' sign is shown in Fig. 5. The dataset is shuffled after normalization.

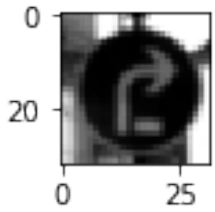

Figure 5: Normalized Grayscale Image of a Traffic Sign

Listing 7: Normalizing and Shuffling

```
def normalize_data(X):
    return (X - 128)/128

def shuffle_data(X, y):
    X, y = shuffle(X, y)
    return X, y
```

Neural Network Architecture

After exploring different model architectures (with less and more layers), the final model architecture is described in Table 2. The final model architecture is based on the famous LeNet architecture for image classification. The LeNet architecture is a well established and verified architecture for MNIST dataset classification. This architecture is chosen as it did a great of classifying images into 10 categories of numbers. The depth of each layer along with activation function is experimented with to arrive at final model.

The architecture consists of two convolutional layers followed by three fully connected layers. 'Relu' activation is used for all the layers except the last fully connected layer. Softmax activation is used to convert the output of last layer to probabilities. Max pooling is used after each convolutional layers. Dropout is used for layers 2, 3 and 4 to avoid any potential overfitting of data. The final model architecture is arrived at after experimenting with different hyperparameters and depth of different

layers. The final set of hyperparameters is represented in Table 3.

Layer	Description
Input	$32 \times 32 \times 1$ Grayscale Image
Convolution 5×5	1×1 stride, 'Valid' padding, $28 \times 28 \times 10$ output
RELU	
Max Pooling	2×2 stride, 2×2 kernel, $14 \times 14 \times 10$ output
Convolution 5×5	1×1 stride, Valid padding, $10 \times 10 \times 20$ output
RELU	
Max Pooling	2×2 stride, 2×2 kernel, $5 \times 5 \times 20$ output
Dropout	keep_prob $= 0.6$
Flatten	1×500 output
Fully Connected Layer	1×240 output
RELU	
Dropout	keep_prob $= 0.6$
Fully Connected Layer	1×168 output
RELU	
Dropout	keep_prob $= 0.6$
Fully Connected Layer	1×43 output
Softmax	

Table 2: Model Architecture for Traffic Sign Classification

Listing 8: Model Definition

```
def conv2d(x, W, b, stride = 1, pad = 'VALID'):
    x = tf.nn.conv2d(x, W, strides = [1,stride,stride,1], padding = pad)
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)

def maxpool2d(x, k = 2, stride = 2):
    return tf.nn.max_pool(x, ksize = [1,k,k,1], strides = [1, stride, stride
        , 1], padding = 'SAME')

from tensorflow.contrib.layers import flatten

def LeNet(x, in_channels = 3, drop_out = 1.):
    ## Arguments used for tf.truncated_normal, randomly defines variables
       for the weights and biases for each layer
```

Hyperparameter	Value
Batch size	64
Dropout (keep_prob)	0.6
Epochs	15
Learning rate	0.001
Optimizer	Adam
Loss function	Cross entropy

Table 3: Neural Network Model Parameters

```
mu = 0
sigma = 0.1

## Define no. of channels in each layer
channels = [10, 20, 500, 240, 168, 43]

## Define nomralized weights and biases
weights = {
    'wc1': tf.Variable(tf.truncated_normal([5,5,in_channels,channels↩
       [0]], mean = mu, stddev = sigma)),
    'wc2': tf.Variable(tf.truncated_normal([5,5,channels[0],channels↩
       [1]], mean = mu, stddev = sigma)),
    'wd1': tf.Variable(tf.truncated_normal([channels[2],channels[3]], ↩
       mean = mu, stddev = sigma)),
    'wd2': tf.Variable(tf.truncated_normal([channels[3],channels[4]], ↩
       mean = mu, stddev = sigma)),
    'out': tf.Variable(tf.truncated_normal([channels[4],channels[5]], ↩
       mean = mu, stddev = sigma))
}

biases = {
    'bc1': tf.Variable(tf.truncated_normal([channels[0]])),
    'bc2': tf.Variable(tf.truncated_normal([channels[1]])),
    'bd1': tf.Variable(tf.truncated_normal([channels[3]])),
    'bd2': tf.Variable(tf.truncated_normal([channels[4]])),
    'out': tf.Variable(tf.truncated_normal([channels[5]]))
}

## Layer 1: Convolutional with Pooling. Input = 32x32x1, Output = 14↩
   x14x6.
conv1 = conv2d(x, weights['wc1'], biases['bc1'], 1, 'VALID')
```

```
    conv1 = maxpool2d(conv1, 2, 2)
#     conv1 = tf.nn.dropout(conv1, 0.9)

    ## Layer 2: Convolutional with Pooling. Input = 14x14x6, Output = 5x5x16↩
        .
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'], 1, 'VALID')
    conv2 = maxpool2d(conv2, 2, 2)
    conv2 = tf.nn.dropout(conv2, drop_out)

    ## Layer 3: Fully Connected. Input = 400. Output = 120.
    fc1 = flatten(conv2)
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    fc1 = tf.nn.dropout(fc1, drop_out)

    ## Layer 4: Fully Connected. Input = 120, Output = 84.
    fc2 = tf.add(tf.matmul(fc1, weights['wd2']), biases['bd2'])
    fc2 = tf.nn.relu(fc2)
    fc2 = tf.nn.dropout(fc2, drop_out)

    ## Layer 5: Fully Connected. Input = 84, Output = 43.
    logits = tf.add(tf.matmul(fc2, weights['out']), biases['out'])

    return logits, conv1, conv2
```

Training CNN Model

The model is trained on training data and the model hyperparameters are tuned base on the accuracy obtained on training and validation dataset. A high accuracy on training dataset and a relatively lower accuracy on validation dataset indicates the model overfitting the data. After a few such tries, a dropout is introduced in the model to prevent overfitting. Changes were then made to the hyperperparameters such feature map size for each layer, no. of layers etc. The performance of the final model is indicated in Table 4. Since the accuracy performance gives the idea about the performance of the model over the entire dataset, precision and recall for each label is also checked to see if the model is classifying individual signs correctly. The Fig. 6 shows that the precision and recall is high for almost all the signs. The performance of the model over certain signs can improved by either improving the model structure and/or augmenting the dataset better.

Dataset	Accuracy
Training	99.3
Validation	96.0
Test	94.0

Table 4: Neural Network Model Performance

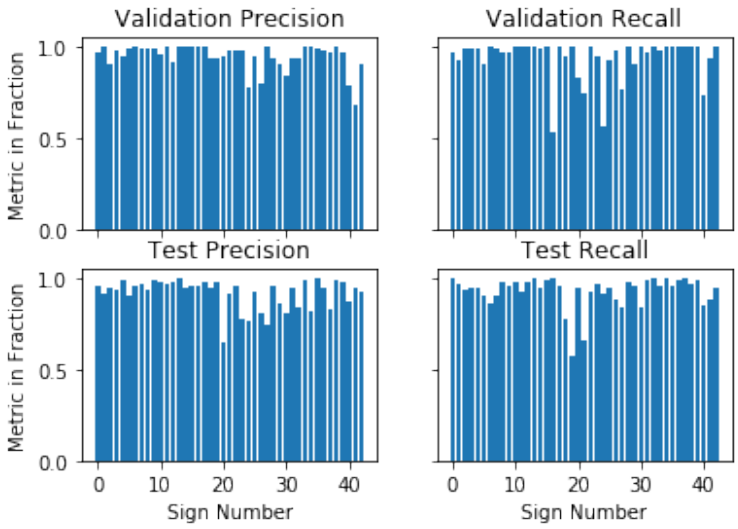

Figure 6: Precision and Recall for Each Sign

Testing CNN Model on Web Images

Listing 9: Prediction on Web Images

```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    test_data_accuracy = evaluate(test_images_gray, test_expect)
    predict = sess.run(prediction, feed_dict={x: test_images_gray, keep_prob↩
        : 1., channel_in: no_channels})
print(test_data_accuracy)
print(test_expect, predict)
```

After getting satisfactory performance on the training and validation dataset, next step is to test the model on totally new images obtained from web. The 10 images are displayed in Fig. 7. The image sequence is from left to right and then top to bottom in the displayed images. The images are preprocessed in the exact same way as training images i.e. they were grayscaled and normalized. Trained model is then loaded and the performance is tested on new images.

The predictions obtained on these images are indicated in Table 5. 10 out of 10 images are classified correctly by the model, which yields 100% accuracy. The percentage here may not be a great way to evaluate the model as only 10 images are tested, but since the model has already been tested successfully on a test dataset which was not used in training in any way, we can be confident about the model's performance.

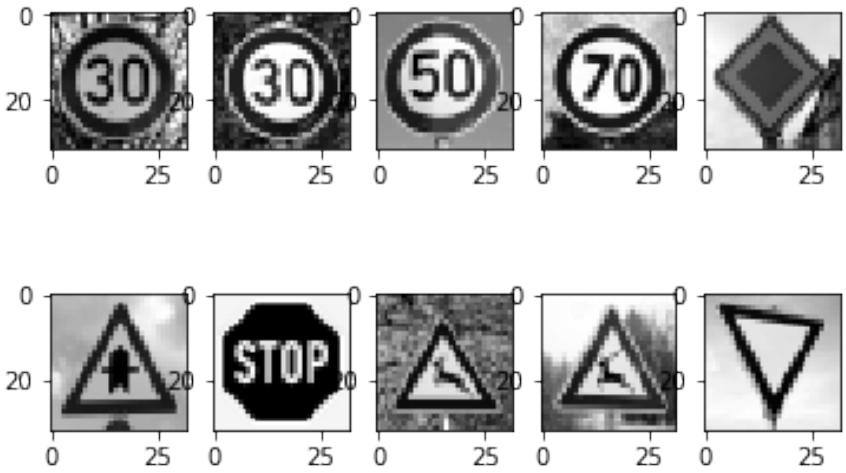

Figure 7: Web Images for Model Testing

Image	Prediction
Speed limit (30km/h)	Speed limit (30km/h)
Speed limit (30km/h)	Speed limit (30km/h)
Speed limit (50km/h)	Speed limit (50km/h)
Speed limit (70km/h)	Speed limit (70km/h)
Priority road	Priority road
Right of way at next intersection	Right of way at next intersection
Stop	Stop
Wild animals crossing	Wild animals crossing
Wild animals crossing	Wild animals crossing
Yield	Yield

Table 5: Neural Network Prediction on Web Images

Analyzing Softmax Probabilities for Web Images

After looking the classification performance on web images, top 5 softmax probabilities are visualized for each image as shown in Fig. 8. The barplot shows that the probability obtained for the

classified sign is very high compared to any other sign. This means that the model is able to make a solid conclusion on the class of image after being presented with one. Probabilities close to each other would have indicated that the model is not able to tell with confidence that the provided image is of the classified class. Table 6 shows the softmax predictions for the ninth image of 'Wild animals crossing' sign. The top 5 probabilities belong to mainly warning signs which makes an intuitive sense as most of them are triangular in shape with white background and dark indicator which easily distinguishes them from other type of signs.

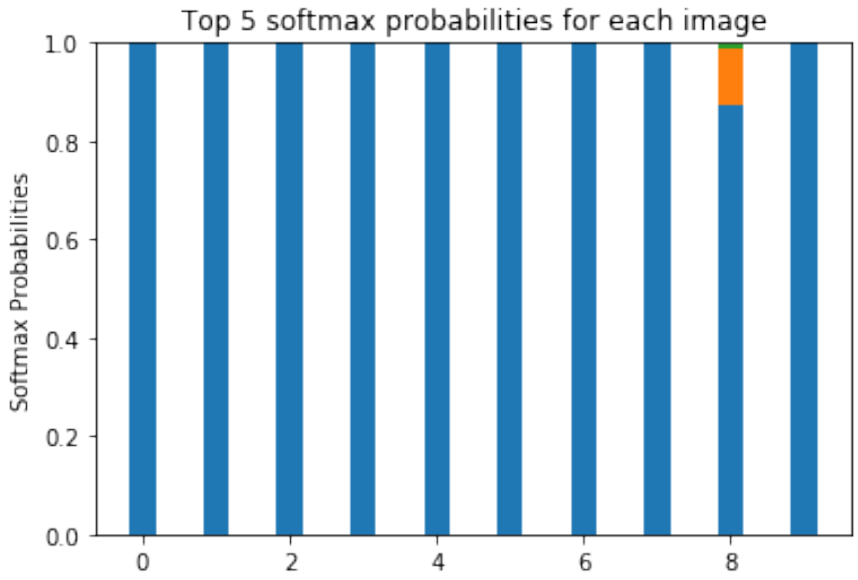

Figure 8: Softmax Probabilities for Web Images

Probability	Prediction
0.874	Wild animals crossing
0.111	Slippery road
0.009	Road Work
0.000	Dangerous curve to the left
0.000	No passing for vehicles over 3.5 metric tons

Table 6: Neural Network Model Performance

Summary and Visualization

In conclusion of this project, it can be said that the Deep Learning neural network model based on the LeNet architecture did a good job of classifying the German traffic signs with 96% accuracy on validation dataset and 94% accuracy on test dataset. The model did a decent job on web images as

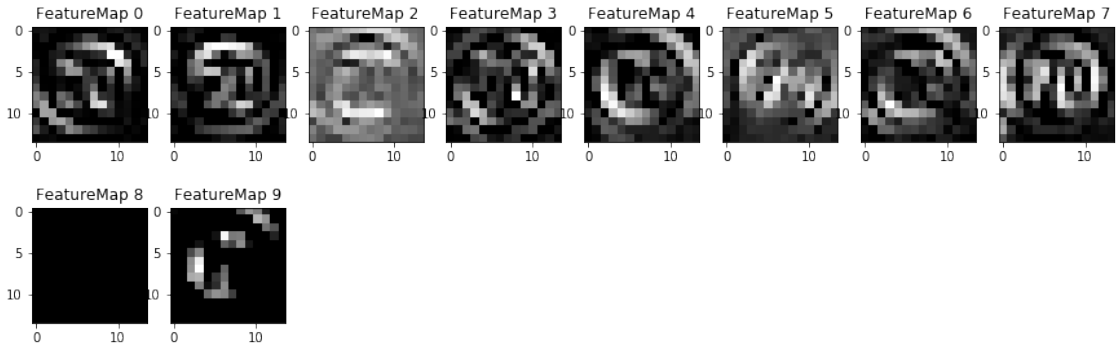

Figure 9: Feature Map Visualization for 70 km/h sign

well. But, at times the model seems to be getting confused between certain speed limit signs. Some modification will be necessary at architecture and/or data augmentation level to further increase the performance.

Finally, a visualization of activations from first convolutional layer is presented to see the features that are being picked up by the network. The visualization is shown in Fig. 9.