

Traffic Sign Classifier

Write up

1st submit: August 10th, **2nd submit: August 11th** Kenta Kumazaki

1. Purpose

The goals / steps of this project are the following:

- Load the data set
- Explore, summarize and visualize the data set
- Design, train and test a model architecture
- Use the model to make predictions on new images
- Analyze the softmax probabilities of the new images
- Summarize the results with a written report

2. Output

- Make a pipeline that run deep learning to classify traffic signs.
I used the Workspace in Udacity for this project.
- Reflect on my work in a written report.

3. Submission

(1) GitHub

https://github.com/kkumazaki/Self-Drivig-Car_Project3_Traffic-Sign-Classifer-Project.git

(2) Directory

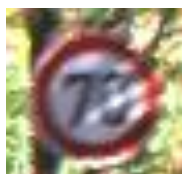
<folder: main>

- **Writeup_of_Lesson14.pdf**: This file
- **Traffic_Sign_Classifier.ipynb**: Pipeline file (Jupyter Notebook)
- **Traffic_Sign_Classifier.html**: Pipeline file (HTML)
- Image files are saved as following:

<folder: Test_pics>

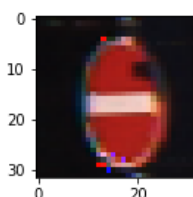
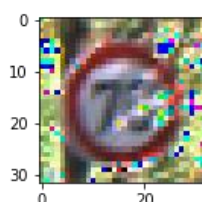
Additional Test images downloaded by the following site.

<https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign/version/1>



<folder: Test_results>

Modified Test images for Traffic Sign Classification.



4. Reflection

(1)Description of my pipeline and results

My pipeline consisted of 4 steps as following, including the preparation Step: 0.

Step 0: Load the Data

Step 1: Dataset Summary & Exploration

Step 2: Design and Test a Model Architecture

Step 3: Test a Model on New Images

Step 0: Load the Data

Load the pre-uploaded datasets at the workspace.

```
# Load pickled data
import pickle

# TODO: Fill this in based on where you saved the training and testing data

training_file = "../data/train.p"
validation_file = "../data/valid.p"
testing_file = "../data/test.p"

with open(training_file, mode='rb') as f:
    train = pickle.load(f)
with open(validation_file, mode='rb') as f:
    valid = pickle.load(f)
with open(testing_file, mode='rb') as f:
    test = pickle.load(f)

#X_train, y_train = train['features'], train['labels']
#X_valid, y_valid = valid['features'], valid['labels']
#X_test, y_test = test['features'], test['labels']
X_train, y_train = train['features'], train['labels']
X_valid, y_valid = valid['features'], valid['labels']
X_test, y_test = test['features'], test['labels']
```

Step 1: Dataset Summary & Exploration

At first, I check the dimensions of each necessary data.

```
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the results
```

```
# Check whether the length of X, y are same.
assert(len(X_train) == len(y_train))
assert(len(X_valid) == len(y_valid))
assert(len(X_test) == len(y_test))

# TODO: Number of training examples
n_train = len(X_train)

# TODO: Number of validation examples
n_validation = len(X_valid)

# TODO: Number of testing examples.
n_test = len(X_test)

# TODO: What's the shape of an traffic sign image?
image_shape = X_train[0].shape
```

```
# TODO: How many unique classes/labels there are in the dataset.
```

```
import csv
with open('signnames.csv') as f:
    next(f)
    reader = csv.reader(f, delimiter=',')
    sign_names = [row for row in reader]

n_classes = len(sign_names)

print("Basic information:")
print("Number of training examples =", n_train)
print("Number of testing examples =", n_test)
print("Image data shape =", image_shape)
print("Number of classes =", n_classes)
```

```
Basic information:
Number of training examples = 34799
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43
```

Next, I visualize the datasets with bar graphs.

```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
# Visualizations will be shown in the notebook.
%matplotlib inline
```

```
import random
import numpy as np
```

```
#Add here
import glob
import re
import cv2
import tensorflow as tf
from tensorflow.contrib.layers.python.layers import flatten
from sklearn.utils import shuffle
```

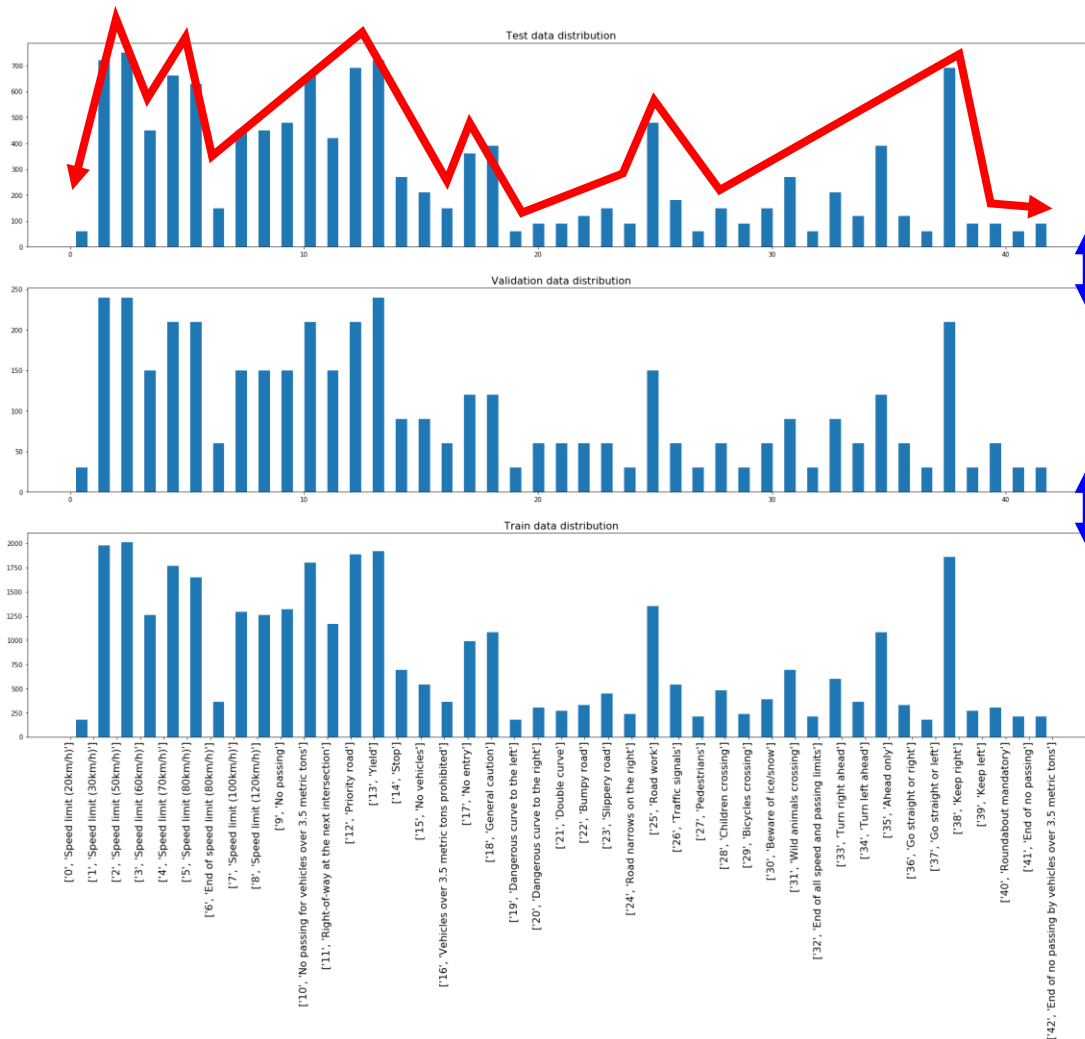
```
index = random.randint(0, len(X_train))
image = X_train[index].squeeze()
```

```
#print("Visualize sample picture:")
#plt.figure(figsize=(1, 1))
#plt.imshow(image)
#plot_index = y_train[index]
#print(plot_index)
#print(sign_names[plot_index])
```

```
#(1)Show dataset distribution
f, axes = plt.subplots(nrows=3, ncols=1, figsize=(30, 20))
axes[2].hist(y_train, bins=n_classes, rwidth=0.5)
axes[2].set_title("Train data distribution", fontsize=16)
plt.xticks(np.arange(len(sign_names)), sign_names, rotation=90, fontsize=16)
axes[1].hist(y_valid, bins=n_classes, rwidth=0.5)
axes[1].set_title("Validation data distribution", fontsize=16)
axes[0].hist(y_test, bins=n_classes, rwidth=0.5)
axes[0].set_title("Test data distribution", fontsize=16)
```

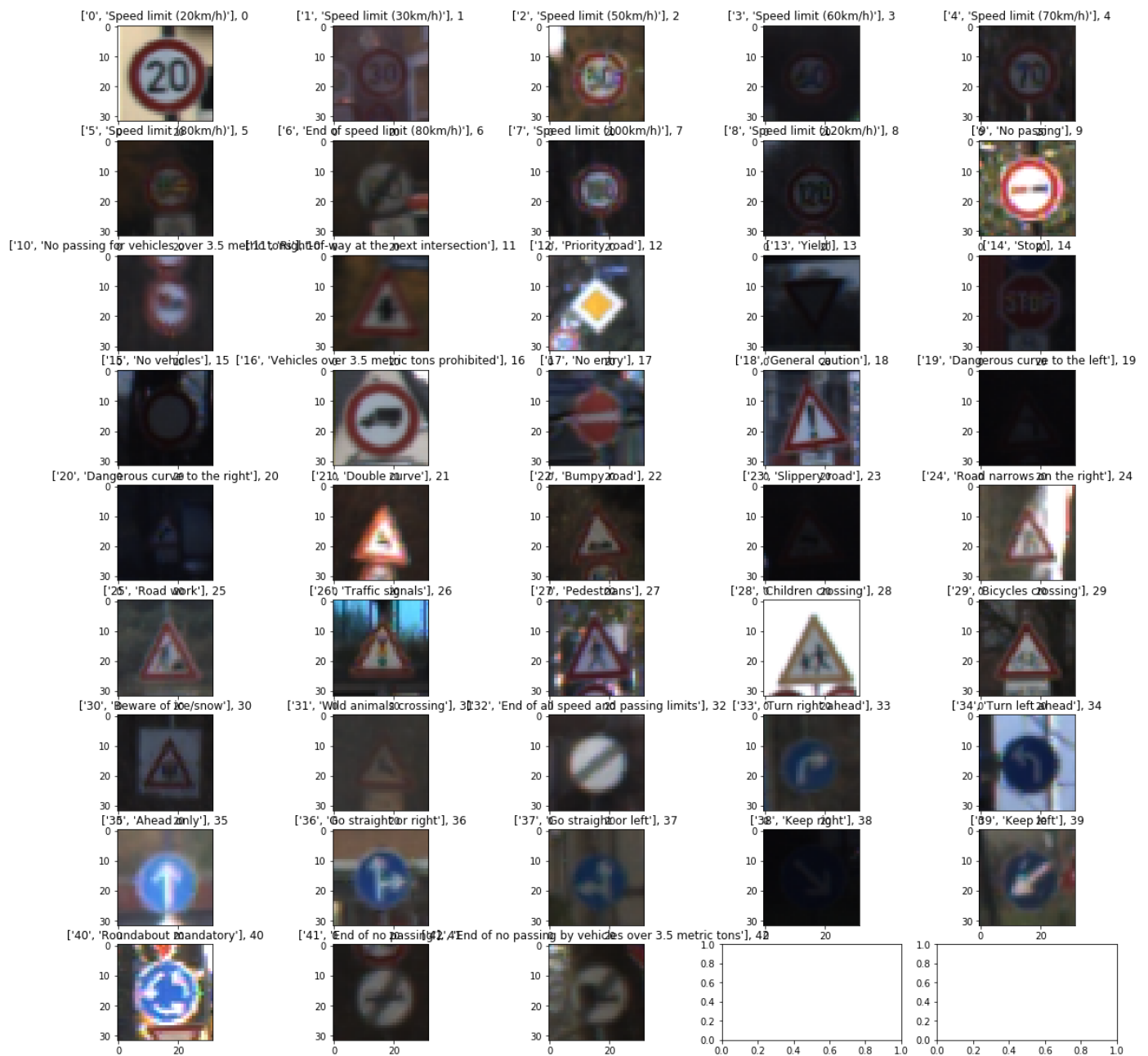
The distributions of Test Data, Validation Data and Tran Data are shown below.

There's **big disperse between each class (0 ~ 42)**, but there's **small disperse between each datasets**, so it will be OK to proceed.



```
#(2) Show example image for each category
fig, axes = plt.subplots(nrows=9, ncols=n_classes//9+1, figsize=(20,20))
for i in range(n_classes):
    ind = np.min(np.where(y_train==i))
    image = X_train[ind].squeeze()
    axes[i//5][i%5].imshow(image)
    axes[i//5][i%5].set_title("%s, %s" % (sign_names[y_train[ind]], y_train[ind]))
fig.savefig("examples.png")
```

The following images are examples of Traffic Signs.



Step 2: Design and Test a Model Architecture

First in the Step2, I pre-process the Data Set.

I proceeded with the following steps:

[Check #1] I trained the model without any preprocess nor any change from the original model,
and the Validation Accuracy was just 89.2%. Target Accuracy is 93%, so it's not good enough.

[Check #2] I made the images from color to gray scale, and normalized them.

Validation Accuracy became 92.5%, but it's not good enough.

[Check #3] I added training data by rotating the original training data by +15/-15 degree.

The number of training data became triple from original, which is 104,397.

Validation Accuracy became 93.7%, but it's better to make it more robust.

[Check #4] I added training data by zooming up/down the original training data by +2/-2 pixels.

The number of training data became five times from original, which is 173,995.

Validation Accuracy became 94.3%, but it's not good enough because Test Accuracy was 92.4%.

I found that it's better to change the parameters of the model to get better results.

```
### Preprocess the data here. It is required to normalize the data. Other preprocessing steps could include  
### converting to grayscale, etc.  
### Feel free to use as many code cells as needed.
```

```
##(1)Without normalization  
#Test1: Only this, accuracy is 89.2%
```

```
##(3)Add rotated pictures in training data  
#Test3: 93.7% with 104,397 training data  
#import cv2
```

```
##(a)Initial parameters  
height = X_train[0].shape[0]  
width = X_train[0].shape[1]  
center = (int(width/2), int(height/2))  
scale = 1.0  
angle1 = 15.0  
angle2 = -15.0  
trans1 = cv2.getRotationMatrix2D(center, angle1, scale)  
trans2 = cv2.getRotationMatrix2D(center, angle2, scale)
```

```
X_train_base = np.copy(X_train)  
y_train_base = np.copy(y_train)
```

```
X_train_trans1 = np.copy(X_train_base)  
X_train_trans2 = np.copy(X_train_base)
```

```
##(b)Calculation  
for i in range(n_train):  
    X_train_trans1[i] = cv2.warpAffine(X_train[i], trans1, (width,height))  
    X_train_trans2[i] = cv2.warpAffine(X_train[i], trans2, (width,height))
```

```
##(c)Append to the training data (Training size will be 3 times as original)  
X_train = np.append(X_train, X_train_trans1, axis=0)  
y_train = np.append(y_train, y_train_base, axis=0)
```

```
X_train = np.append(X_train, X_train_trans2, axis=0)  
y_train = np.append(y_train, y_train_base, axis=0)
```

```
##(4)Zoom in/out pictures in training data  
#Test4: % with training data  
##(a)Initial parameters  
pixels = 2
```

```
X_train_zoomup = np.copy(X_train_base)  
X_train_zoomout = np.copy(X_train_base)  
#y_train_base = np.copy(y_train) # Already done
```

```

zoomup_rate = (X_train_base[0].shape[0]+2*pixels)/X_train_base[0].shape[0]
zoomout_rate = (X_train_base[0].shape[0]-2*pixels)/X_train_base[0].shape[0]

#(b)Calculation
for i in range(n_train):
    zoom_img = cv2.resize(X_train_base[i], None, fx=zoomup_rate, fy=zoomup_rate, interpolation=cv2.INTER_CUBIC)
    X_train_zoomup[i] = zoom_img[pixels:pixels+X_train_base[0].shape[0], pixels:pixels+X_train_base[0].shape[0]]

    zoom_img = cv2.resize(X_train_base[i], None, fx=zoomout_rate, fy=zoomout_rate, interpolation=cv2.INTER_CUBIC)
    X_train_zoomout[i] = np.pad(zoom_img, ((pixels,pixels),(pixels,pixels), (0,0)), 'constant')

#(c)Append to the training data (Training size will be 5 times as original, after rotation and zoom in/out)
X_train = np.append(X_train, X_train_zoomup, axis=0)
y_train = np.append(y_train, y_train_base, axis=0)

X_train = np.append(X_train, X_train_zoomout, axis=0)
y_train = np.append(y_train, y_train_base, axis=0)

```

```

#(2)Gray Scale and Normalization
#Test2: 92.5% with 34,799 training data
#import cv2
#(a)Gray Scale
for i in range(n_train):
    X_train[i] = np.expand_dims(cv2.cvtColor(X_train[i], cv2.COLOR_RGB2GRAY), axis=3)

for i in range(n_validation):
    X_valid[i] = np.expand_dims(cv2.cvtColor(X_valid[i], cv2.COLOR_RGB2GRAY), axis=3)

for i in range(n_test):
    X_test[i] = np.expand_dims(cv2.cvtColor(X_test[i], cv2.COLOR_RGB2GRAY), axis=3)

# Size change: 32x32x3 ----> 32x32x1
X_train = X_train[:, :, :, 0:1]
X_valid = X_valid[:, :, :, 0:1]
X_test = X_test[:, :, :, 0:1]

```

```

#(b)Normalization
X_train = (X_train - np.mean(X_train)) / np.std(X_train)
X_valid = (X_valid - np.mean(X_valid)) / np.std(X_valid)
X_test = (X_test - np.mean(X_test)) / np.std(X_test)

#Confirmation
print("The shape of loaded X_train is ", X_train.shape)
print("The shape of loaded y_train is ", y_train.shape)
print("The shape of loaded X_valid is ", X_valid.shape)
print("The shape of loaded y_valid is ", y_valid.shape)
print("The shape of loaded X_test is ", X_test.shape)
print("The shape of loaded y_test is ", y_test.shape)

#Save pickle
X_train_file = './augmented_X_train.p'
y_train_file = './augmented_y_train.p'
with open(X_train_file, mode='wb') as f:
    pickle.dump(X_train, f)
with open(y_train_file, mode='wb') as f:
    pickle.dump(y_train, f)

```

```

The shape of loaded X_train is (173995, 32, 32, 1)
The shape of loaded y_train is (173995,)
The shape of loaded X_valid is (4410, 32, 32, 1)
The shape of loaded y_valid is (4410,)
The shape of loaded X_test is (12630, 32, 32, 1)
The shape of loaded y_test is (12630,)

```

The following code shows the final model of this project.

[Check #5] The original LeNet has output dimension: 10 (0, 1, ..., 8, 9), but our project has **output dimension 43**.

It's almost 4 times bigger than original one, so I made **the depth of the hidden layers 4 times bigger than original**.

```
### Define your architecture here.
### Feel free to use as many code cells as needed.
#Try. 2

#(1)Setup TensorFlow
#The EPOCH and BATCH_SIZE values affect the training speed and model accuracy.
#import tensorflow as tf

EPOCHS = 10
BATCH_SIZE = 128

#(2)Implement LeNet-5
#from tensorflow.contrib.layers import flatten

def LeNet(x):
    # Arguments used for tf.truncated_normal, randomly defines variables for the weights and biases for each layer
    mu = 0
    sigma = 0.1

    # SOLUTION: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x24. # 4 times
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 24), mean = mu, stddev = sigma)) # 4 times
    #conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 3, 6), mean = mu, stddev = sigma)) # Color has 3 dimensions
    conv1_b = tf.Variable(tf.zeros(24)) # 4 times
    conv1 = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b

    # SOLUTION: Activation.
    conv1 = tf.nn.relu(conv1)

    # SOLUTION: Pooling. Input = 28x28x6. Output = 14x14x24. # 4 times
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # SOLUTION: Layer 2: Convolutional. Output = 10x10x64. # 4 times
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 24, 64), mean = mu, stddev = sigma)) # 4 times
    conv2_b = tf.Variable(tf.zeros(64)) # 4 times
    conv2 = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALID') + conv2_b

    # SOLUTION: Activation.
    conv2 = tf.nn.relu(conv2)

    # SOLUTION: Pooling. Input = 10x10x16. Output = 5x5x64. # 4 times
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # SOLUTION: Flatten. Input = 5x5x64. Output = 1600. # 4 times
    fc0 = flatten(conv2)

    # SOLUTION: Layer 3: Fully Connected. Input = 1600. Output = 480. # 4 times
    fc1_W = tf.Variable(tf.truncated_normal(shape=(1600, 480), mean = mu, stddev = sigma)) # 4 times
    fc1_b = tf.Variable(tf.zeros(480)) # 4 times
    fc1 = tf.matmul(fc0, fc1_W) + fc1_b

    # SOLUTION: Activation.
    fc1 = tf.nn.relu(fc1)

    # SOLUTION: Layer 4: Fully Connected. Input = 480. Output = 336. # 4 times
    fc2_W = tf.Variable(tf.truncated_normal(shape=(480, 336), mean = mu, stddev = sigma)) # 4 times
    fc2_b = tf.Variable(tf.zeros(336)) # 4 times
    fc2 = tf.matmul(fc1, fc2_W) + fc2_b

    # SOLUTION: Activation.
    fc2 = tf.nn.relu(fc2)

    # SOLUTION: Layer 5: Fully Connected. Input = 336. Output = 43. # 4 times
    #fc3_W = tf.Variable(tf.truncated_normal(shape=(34, 10), mean = mu, stddev = sigma))
    fc3_W = tf.Variable(tf.truncated_normal(shape=(336, 43), mean = mu, stddev = sigma)) # Output has 43 dimensions
    #fc3_b = tf.Variable(tf.zeros(10))
    fc3_b = tf.Variable(tf.zeros(43)) # Output has 43 dimensions
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

```

#(3)Features and Labels
#Train LeNet to classify MNIST data.
#x is a placeholder for a batch of input images. y is a placeholder for a batch of output labels.

x = tf.placeholder(tf.float32, (None, 32, 32, 1))
#x = tf.placeholder(tf.float32, (None, 32, 32, 3)) # Color has 3 dimensions
y = tf.placeholder(tf.int32, (None))
#one_hot_y = tf.one_hot(y, 10)
one_hot_y = tf.one_hot(y, 43) # Output has 43 dimensions

#(4)Setting for Training Pipeline
rate = 0.001

logits = LeNet(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logits=logits)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)

```

The source code of training is shown below.

```

def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offset+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples

```

```

### Train your model here.
### Calculate and report the accuracy on the training and validation set.
### Once a final model architecture is selected,
### the accuracy on the test set should be calculated and reported as well.
### Feel free to use as many code cells as needed.
#Try.2

```

```

#(6)Train the Model
#Run the training data through the training pipeline to train the model.
#Before each epoch, shuffle the training set.
#After each epoch, measure the loss and accuracy of the validation set.
#Save the model after training.

```

#This needs GPU.

#from sklearn.utils import shuffle

```

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)

    print("Training...")
    print()
    for i in range(EPOCHS):
        X_train, y_train = shuffle(X_train, y_train)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train[offset:end], y_train[offset:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y})

        validation_accuracy = evaluate(X_valid, y_valid)
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

    saver.save(sess, './lenet')
    print("Model saved")

```

The results of Validation Accuracy is shown below. Final Validation Accuracy is **94.4%**, which is better than the Target Accuracy: 93%. It can have better result by Early Termination at **EPOCH 7~9**, but it may have better results with other Test Data, so I proceed by taking this trained model.

Training...	EPOCH 4 ... Validation Accuracy = 0.945	EPOCH 8 ... Validation Accuracy = <u>0.957</u>
EPOCH 1 ... Validation Accuracy = 0.926	EPOCH 5 ... Validation Accuracy = 0.945	EPOCH 9 ... Validation Accuracy = <u>0.958</u>
EPOCH 2 ... Validation Accuracy = 0.941	EPOCH 6 ... Validation Accuracy = 0.948	<div style="border: 2px solid blue; padding: 2px;">EPOCH 10 ... Validation Accuracy = 0.944</div>
EPOCH 3 ... Validation Accuracy = 0.955	EPOCH 7 ... Validation Accuracy = <u>0.958</u>	Model saved

I evaluated the Model with Test Data.

Test Accuracy is 94.3%, so it passes the Target Accuracy.

```
#(7)Evaluate the Model  
#Once you are completely satisfied with your model, evaluate the performance of the model on the test set.  
#Be sure to only do this once!  
#If you were to measure the performance of your trained model on the test set, then improve your model,  
# and then measure the performance of your model on the test set again, that would invalidate your test results.  
#You wouldn't get a true measure of how well your model would perform against real data.  
  
with tf.Session() as sess:  
    saver.restore(sess, tf.train.latest_checkpoint('.'))  
  
    test_accuracy = evaluate(X_test, y_test)  
    print("Test Accuracy = {:.3f}".format(test_accuracy))
```

```
INFO:tensorflow:Restoring parameters from ./lenet  
Test Accuracy = 0.943
```

<Additional page at 2nd submission, August 11th>

I got 3 feedbacks at the 1st submission, so I added this and next page with 2nd submission.

(1) Details of the characteristics and qualities of the architecture

I describe my model architecture in detail as below. The original is LeNet-5, and I modified dimensions.

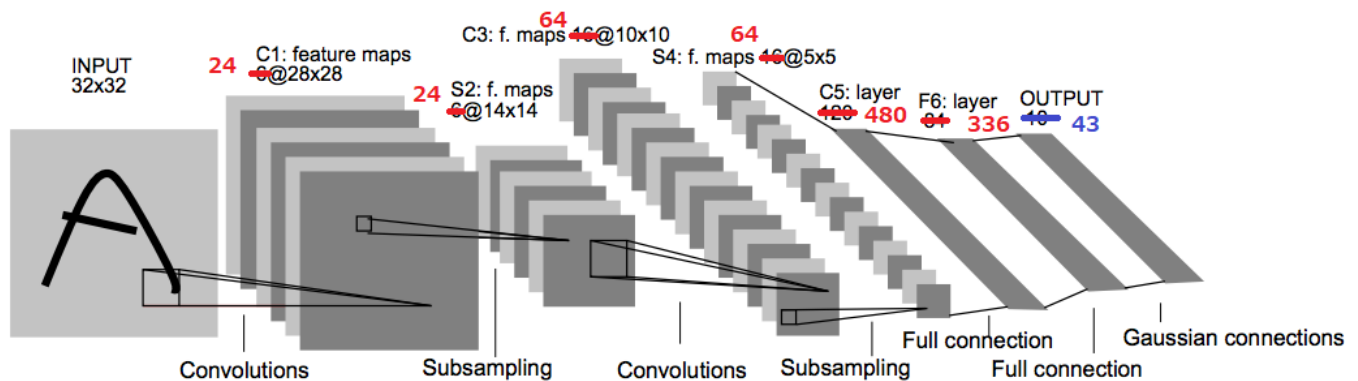


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Layer	Description
Input	32x32x1 Grayscale image
C1: Convolution	1x1 stride, valid padding, output = 28x28x24
Activation function: RELU	
S2: Max pooling	2x2 stride, valid padding, output = 14x14x24
C3: Convolution	1x1 stride, valid padding, output = 10x10x64
Activation function: RELU	
S4: Max pooling	2x2 stride, valid padding, output = 5x5x64
C5: Fully Connected	output = 480
Activation function: RELU	
F6: Fully Connected	output = 336
Activation function: RELU	
Output: Softmax	output = 43

*: For each Convolution and Fully connected, I used the same parameter: Mean=0, stddev=0.1

(2) How the model was trained

To train this model, I used the "Adam Optimizer".

At first, I used the same parameters as the Lesson shown in below.

- Batch size: 128
- Number of epochs: 10
- Learning rate: 0.001

(3) Approach to find a solution

(a) Choice of architecture

As shown in the previous page, I chose LeNet-5 as the original model.

In the paper it was used to classify characters, but I thought it can be used to classify Traffic Sign images by the following reasons:

- LeNet-5 is the basic Convolutional Neural Network, which was referred and arranged by many people to classify more complex images.
- It deals with input images with 32x32 size, which is good size for Traffic Sign images.

(b) Approach and results

To check which item works well, I ran training every time for each item.

Check item	Accuracy		Comments
	Validation	Test	
1.Default	89.2%	-	I used LeNet-5 for Traffic Sign images without any change at first, and Validation Accuracy is much less than 93%.
2.Grayscale and Normalization	92.5%	-	When I saw the examples of Traffic Sign images in page 4, there are many images whose color are dark and difficult to identify only by color. On the contrary, shape of image looks important to classify each sign. I applied Grayscale and Normalization to Traffic Sign images and accuracy improved a lot as I assumed. However, it's not good enough.
3.Rotate (+15/-15 degree)	93.7%	-	Same as above, there are some images who lean toward left or right. I increased training data by rotating the original train images, and accuracy improved.
4.Zoom (+2/-2 pixels)	94.3%	92.4%	Same as above, there are some images whose size is bigger or smaller. I increased training data by zooming the original train images, and accuracy improved. However, Test Accuracy is less than 93%, so I thought that there's over fitting with training data and validation data. I figured out that I need to change something with the Model to get the similar amount of Test Accuracy.
5.Change depth of hidden layers	94.4%	94.3%	Output classes of LeNet-5 is 10 (0, 1, ..., 8, 9), but output classes of Traffic Sign images is 43. It's almost 4 times bigger than LeNet-5, so I figured out that I need deeper data in the hidden layers. I made depth 4 times bigger than LeNet-5 as shown in the previous page, then both Validation Accuracy / Test Accuracy achieved the target accuracy: more than 93%.

Step 3: Test a Model on New Images

I downloaded **10 new Test Images** by the following Website, and evaluated the Model with them.

<https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign/version/1>

These images have different image size and data type is float, so I modified them into good data type with the Model. The size of Test Data by new images is **10(image numbers)x32x32(size)x3(RGB colors)**.

```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
```

```
#Download Source: https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign/version/1
#There are 2 points to modify with these data as below:
#(a)Data size: ** x ** x 3 --> 32 x 32 x 3
#(b)Data type: float --> int conversion
```

```
#(1) Import pictures
#import glob
#import re
#import cv2
```

```
Test_pics = np.array(glob.glob('Test_pics/*.png'))
new_test_images = [plt.imread(file) for file in Test_pics]
```

```
#(2) Initialize
#new_test_modify = np.zeros([32, 32, 3])
X_data = np.zeros([1, 32, 32, 3])
y_data = []
```

```
#(3)Test
#print(Test_pics)
#classes= re.findall(r"%d+", Test_pics[9])[0]
#print(classes)
```

```
#(4)Modify raw data and make Test data
plt.figure(figsize=(10,10))
for i, image in enumerate(new_test_images):
    xrate = 32./new_test_images[i].shape[1]
    yrate = 32./new_test_images[i].shape[0]
    new_test_resize = cv2.resize(new_test_images[i], None, fx=xrate, fy=yrate, interpolation=cv2.INTER_CUBIC) #(a)Data size modification
    new_test_modify = (new_test_resize * 256).astype(np.uint8) #(b)Data type conversion
```

```
X_data = np.append(X_data, np.expand_dims(new_test_modify,axis=0), axis=0)
classes= int(re.findall(r"%d+", Test_pics[i])[0])
y_data.append(classes)
```

```
plt.subplot(5, len(Test_pics)//5, i+1)
plt.title('%s %' % (sign_names[classes]))
plt.imshow(image)
plt.xticks([])
plt.yticks([])
```

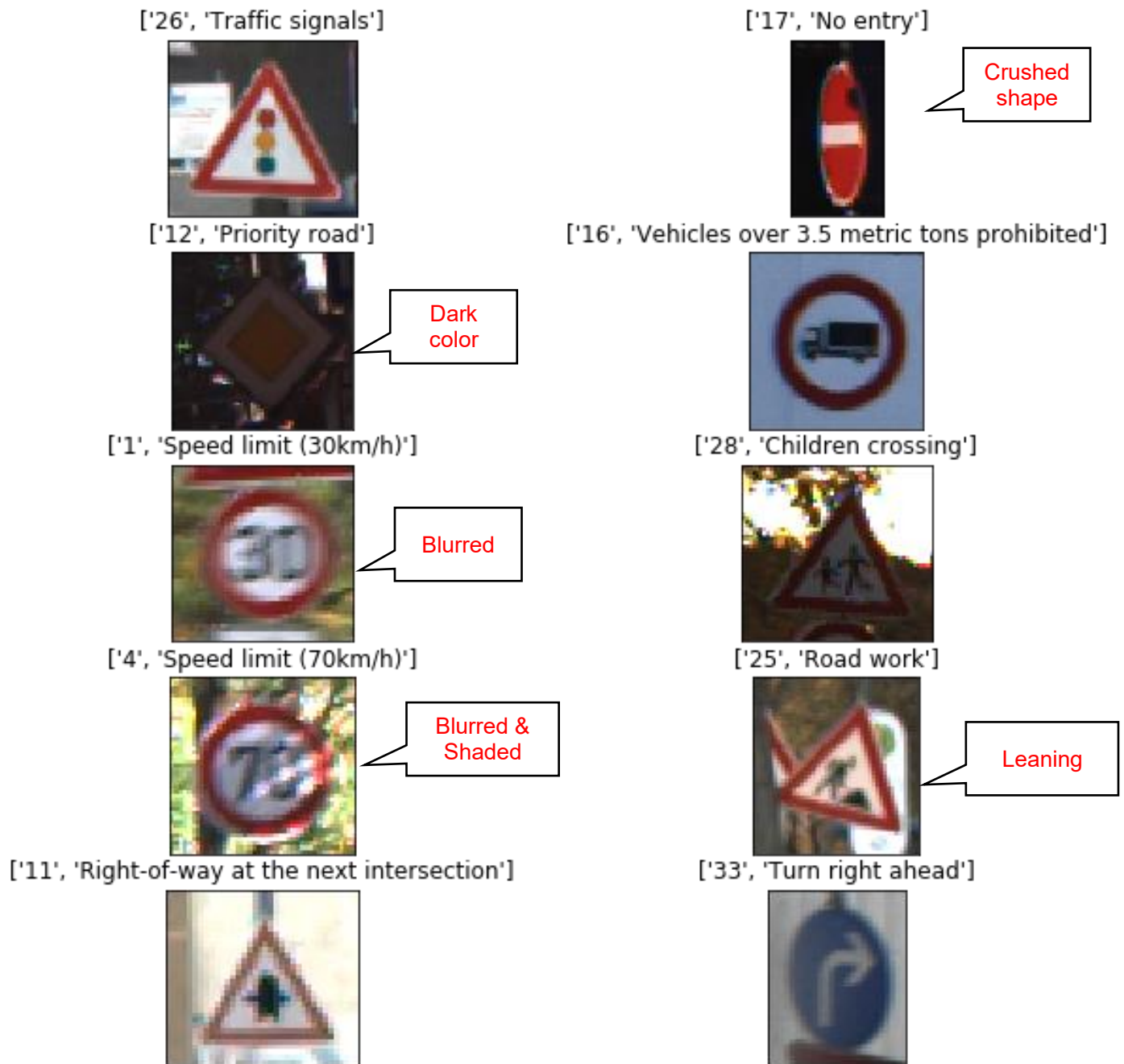
```
plt.show()
```

```
# (N+1)x32x32xC ----> Nx32x32xC
X_data = X_data[1: , ...].astype(np.uint8)
```

```
print('X data shape:', np.array(X_data).shape)
```

```
X data shape: (10, 32, 32, 3)
```

I chose 10 images. **Some of them are challenging for classification.**



I adapted Gray scale and Normalization to the new 10 images same as other image data, and I ran the prediction of the Sign Type for each image.

```
### Run the predictions here and use the model to output the prediction for each image.
### Make sure to pre-process the images with the same pre-processing pipeline used earlier.
### Feel free to use as many code cells as needed.

#(0) Initialize
X_data_test = np.copy(X_data)

#(1) Gray Scale and Normalization
#(a) Gray Scale
for i in range(len(X_data)):
    X_data_test[i] = np.expand_dims(cv2.cvtColor(X_data_test[i], cv2.COLOR_RGB2GRAY), axis=3)

# Size change: 32x32x3 ---> 32x32x1
X_data_test = X_data_test[:, :, :, 0:1]

#(b) Normalization
X_data_test = (X_data_test - np.mean(X_data_test)) / np.std(X_data_test)
```

The result of prediction matched the labeled answer **100%**, so my Model is well trained for new images as well.

##(2)Run predictions

```
with tf.Session() as sess:
    saver.restore(sess, './lenet')
    #saver = tf.train.import_meta_graph('lenet.meta') #doesn't work well
    #saver.restore(sess, tf.train.latest_checkpoint('./')) #doesn't work well
    predicted_logits = sess.run(logits, feed_dict={x: X_data_test})
    predicted_labels = np.argmax(predicted_logits, axis=1)

    print("Result of prediction: %n", predicted_labels)

print("Answer: %n", y_data)
```

INFO:tensorflow:Restoring parameters from ./lenet

```
Result of prediction:
[26 17 12 16  1 28  4 25 11 33]
Answer:
[26, 17, 12, 16, 1, 28, 4, 25, 11, 33]
```

Calculate the accuracy for these 5 new images.

For example, if the model predicted 1 out of 5 signs correctly, it's 20% accurate on these new images.

```
length = len(y_data)
score = 0

for i in range(length):
    if (y_data[i] == predicted_labels[i]):
        score += 1
```

```
accuracy = score/length * 100
print("Accuracy for the new images is :~ + str(accuracy) + "%.")

Accuracy for the new images is :100.0%.
```

Finally, I show the Output Top 5 Softmax Probabilities for each image found on the Web site.

Print out the top five softmax probabilities for the predictions on the German traffic sign images found on the web.

Feel free to use as many code cells as needed.

```
with tf.Session() as sess:
    softmax = tf.nn.softmax(predicted_logits)
    results = sess.run(tf.nn.top_k(softmax, k=5))

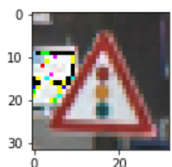
for x in range(len(y_data)):
    print("Original Image (size is random): ~ + str(x+1))
    plt.figure(figsize=(2,2))
    plt.imshow(new_test_images[x])
    plt.show()
    print("Modified Image (size: 32x32x3): ~ + str(x+1))
    plt.figure(figsize=(2,2))
    plt.imshow(X_data[x])
    plt.show()
    if(sign_names[y_data[x]] == sign_names[predicted_labels[x]]):
        print("{0}{1}:".format('Correct prediction: ', sign_names[y_data[x]]))
    else:
        print("{0}{1}:".format('Incorrect prediction: ', sign_names[y_data[x]]))
    for y in range(5):
        print("{s}: {:.2f}%".format(str(sign_names[results[1][x][y]]), results[0][x][y]*100))

    print()
```

Original Image (size is random): 1

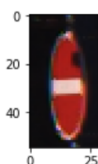


Modified Image (size: 32x32x3): 1

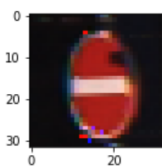


Correct prediction: ['26', 'Traffic signals']:
['26', 'Traffic signals']: 100.00%
['18', 'General caution']: 0.00%
['37', 'Go straight or left']: 0.00%
['35', 'Ahead only']: 0.00%
['5', 'Speed limit (80km/h)']: 0.00%

Original Image (size is random): 2

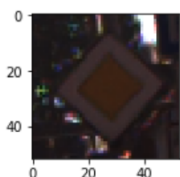


Modified Image (size: 32x32x3): 2

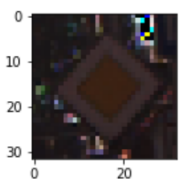


Correct prediction: ['17', 'No entry']:
['17', 'No entry']: 100.00%
['38', 'Keep right']: 0.00%
['9', 'No passing']: 0.00%
['40', 'Roundabout mandatory']: 0.00%
['20', 'Dangerous curve to the right']: 0.00%

Original Image (size is random): 3

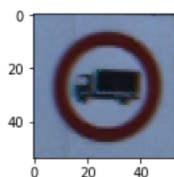


Modified Image (size: 32x32x3): 3

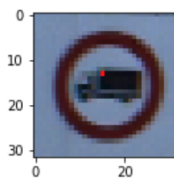


Correct prediction: ['12', 'Priority road']:
['12', 'Priority road']: 100.00%
['13', 'Yield']: 0.00%
['5', 'Speed limit (80km/h)']: 0.00%
['9', 'No passing']: 0.00%
['15', 'No vehicles']: 0.00%

Original Image (size is random): 4

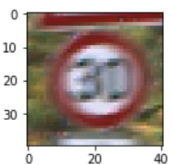


Modified Image (size: 32x32x3): 4

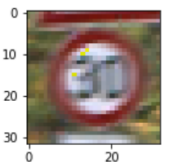


Correct prediction: ['16', 'Vehicles over 3.5 metric tons prohibited']:
['16', 'Vehicles over 3.5 metric tons prohibited']: 100.00%
['40', 'Roundabout mandatory']: 0.00%
['7', 'Speed limit (100km/h)']: 0.00%
['41', 'End of no passing']: 0.00%
['37', 'Go straight or left']: 0.00%

Original Image (size is random): 5

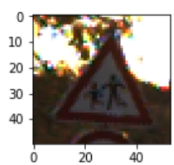


Modified Image (size: 32x32x3): 5

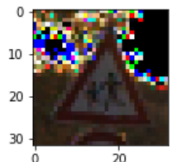


Correct prediction: ['1', 'Speed limit (30km/h)']:
['1', 'Speed limit (30km/h)']: 100.00%
['4', 'Speed limit (70km/h)']: 0.00%
['40', 'Roundabout mandatory']: 0.00%
['0', 'Speed limit (20km/h)']: 0.00%
['8', 'Speed limit (120km/h)']: 0.00%

Original Image (size is random): 6

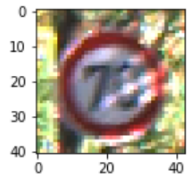


Modified Image (size: 32x32x3): 6

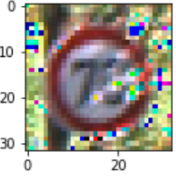


Correct prediction: ['28', 'Children crossing']:
['28', 'Children crossing']: 100.00%
['24', 'Road narrows on the right']: 0.00%
['29', 'Bicycles crossing']: 0.00%
['11', 'Right-of-way at the next intersection']: 0.00%
['5', 'Speed limit (80km/h)']: 0.00%

Original Image (size is random): 7

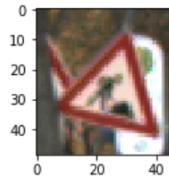


Modified Image (size: 32x32x3): 7

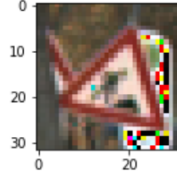


Correct prediction: ['4', 'Speed limit (70km/h)']:
['4', 'Speed limit (70km/h)']: 100.00%
['24', 'Road narrows on the right']: 0.00%
['0', 'Speed limit (20km/h)']: 0.00%
['1', 'Speed limit (30km/h)']: 0.00%
['2', 'Speed limit (50km/h)']: 0.00%

Original Image (size is random): 8

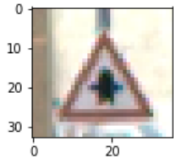


Modified Image (size: 32x32x3): 8

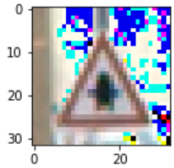


Correct prediction: ['25', 'Road work']:
['25', 'Road work']: 85.34%
['33', 'Turn right ahead']: 14.66%
['20', 'Dangerous curve to the right']: 0.00%
['11', 'Right-of-way at the next intersection']: 0.00%
['37', 'Go straight or left']: 0.00%

Original Image (size is random): 9

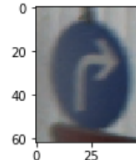


Modified Image (size: 32x32x3): 9

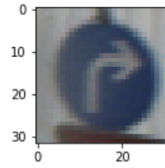


Correct prediction: ['11', 'Right-of-way at the next intersection']:
['11', 'Right-of-way at the next intersection']: 100.00%
['27', 'Pedestrians']: 0.00%
['30', 'Beware of ice/snow']: 0.00%
['24', 'Road narrows on the right']: 0.00%
['28', 'Children crossing']: 0.00%

Original Image (size is random): 10



Modified Image (size: 32x32x3): 10



Correct prediction: ['33', 'Turn right ahead']:
['33', 'Turn right ahead']: 100.00%
['39', 'Keep left']: 0.00%
['37', 'Go straight or left']: 0.00%
['35', 'Ahead only']: 0.00%
['20', 'Dangerous curve to the right']: 0.00%

1st prediction is 85.3% and 2nd prediction is 14.7% with No.8, but mostly the 1st prediction is 100%.