Self-Driving Car Engineer Nanodegree

Deep Learning

Project: Build a Traffic Sign Recognition Classifier

In this notebook, a template is provided for you to implement your functionality in stages which is required to successfully complete this project. If additional code is required that cannot be included in the notebook, be sure that the Python code is successfully imported and included in your submission, if necessary. Sections that begin with **'Implementation'** in the header indicate where you should begin your implementation for your project. Note that some sections of implementation are optional, and will be marked with **'Optional'** in the header.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

Imports

In [1]:

```
import pickle
import time
import os
import scipy.ndimage

import numpy as np
import pandas as pd
import tensorflow as tf
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import matplotlib.gridspec as gridspec

from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score, f1_score
```

```
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn import decomposition
import cv2
```

Supporting Functions

In [2]:

```
#one hot coding function
def OHE_Encode(Y_tr,N_classes):
    OHC = OneHotEncoder()
    Y_ohc = OHC.fit(np.arange(N_classes).reshape(-1, 1))
    Y_labels = Y_ohc.transform(Y_tr.reshape(-1, 1)).toarray()
    return Y_labels

def OHE_Validate(cls,y):
    check = np.linalg.norm(np.argmax(cls,axis=1)-y)
    if check == 0:
        print('One hot encoding Validated')
    else:
        print('One hot encoding doesnt match the output, check code!!!')
```

Step 0: Load The Data

In [3]:

```
print('Tensor Flow version : '+tf.__version__)
training_file = 'traffic-signs-data/train.p'
testing_file = 'traffic-signs-data/test.p'

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

X_train, y_train = train['features'], train['labels']
X_test, y_test = test['features'], test['labels']

assert(len(X_train)== len(y_train))
assert(len(X_test)== len(y_test))

#data_pd = pd.read_csv('signnames.csv')
#data_pd.head()
```

```
Tensor Flow version : 0.12.1
```

Step 1: Dataset Summary & Exploration

The pickled data is a dictionary with 4 key/value pairs:

- `'features'` is a 4D array containing raw pixel data of the traffic sign images, (num examples, width, height, channels).
- `'labels'` is a 2D array containing the label/class id of the traffic sign. The file `signnames.csv` contains id -> name mappings for each id.
- `'sizes'` is a list containing tuples, (width, height) representing the the original width and height the image.
- `'coords'` is a list containing tuples, (x1, y1, x2, y2) representing coordinates of a bounding box around the sign in the image. **THESE COORDINATES ASSUME THE ORIGINAL IMAGE. THE PICKLED DATA CONTAINS RESIZED VERSIONS (32 by 32) OF THESE IMAGES**

Complete the basic data summary below.

In [4]:

```
### Replace each question mark with the appropriate value.

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

# 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.
n_classes = len(set(y_train))

# Number of Channels
n_channels = X_train[0].shape[2]

print("Number of training examples =", n_train)
print("Number of testing examples =", n_test)
print("Image data shape =", image_shape)
print('Training labels shape', y_train.shape)
print("Number of classes =", n_classes)
print("Number of channels =", n_channels)
print("Number of features =", X_train.shape[1])

#One Hot Encoding
y_train_OneHot = OHE_Encode(y_train,n_classes)
y_test_OneHot = OHE_Encode(y_test,n_classes)

OHE_Validate(y_test_OneHot ,y_test)
OHE_Validate(y_train_OneHot,y_train)
```

```
Number of training examples = 39209
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Training labels shape (39209,)
Number of classes = 43
Number of channels = 3
Number of features = 32
One hot encoding Validated
One hot encoding Validated
```

Visualize the German Traffic Signs Dataset using the pickled file(s). This is open ended, suggestions include: plotting traffic sign images, plotting the count of each sign, etc.

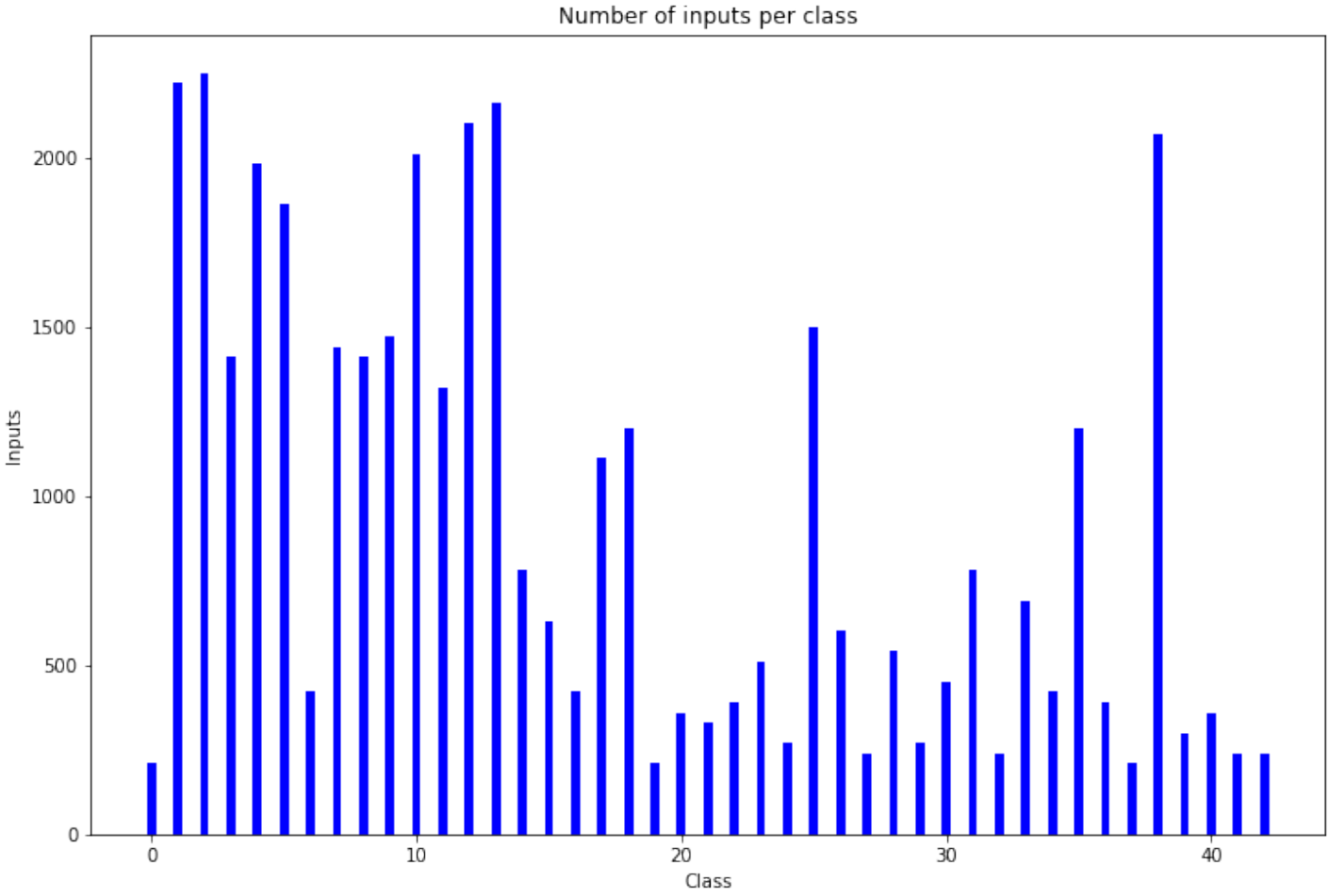
The Matplotlib examples and gallery pages are a great resource for doing visualizations in Python.

NOTE: It's recommended you start with something simple first. If you wish to do more, come back to it after you've completed the rest of the sections.

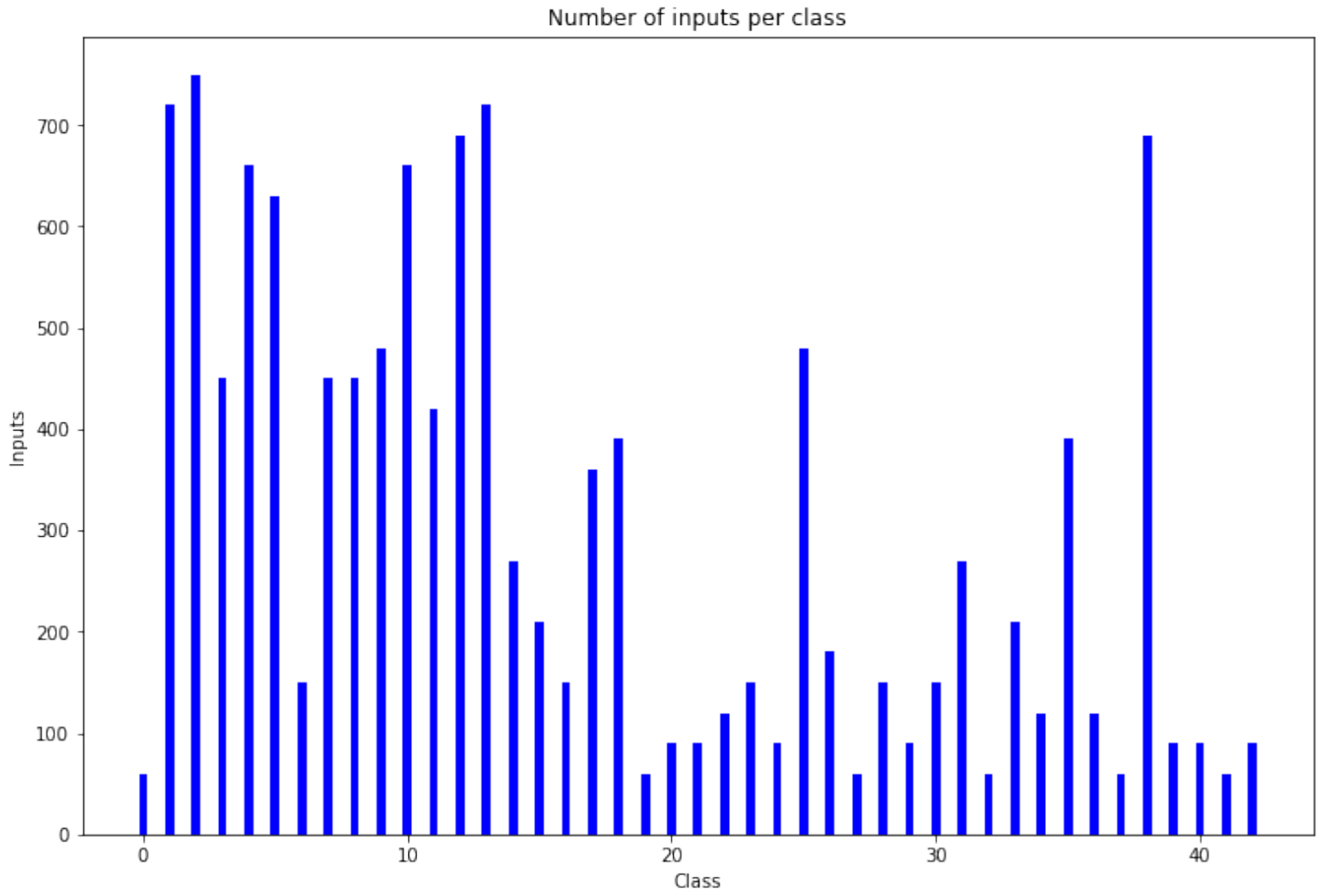
In [5]:

```
print('Training Data')
train_features = np.array(train['features'])
train_labels = np.array(train['labels'])

inputs_per_class = np.bincount(train_labels)
max_inputs = np.max(inputs_per_class)

mpl_fig = plt.figure(figsize=(12,8))
ax = mpl_fig.add_subplot(111)
ax.set_ylabel('Inputs')
ax.set_xlabel('Class')
ax.set_title('Number of inputs per class')
ax.bar(range(len(inputs_per_class)), inputs_per_class, 1/3, color='blue',
label='Inputs per class')
plt.show()


print('Testing Data')
test_features = np.array(test['features'])
test_labels = np.array(test['labels'])

inputs_per_class = np.bincount(test_labels)
max_inputs = np.max(inputs_per_class)

mpl_fig = plt.figure(figsize=(12,8))
ax = mpl_fig.add_subplot(111)
ax.set_ylabel('Inputs')
ax.set_xlabel('Class')
ax.set_title('Number of inputs per class')
ax.bar(range(len(inputs_per_class)), inputs_per_class, 1/3, color='blue',
label='Inputs per class')
plt.show()
```

```
Training Data
```



Testing Data



In [6]:

```
data_i = [[i,sum(y_train == i)] for i in range(len(np.unique(y_train)))]
data_i_sorted = sorted(data_i, key=lambda x: x[1])
data_pd = pd.read_csv('signnames.csv')
data_pd['Occurance'] = pd.Series(np.asarray(data_i_sorted).T[1],
index=np.asarray(data_i_sorted).T[0])
data_pd_sorted = data_pd.sort_values(['Occurance'],ascending=[0]).reset_index()
data_pd_sorted = data_pd_sorted.drop('index', 1)
data_pd_sorted
```

Out[6]:

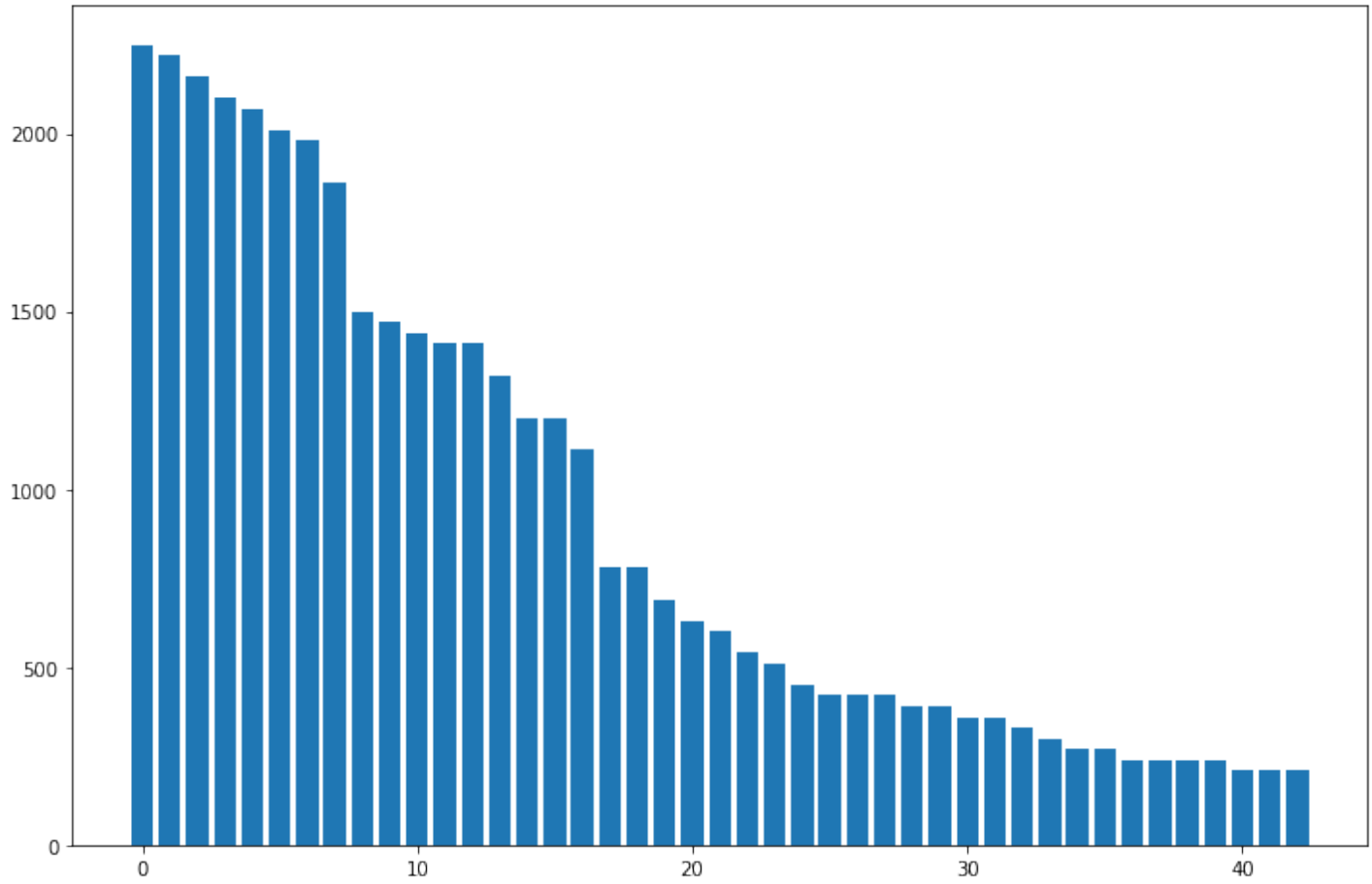
	ClassId	SignName	Occurance
0	2	Speed limit (50km/h)	2250
1	1	Speed limit (30km/h)	2220
2	13	Yield	2160
3	12	Priority road	2100
4	38	Keep right	2070
5	10	No passing for vehicles over 3.5 metric tons	2010
6	4	Speed limit (70km/h)	1980
7	5	Speed limit (80km/h)	1860

8	25	Road work	1500
9	9	No passing	1470
10	7	Speed limit (100km/h)	1440
11	3	Speed limit (60km/h)	1410
12	8	Speed limit (120km/h)	1410
13	11	Right-of-way at the next intersection	1320
14	35	Ahead only	1200
15	18	General caution	1200
16	17	No entry	1110
17	31	Wild animals crossing	780
18	14	Stop	780
19	33	Turn right ahead	689
20	15	No vehicles	630
21	26	Traffic signals	600
22	28	Children crossing	540
23	23	Slippery road	510
24	30	Beware of ice/snow	450
25	16	Vehicles over 3.5 metric tons prohibited	420
26	34	Turn left ahead	420
27	6	End of speed limit (80km/h)	420
28	36	Go straight or right	390
29	22	Bumpy road	390
30	40	Roundabout mandatory	360
31	20	Dangerous curve to the right	360
32	21	Double curve	330
33	39	Keep left	300
34	29	Bicycles crossing	270
35	24	Road narrows on the right	270
36	41	End of no passing	240
37	42	End of no passing by vehicles over 3.5 metric ...	240
38	32	End of all speed and passing limits	240
39	27	Pedestrians	240
40	37	Go straight or left	210

41	19	Dangerous curve to the left	210
42	0	Speed limit (20km/h)	210

In [7]:

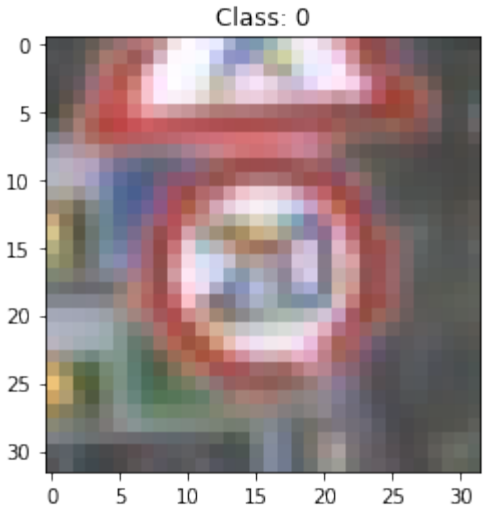
```
plt.figure(figsize=(12,8))
plt.bar(range(n_classes),height=data_pd_sorted["Occurance"])
plt.show()
```

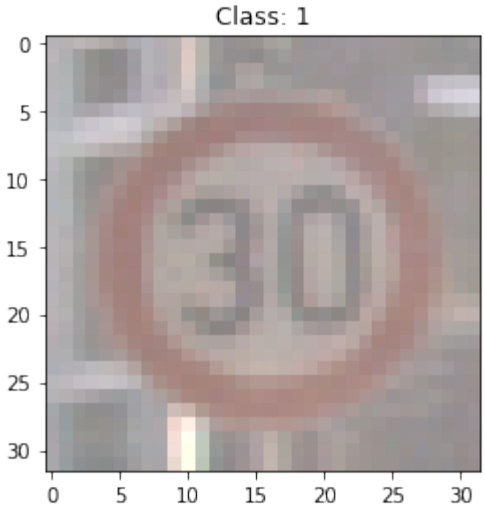


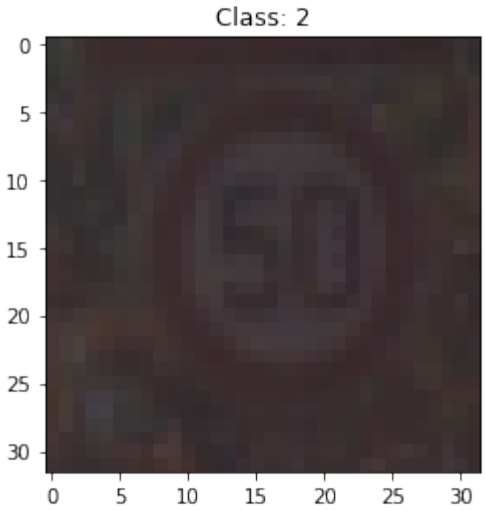
In [8]:

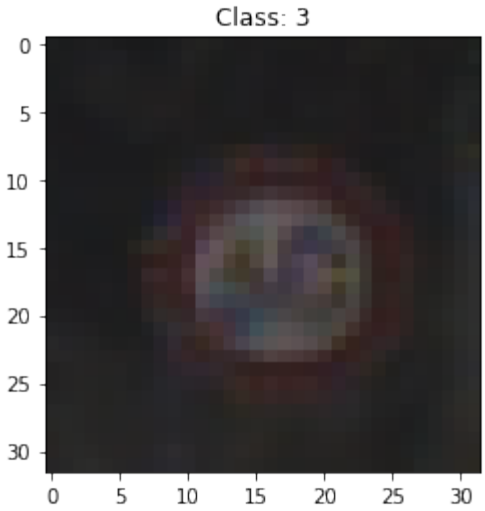
```
sign_dict = {}

print('List of Classes')
mpl_fig = plt.figure()
for i in range(n_classes):
    for j in range(len(train_labels)):
        if (i == train_labels[j]):
            sign_dict[i]=train_features[j]
            plt.title('Class: '+ str(i))
            plt.imshow(train_features[j])
            plt.show()
            break
```

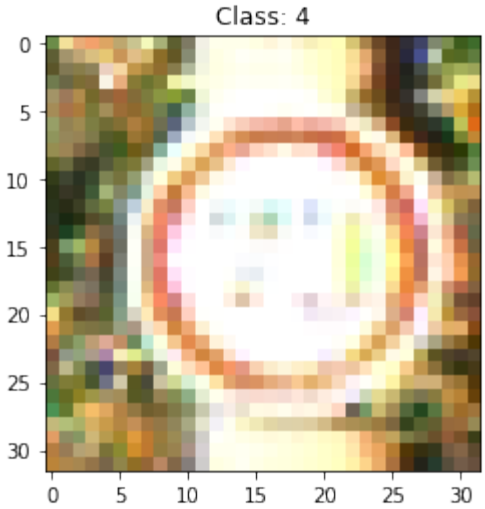
```
List of Classes
```

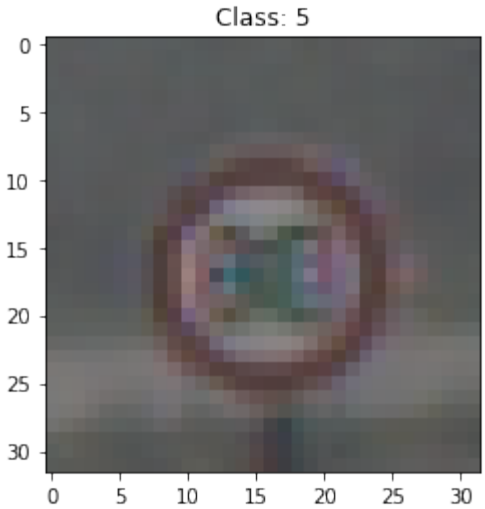

Class: 0

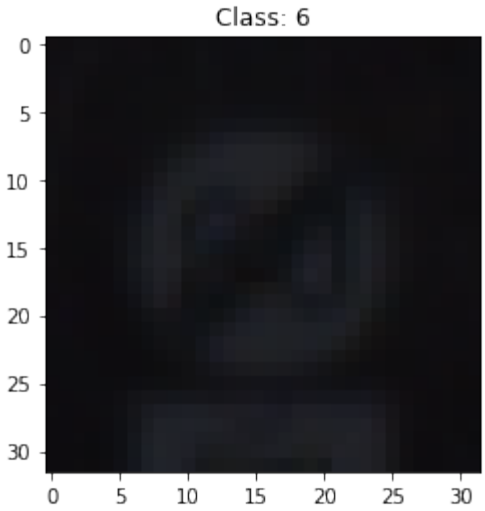

Class: 1

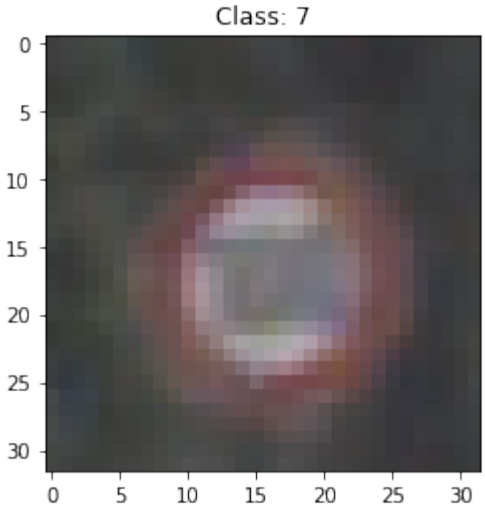

Class: 2

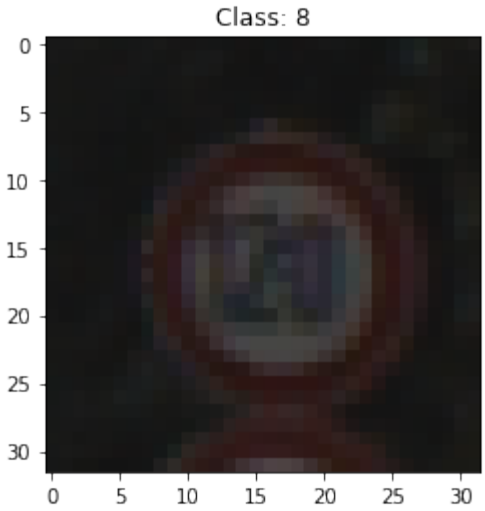

Class: 3
0
5
10
15
20
25
30
0
5
10
15
20
25
30

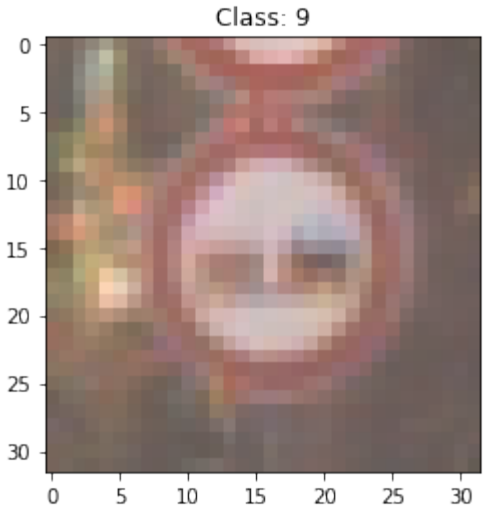

Class: 4
0
5
10
15
20
25
30
0
5
10
15
20
25
30

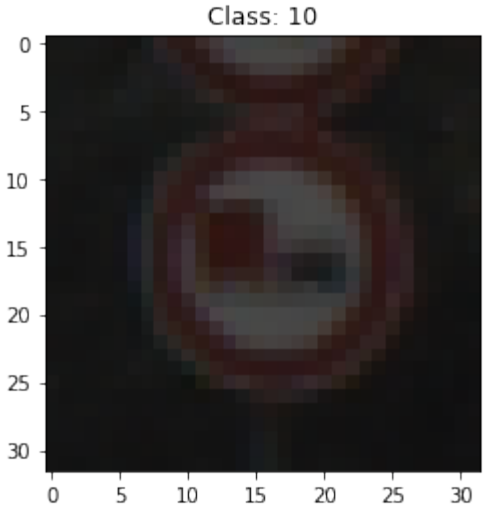

Class: 5
0
5
10
15
20
25
30
0
5
10
15
20
25
30

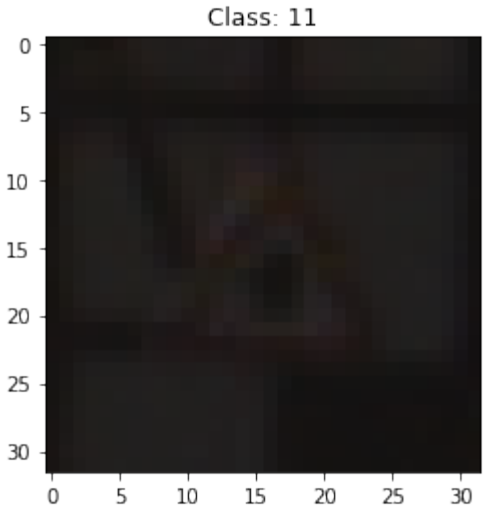

Class: 6

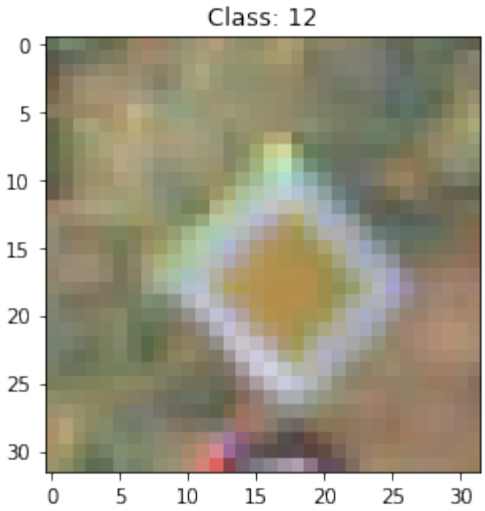

Class: 7

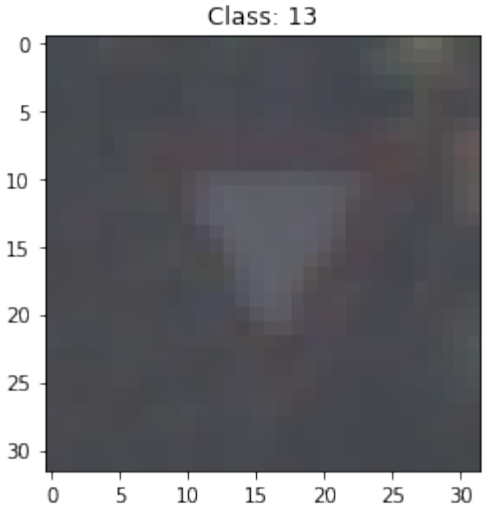

Class: 8

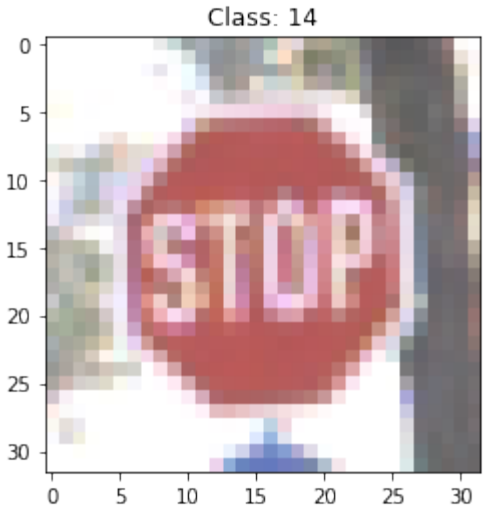

Class: 9
0
5
10
15
20
25
30
0
5
10
15
20
25
30

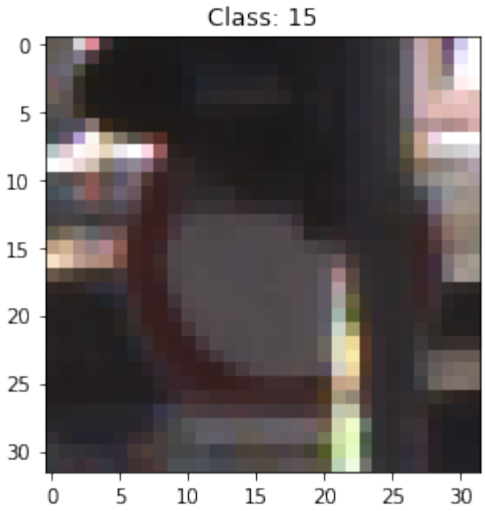

Class: 10
0
5
10
15
20
25
30
0
5
10
15
20
25
30

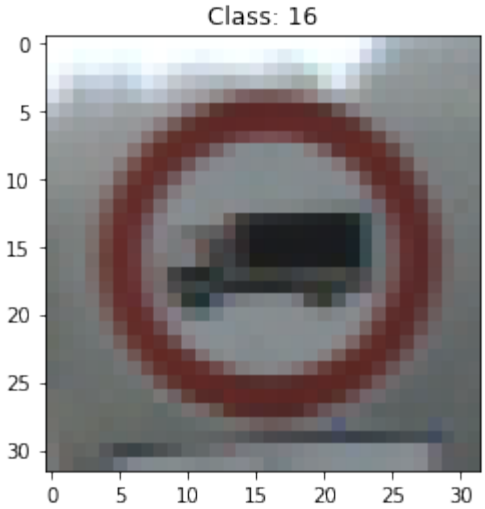

Class: 11
0
5
10
15
20
25
30
0
5
10
15
20
25
30

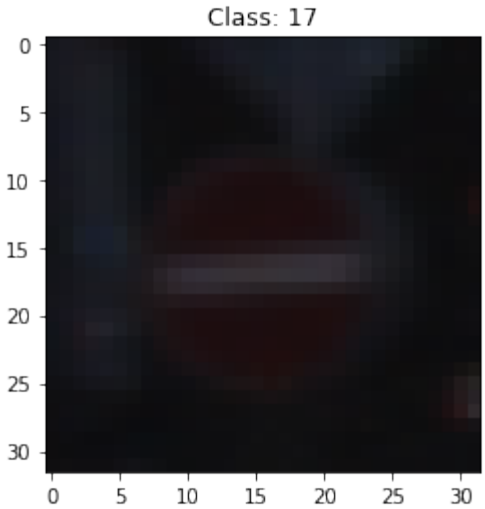

Class: 12
0
5
10
15
20
25
30
0
5
10
15
20
25
30

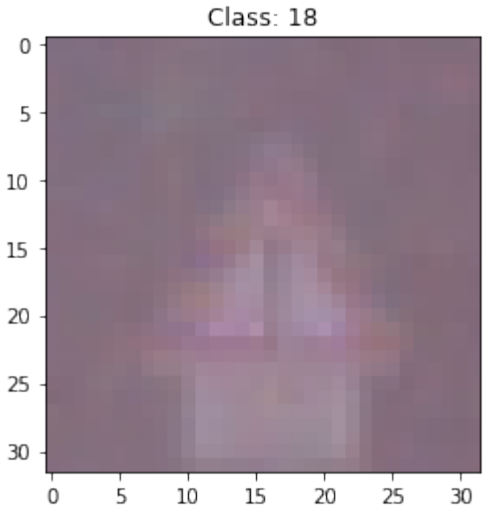

Class: 13
0
5
10
15
20
25
30
0
5
10
15
20
25
30

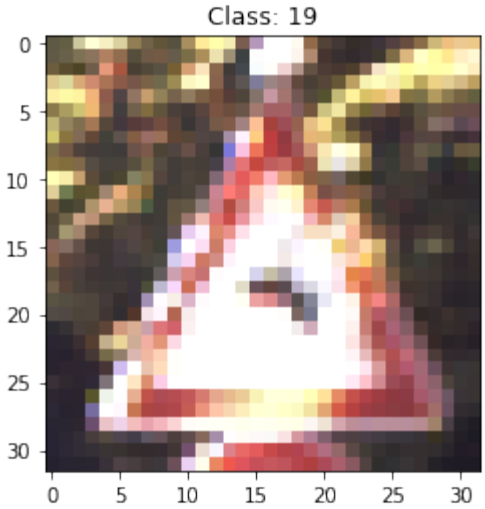

Class: 14
0
5
10
15
20
25
30
0
5
10
15
20
25
30
STOP

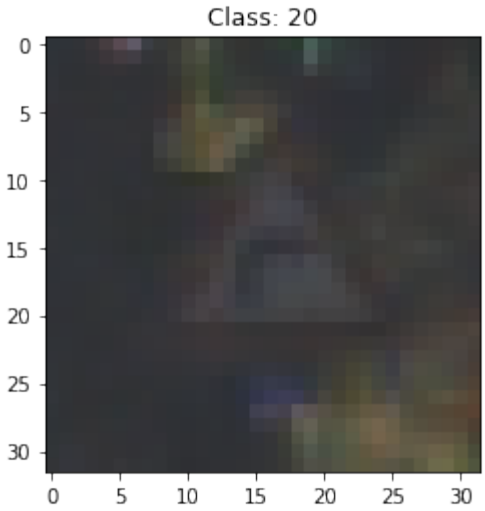

Class: 15
0
5
10
15
20
25
30
0
5
10
15
20
25
30

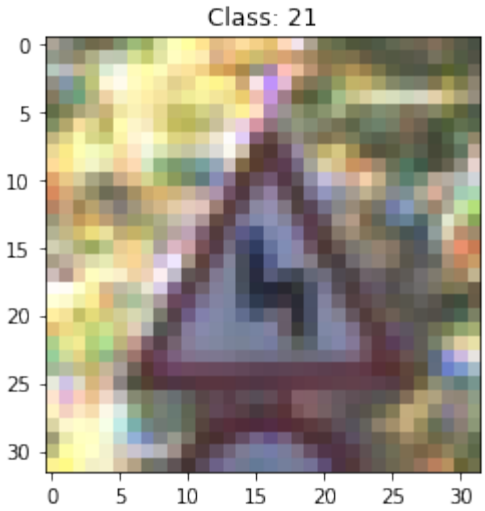

Class: 16
0
5
10
15
20
25
30
0
5
10
15
20
25
30

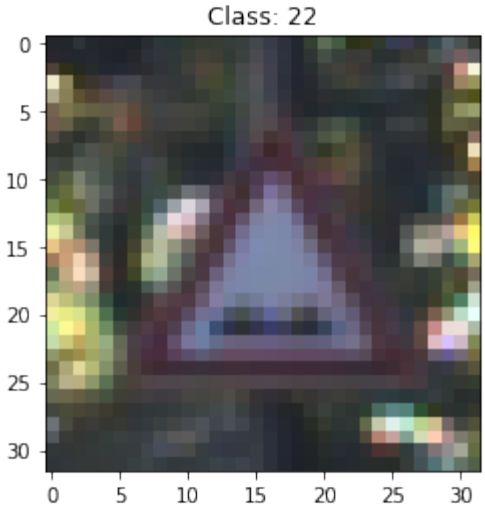

Class: 17
0
5
10
15
20
25
30
0
5
10
15
20
25
30

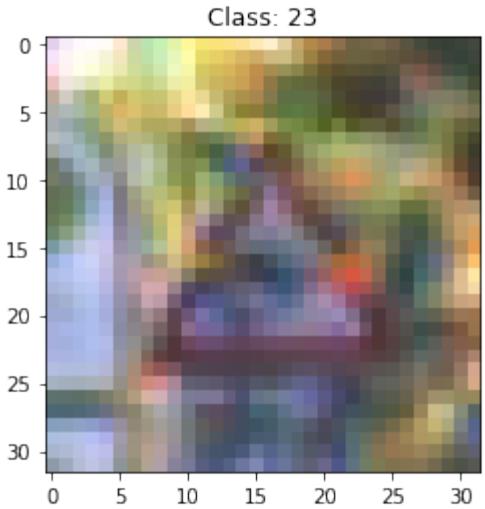

Class: 18

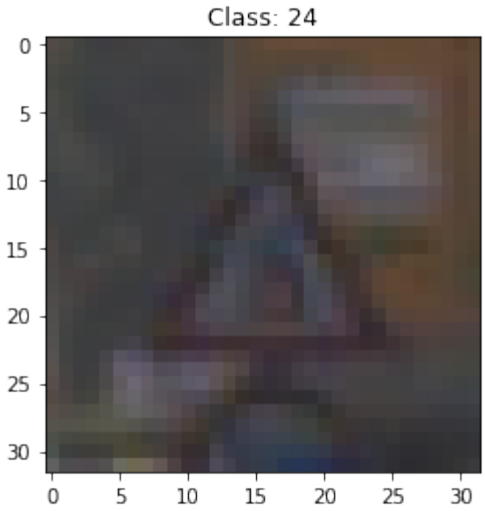

Class: 19

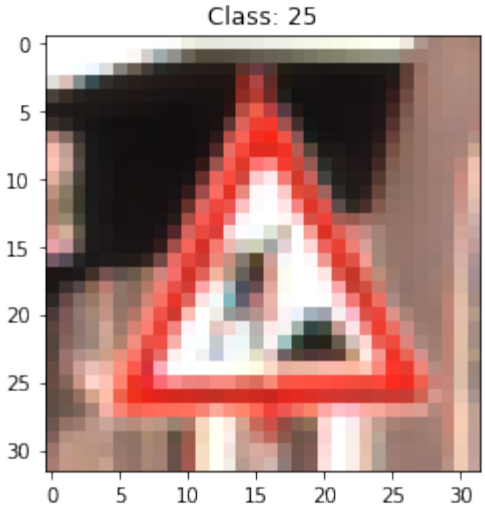

Class: 20

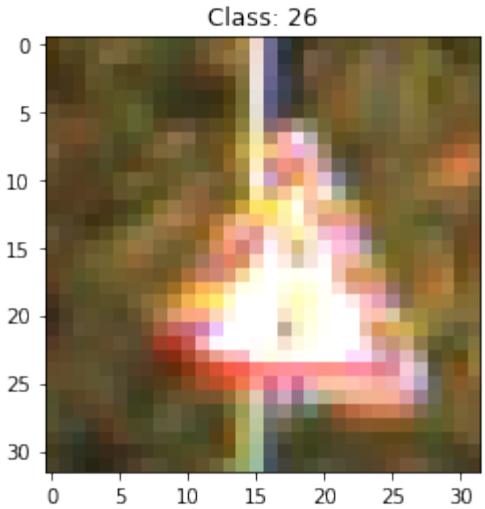

Class: 21
0
5
10
15
20
25
30
0
5
10
15
20
25
30

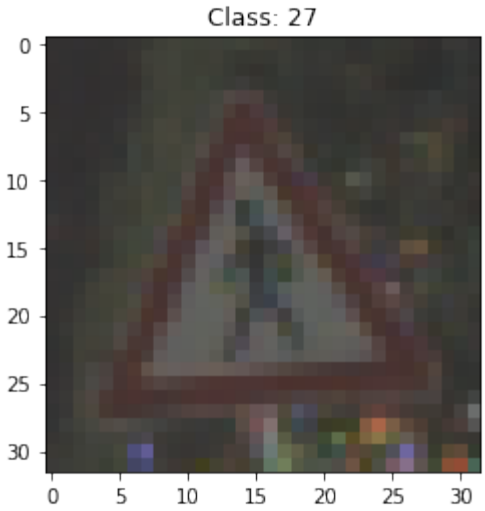

Class: 22
0
5
10
15
20
25
30
0
5
10
15
20
25
30

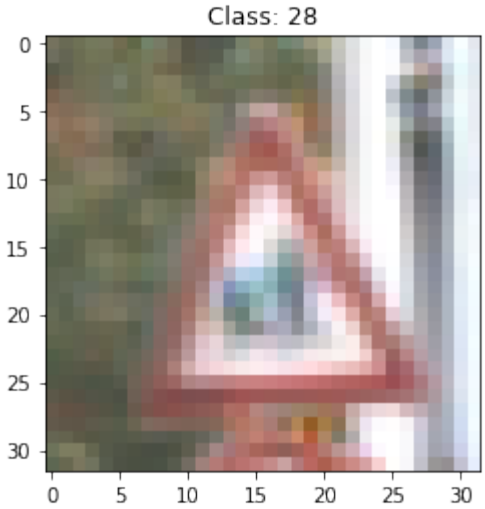

Class: 23
0
5
10
15
20
25
30
0
5
10
15
20
25
30

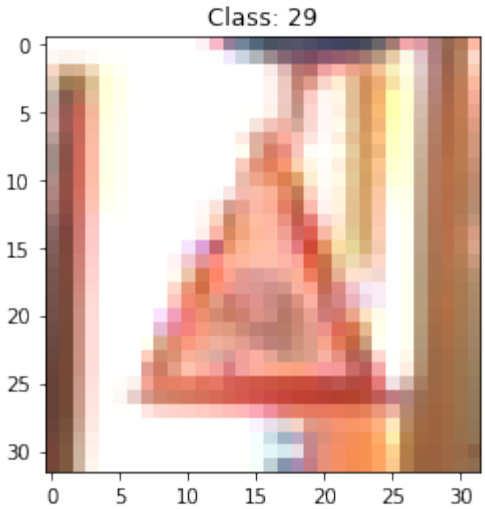

Class: 24
0
5
10
15
20
25
30
0
5
10
15
20
25
30

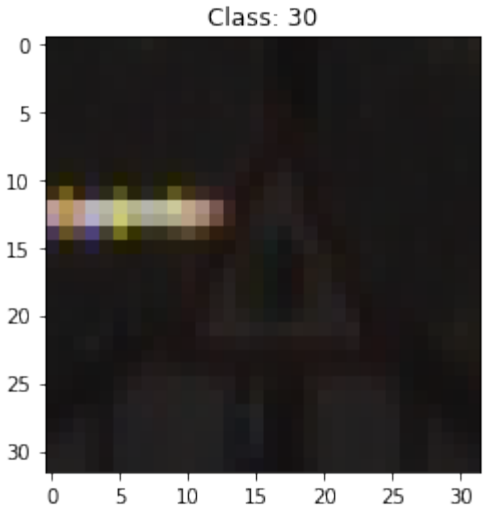

Class: 25
0
5
10
15
20
25
30
0
5
10
15
20
25
30

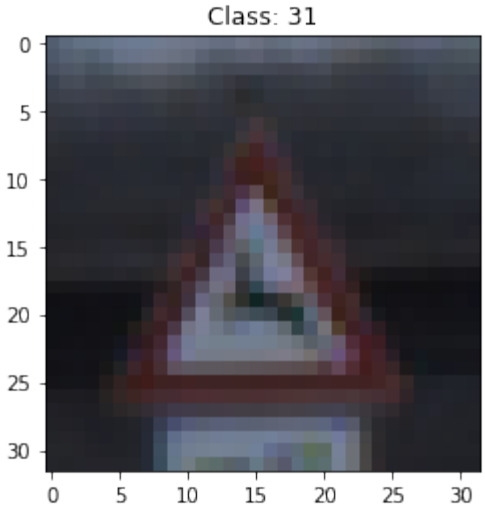

Class: 26
0
5
10
15
20
25
30
0
5
10
15
20
25
30

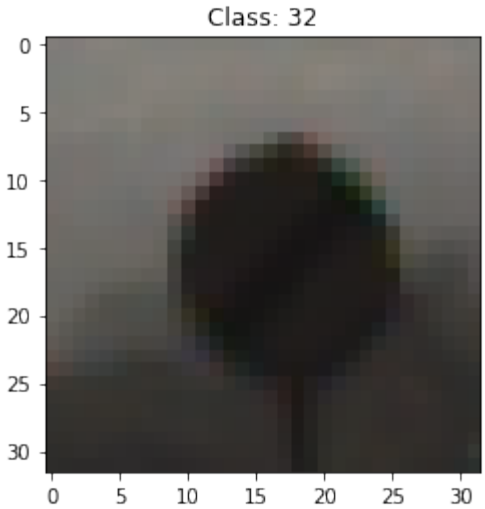

Class: 27
0
5
10
15
20
25
30
0
5
10
15
20
25
30

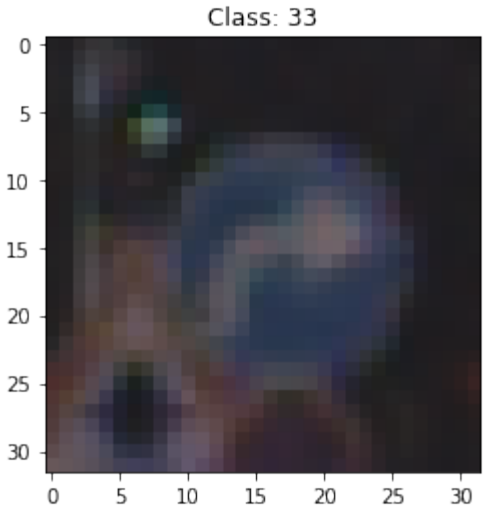

Class: 28
0
5
10
15
20
25
30
0
5
10
15
20
25
30

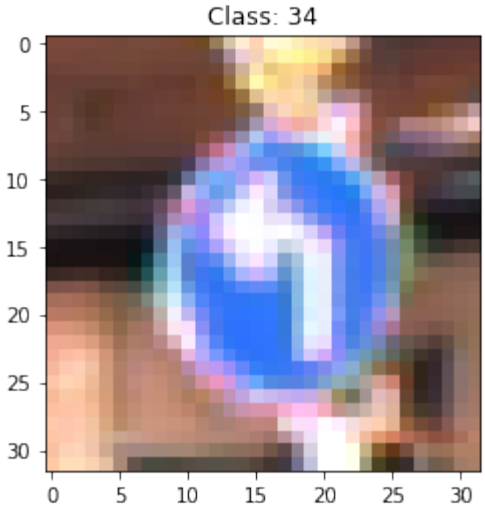

Class: 29
0
5
10
15
20
25
30
0
5
10
15
20
25
30

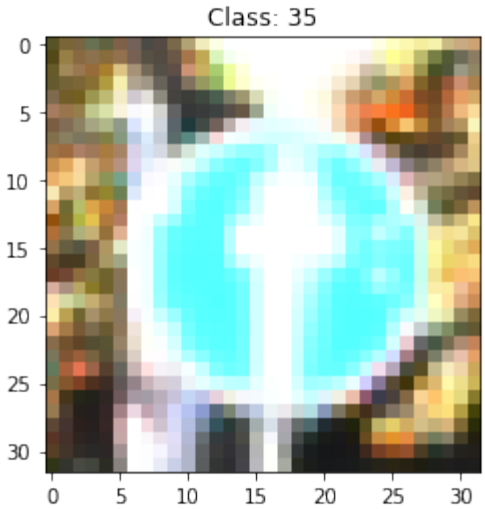

Class: 30
0
5
10
15
20
25
30
0
5
10
15
20
25
30

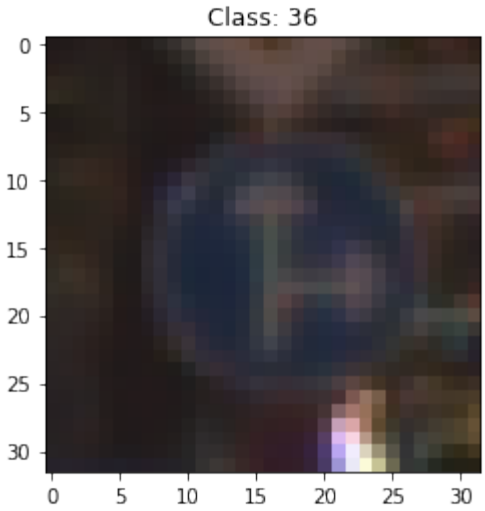

Class: 31
0
5
10
15
20
25
30
0
5
10
15
20
25
30

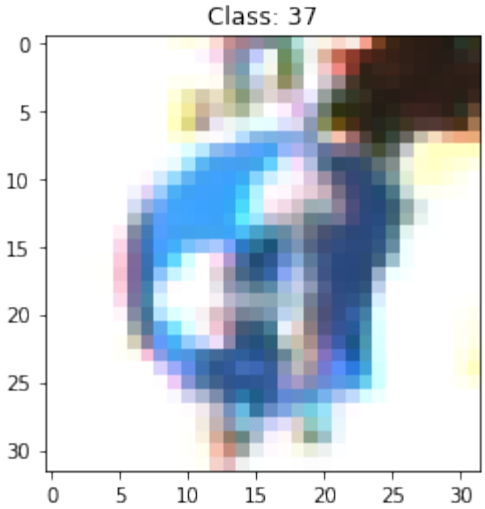

Class: 32
0
5
10
15
20
25
30
0
5
10
15
20
25
30

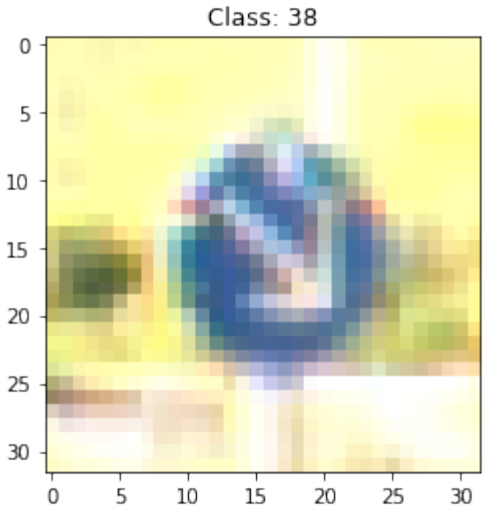

Class: 33

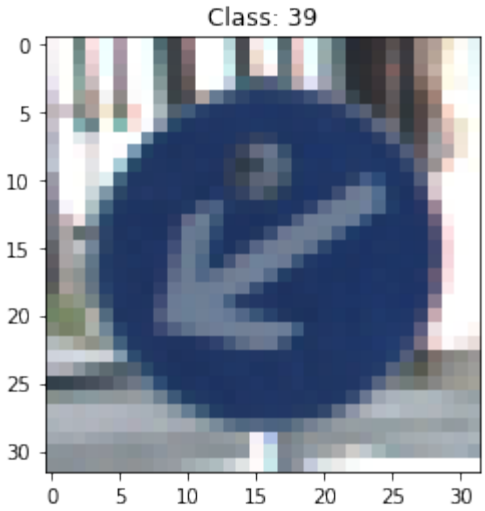

Class: 34

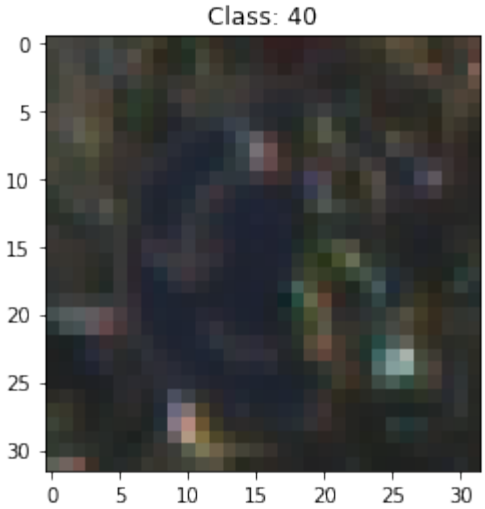

Class: 35

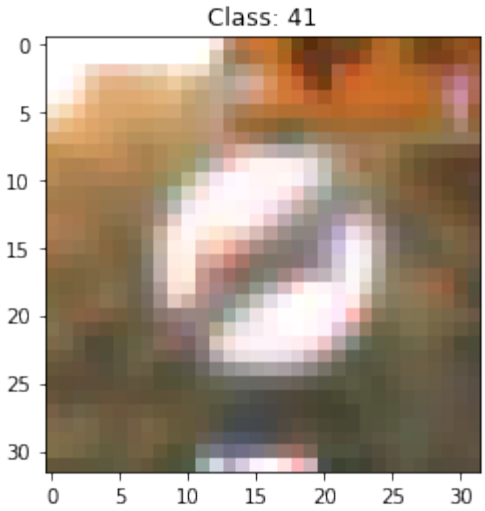

Class: 36
0
5
10
15
20
25
30
0
5
10
15
20
25
30

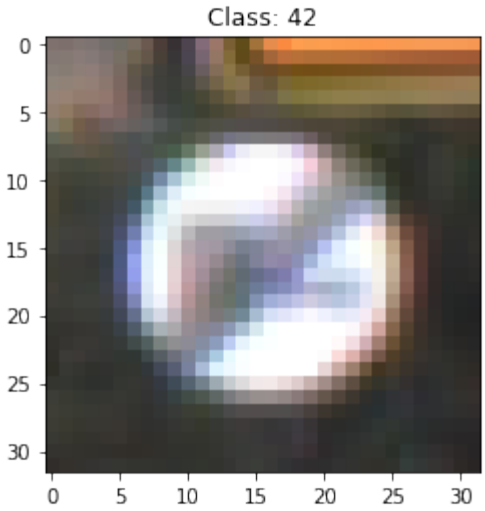

Class: 37
0
5
10
15
20
25
30
0
5
10
15
20
25
30


Class: 38
0
5
10
15
20
25
30
0
5
10
15
20
25
30


Class: 39
0
5
10
15
20
25
30
0
5
10
15
20
25
30


Class: 40
0
5
10
15
20
25
30
0
5
10
15
20
25
30


Class: 41
0
5
10
15
20
25
30
0
5
10
15
20
25
30



Add Extra Data to avoid OverFitting

In [9]:

```
def get_index_dict(y_train):
    # Returns indices of each label
    # Assumes that the labels are 0 to N-1
    dict_indices = {}
    ind_all = np.arange(len(y_train))

    for i in range(len(np.unique(y_train))):
        ind_i = ind_all[y_train == i]
        dict_indices[i] = ind_i
    return dict_indices

def transform_image(image,ang_range,shear_range,trans_range):

    # Rotation

    ang_rot = np.random.uniform(ang_range)-ang_range/2
    rows,cols,ch = image.shape
    Rot_M = cv2.getRotationMatrix2D((cols/2,rows/2),ang_rot,1)

    # Translation
    tr_x = trans_range*np.random.uniform()-trans_range/2
    tr_y = trans_range*np.random.uniform()-trans_range/2
    Trans_M = np.float32([[1,0,tr_x],[0,1,tr_y]])

    # Shear
    pts1 = np.float32([[5,5],[20,5],[5,20]])

    pt1 = 5+shear_range*np.random.uniform()-shear_range/2
    pt2 = 20+shear_range*np.random.uniform()-shear_range/2

    pts2 = np.float32([[pt1,5],[pt2,pt1],[5,pt2]])

    shear_M = cv2.getAffineTransform(pts1,pts2)

    image = cv2.warpAffine(image,Rot_M,(cols,rows))
    image = cv2.warpAffine(image,Trans_M,(cols,rows))
```

```
    image = cv2.warpAffine(image,shear_M,(cols,rows))

    return image

def
gen_extra_data(X_train,y_train,N_classes,n_each,ang_range,shear_range,trans_range,ran

    dict_indices = get_index_dict(y_train)
    n_class = len(np.unique(y_train))
    X_arr = []
    Y_arr = []
    n_train = len(X_train)
    for i in range(n_train):
        for i_n in range(n_each):
            img_trf = transform_image(X_train[i], ang_range,shear_range,trans_range)
            X_arr.append(img_trf)
            Y_arr.append(y_train[i])

    return X_arr,Y_arr


i_train =1
ang_rot = 10*0.9**(i_train)
trans_rot = 2*0.9**(i_train)
shear_rot = 2*0.9**(i_train)

#X_train,y_train =
gen_extra_data(X_train,y_train,43,5,ang_rot,trans_rot,shear_rot,1)


#print('Training Data')
#train_features = np.array(X_train)
#train_labels = np.array(y_train)

#inputs_per_class = np.bincount(train_labels)
#max_inputs = np.max(inputs_per_class)

#mpl_fig = plt.figure(figsize=(12,8))
#ax = mpl_fig.add_subplot(111)
#ax.set_ylabel('Inputs')
#ax.set_xlabel('Class')
#ax.set_title('Number of inputs per class')
#ax.bar(range(len(inputs_per_class)), inputs_per_class, 1/3, color='blue',
label='Inputs per class')
#plt.show()
```

Step 2: Design and Test a Model Architecture

Design and implement a deep learning model that learns to recognize traffic signs. Train and test your model on the German Traffic Sign Dataset.

There are various aspects to consider when thinking about this problem:

- Neural network architecture

- Play around preprocessing techniques (normalization, rgb to grayscale, etc)
- Number of examples per label (some have more than others).
- Generate fake data.

Here is an example of a published baseline model on this problem. It's not required to be familiar with the approach used in the paper but, it's good practice to try to read papers like these.

NOTE: The LeNet-5 implementation shown in the classroom at the end of the CNN lesson is a solid starting point. You'll have to change the number of classes and possibly the preprocessing, but aside from that it's plug and play!

Implementation

Use the code cell (or multiple code cells, if necessary) to implement the first step of your project. Once you have completed your implementation and are satisfied with the results, be sure to thoroughly answer the questions that follow.

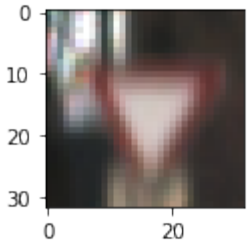
2.1 Preprocessing

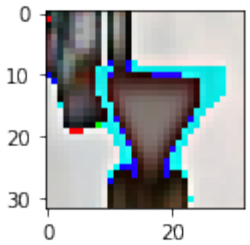
In [10]:

```
### Step 1
### Images are already resized the images to 32x32. so no padding required



### Step 2
### its important to shuffle data because order of data could effect how well is
network gets trained
X_train, y_train = shuffle(X_train, y_train, random_state=42)



### Step 3 Normalisation (not standardising)
# The goal is to independently normalize each feature component to the [0,1] range
# In image processing, normalization is a process that changes the range of pixel
intensity values.
# In stochastic gradient descent, Normalization can sometimes improve the
convergence speed of the algorithm

X_train_org = X_train
X_test_org = X_test
X_train = (X_train - X_train.mean()) / (np.max(X_train) - np.min(X_train))
X_test = (X_test - X_test.mean()) / (np.max(X_test) - np.min(X_test))
print('')
print('---Before and after Normalisation Sample---')
image_index = 2
plt.subplot(2,2,1)
plt.imshow(X_train_org[image_index])
plt.show()
plt.subplot(2,2,2)
plt.imshow(X_train[image_index])
plt.show()
```

```
### Step 4 Segmenting data into training, test, and validation
X_train, X_validation, y_train, y_validation = train_test_split(X_train, y_train,
test_size=0.2, random_state=42)
y_validation_OneHot = OHE_Encode(y_validation,n_classes)
OHE_Validate(y_validation_OneHot,y_validation)



print('')
print('---Data shape---')
print('Validation data shape', X_validation.shape)
print('Training data shape', X_train.shape)
print('Testing data shape', X_test.shape)

print('')
print('---Label shape---')
print('Validation Label data shape', y_validation.shape)
print('Training Label data shape', y_train.shape)
print('Testing Label data shape', y_test.shape)
```

```
---Before and after Normalisation Sample---
```






```
One hot encoding Validated

---Data shape---
Validation data shape (7842, 32, 32, 3)
Training data shape (31367, 32, 32, 3)
Testing data shape (12630, 32, 32, 3)

---Label shape---
Validation Label data shape (7842,)
Training Label data shape (31367,)
Testing Label data shape (12630,)
```

Step 3: Define Network

3.1 Network parameters

In [11]:

```
n_filters = 32
kernel_size = (3, 3)
n_fc1 = 512
n_fc2 = 128
pool_size = 2 # i.e. (2,2)

dropout_conv = 0.9
dropout_fc = 0.9

weights_stddev = 0.1
weights_mean = 0.0
biases_mean = 0.0

padding = 'VALID'
if padding == 'SAME':
    conv_output_length = 6
elif padding == 'VALID':
    conv_output_length = 5
else:
    raiseException("Unknown padding.")
```

3.2 tf Graph input

In [12]:

```
x_unflattened = tf.placeholder("float", [None,
image_shape[0],image_shape[1],image_shape[2]])
x = x_unflattened

y_rawlabels = tf.placeholder("int32", [None])
y = tf.one_hot(y_rawlabels, depth=n_classes, on_value=1., off_value=0., axis=-1)
```

3.3 Model

In [13]:

```
def weight_variable(shape, weight_mean, weight_stddev):
    return tf.Variable(tf.truncated_normal(shape, stddev=weight_stddev,
mean=weight_mean))

def bias_variable(shape, bias_mean):
    return tf.Variable(tf.constant(bias_mean, shape=shape))

def conv2d(x, W, b, strides=3):
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)

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

```
padding=padding_setting)

def plot_metric_per_epoch(accuracies):
    x_epochs = []
    y_epochs = []
    for i, val in enumerate(accuracies):
        x_epochs.append(i)
        y_epochs.append(val)

    plt.figure(figsize=(15,8))
    plt.xlabel('epoch')
    plt.ylabel('score')
    plt.title('Score per epoch')
    #plt.legend()
    plt.grid()
    plt.scatter(x_epochs, y_epochs,s=50,c='lightgreen', marker='s', label='score')
    plt.show()
```

In [14]:

```
weights = {
    'conv1': weight_variable([kernel_size[0], kernel_size[1], n_channels,
n_filters], weights_mean, weights_stddev),
    'fc1': weight_variable([n_filters * conv_output_length**2, n_fc1], weights_mean,
weights_stddev),
    'fc2': weight_variable([n_fc1, n_fc2], weights_mean, weights_stddev),
    'out': weight_variable([n_fc2, n_classes], weights_mean, weights_stddev)
}

biases = {
    'conv1': bias_variable([n_filters], biases_mean),
    'fc1': bias_variable([n_fc1], biases_mean),
    'fc2': bias_variable([n_fc2], biases_mean),
    'out': bias_variable([n_classes], biases_mean)
}
```

In [15]:

```
def conv_net(model_x, model_weights, model_biases, model_pool_size,
             model_dropout_conv, model_dropout_fc, padding='SAME'):

    # Convolution Layer 1
    conv1 = conv2d(model_x, model_weights['conv1'], model_biases['conv1'])

    # Max Pooling (down-sampling)
    conv1 = maxpool2d(conv1, k=model_pool_size, padding_setting=padding)
    conv1 = tf.nn.dropout(conv1, model_dropout_conv)

    # Reshape conv1 output to fit fully connected layer input
    conv1_shape = conv1.get_shape().as_list()
    fc1 = tf.reshape(conv1, [-1, conv1_shape[1]*conv1_shape[2]*conv1_shape[3]])

    # Fully connected layer 1
    fc1 = tf.add(tf.matmul(fc1, model_weights['fc1']), model_biases['fc1'])
    fc1 = tf.nn.elu(fc1)
    fc1 = tf.nn.dropout(fc1, model_dropout_fc)

    # Fully connected layer 2
    fc2 = tf.add(tf.matmul(fc1, model_weights['fc2']), model_biases['fc2'])
    fc2 = tf.nn.elu(fc2)
```

```
    fc2 = tf.nn.dropout(fc2, model_dropout_fc)

    # Output layer
    output = tf.add(tf.matmul(fc2, model_weights['out']), model_biases['out'])

    return output
```

Step 4: Train Model

4.1 Training Parameters

In [16]:

```
#learning Rate
#It represents the step that is taken in a gradient decent algorithm to find an
optimal solution.
#If steps is too big, algorithm can go from peak to peak, and skip lows altogether.
#if steps is too small, algorithm will take lot of time to converge.
#To train a model, it is often recommended to lower the learning rate as the
training progresses.
#We will use exponential_decay function to do so.
#It requires 'initial learning rate' and 'global_step' value to compute the decayed
learning rate.
learning_rate = 0.001
initial_learning_rate = learning_rate
annealing_rate = 1

#epochs number
#this represents the number of times that we will run our main "for" loop. this is
the loop that we will run to provide
#data to our algorithms for training and testing.
#this is not number of batches
training_epochs = 150

#Batch Size
#Deep learning algorithms are iterative in the sense that they load samples in
batches to avoid running out of memory.
#number of batch (bin) = number of training samples / batch size.
batch_size = 100

display_step = 1
anneal_mod_frequency = 15
print_accuracy_mod_frequency = 1
```

4.2 Model/Loss/Optimizer

In [17]:

```
# Construct model
pred = conv_net(x, weights, biases, pool_size, dropout_conv, dropout_fc,
padding=padding)
pred_probs = tf.nn.softmax(pred)
```

```
# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(pred, y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
```

4.3 Train

In [18]:

```
# Function to initialise the variables
init = tf.global_variables_initializer()

# Launch the graph
sess = tf.Session()

# Initialise variables
sess.run(init)

# Initialise time logs
init_time = time.time()
epoch_time = init_time

five_epoch_moving_average = 0.
epoch_accuracies = []
print_accuracy_mod_frequency = 1
total_batch = int(len(X_train) / batch_size)

# Training cycle
for epoch in range(training_epochs):
    if five_epoch_moving_average > 0.96:
        break

    for i in range(total_batch):
        batch_x, batch_y = np.array(X_train[i * batch_size:(i + 1) * batch_size]), \
                           np.array(y_train[i * batch_size:(i + 1) * batch_size])
        _, epoch_cost = sess.run([optimizer, cost], feed_dict={x_unflattened:
batch_x, y_rawlabels: batch_y})

    # Display logs per epoch step
    if epoch % display_step == 0:
        last_epoch_time = epoch_time
        epoch_time = time.time()
        print("Epoch:", '%04d' % (epoch + 1), "cost:","{:.9f}".format(epoch_cost),
"Time since last epoch: ", epoch_time - last_epoch_time)

    # Anneal learning rate
    if (epoch + 1) % anneal_mod_frequency == 0:
        learning_rate *= annealing_rate
        print("New learning rate: ", learning_rate)

    if (epoch + 1) % print_accuracy_mod_frequency == 0:
        correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
        # Line below needed only when not using `with tf.Session() as sess`
        with sess.as_default():
            epoch_accuracy = accuracy.eval({x_unflattened: X_validation,
y_rawlabels: y_validation})
            epoch_accuracies.append(epoch_accuracy)
```

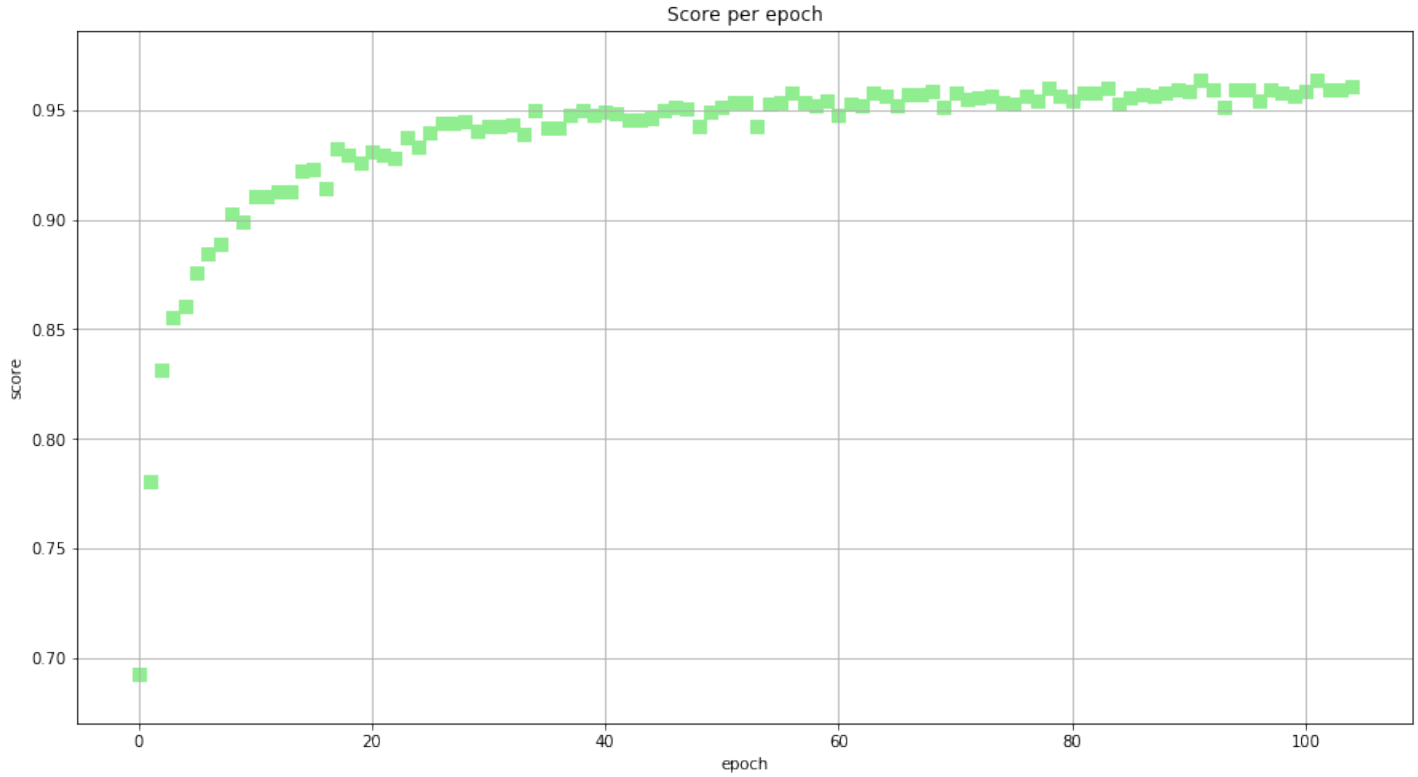
```
            ji = (len(epoch_accuracies), 5)[len(epoch_accuracies)>4]
            five_epoch_moving_average =
np.sum(epoch_accuracies[epoch+1-ji:epoch+1])/ji
            print("epoch Accuracy (validation) = " + str(epoch_accuracy) + ",
five_epoch_moving_average = " + "{:.6f}".format(five_epoch_moving_average))


print("Optimization Finished!")
plot_metric_per_epoch(epoch_accuracies)
```

```
Epoch: 0001 cost: 0.894962311 Time since last epoch:  9.886565685272217
epoch Accuracy (validation) = 0.692808, five_epoch_moving_average = 0.692808
Epoch: 0002 cost: 0.513390124 Time since last epoch:  11.493657350540161
epoch Accuracy (validation) = 0.780796, five_epoch_moving_average = 0.736802
Epoch: 0003 cost: 0.471693128 Time since last epoch:  10.93262529373169
epoch Accuracy (validation) = 0.831038, five_epoch_moving_average = 0.768214
Epoch: 0004 cost: 0.333910704 Time since last epoch:  10.555603742599487
epoch Accuracy (validation) = 0.855522, five_epoch_moving_average = 0.790041
Epoch: 0005 cost: 0.254200816 Time since last epoch:  11.060632705688477
epoch Accuracy (validation) = 0.860495, five_epoch_moving_average = 0.804132
Epoch: 0006 cost: 0.305052042 Time since last epoch:  10.865621328353882
epoch Accuracy (validation) = 0.875797, five_epoch_moving_average = 0.840729
Epoch: 0007 cost: 0.190831408 Time since last epoch:  10.988628625869751
epoch Accuracy (validation) = 0.884468, five_epoch_moving_average = 0.861464
Epoch: 0008 cost: 0.217563510 Time since last epoch:  11.728670835494995
epoch Accuracy (validation) = 0.888804, five_epoch_moving_average = 0.873017
Epoch: 0009 cost: 0.222239792 Time since last epoch:  10.98762845993042
epoch Accuracy (validation) = 0.902703, five_epoch_moving_average = 0.882453
Epoch: 0010 cost: 0.159320474 Time since last epoch:  12.082690954208374
epoch Accuracy (validation) = 0.899133, five_epoch_moving_average = 0.890181
Epoch: 0011 cost: 0.159388766 Time since last epoch:  11.278645038604736
epoch Accuracy (validation) = 0.910227, five_epoch_moving_average = 0.897067
Epoch: 0012 cost: 0.081200264 Time since last epoch:  11.33664846420288
epoch Accuracy (validation) = 0.910482, five_epoch_moving_average = 0.902270
Epoch: 0013 cost: 0.149247259 Time since last epoch:  10.742614507675171
epoch Accuracy (validation) = 0.912777, five_epoch_moving_average = 0.907065
Epoch: 0014 cost: 0.096895695 Time since last epoch:  10.91262412071228
epoch Accuracy (validation) = 0.912777, five_epoch_moving_average = 0.909079
Epoch: 0015 cost: 0.142583758 Time since last epoch:  10.819618940353394
New learning rate:  0.001
epoch Accuracy (validation) = 0.922469, five_epoch_moving_average = 0.913747
Epoch: 0016 cost: 0.098467723 Time since last epoch:  11.094634532928467
epoch Accuracy (validation) = 0.922979, five_epoch_moving_average = 0.916297
Epoch: 0017 cost: 0.076421507 Time since last epoch:  10.678610801696777
epoch Accuracy (validation) = 0.914308, five_epoch_moving_average = 0.917062
Epoch: 0018 cost: 0.064703993 Time since last epoch:  10.969627380371094
epoch Accuracy (validation) = 0.932033, five_epoch_moving_average = 0.920913
Epoch: 0019 cost: 0.080188505 Time since last epoch:  10.711612701416016
epoch Accuracy (validation) = 0.92961, five_epoch_moving_average = 0.924279
Epoch: 0020 cost: 0.050790042 Time since last epoch:  10.613606929779053
epoch Accuracy (validation) = 0.926167, five_epoch_moving_average = 0.925019
Epoch: 0021 cost: 0.135333493 Time since last epoch:  11.087634325027466
epoch Accuracy (validation) = 0.930757, five_epoch_moving_average = 0.926575
Epoch: 0022 cost: 0.040186983 Time since last epoch:  10.688611268997192
epoch Accuracy (validation) = 0.92961, five_epoch_moving_average = 0.929635
Epoch: 0023 cost: 0.052411109 Time since last epoch:  10.692611694335938
epoch Accuracy (validation) = 0.928335, five_epoch_moving_average = 0.928896
Epoch: 0024 cost: 0.105956800 Time since last epoch:  11.231642484664917
epoch Accuracy (validation) = 0.937388, five_epoch_moving_average = 0.930451
Epoch: 0025 cost: 0.110308446 Time since last epoch:  11.707669496536255
```

```
epoch Accuracy (validation) = 0.932925, five_epoch_moving_average = 0.931803
Epoch: 0026 cost: 0.074984819 Time since last epoch:  11.015630006790161
epoch Accuracy (validation) = 0.939811, five_epoch_moving_average = 0.933614
Epoch: 0027 cost: 0.094455145 Time since last epoch:  11.245643377304077
epoch Accuracy (validation) = 0.943637, five_epoch_moving_average = 0.936419
Epoch: 0028 cost: 0.103398502 Time since last epoch:  10.91262412071228
epoch Accuracy (validation) = 0.943637, five_epoch_moving_average = 0.939480
Epoch: 0029 cost: 0.076381795 Time since last epoch:  10.97462773323059
epoch Accuracy (validation) = 0.944529, five_epoch_moving_average = 0.940908
Epoch: 0030 cost: 0.083059616 Time since last epoch:  10.780616521835327
New learning rate:  0.001
epoch Accuracy (validation) = 0.940449, five_epoch_moving_average = 0.942413
Epoch: 0031 cost: 0.035957664 Time since last epoch:  10.972627639770508
epoch Accuracy (validation) = 0.942234, five_epoch_moving_average = 0.942897
Epoch: 0032 cost: 0.059762359 Time since last epoch:  13.119750499725342
epoch Accuracy (validation) = 0.942617, five_epoch_moving_average = 0.942693
Epoch: 0033 cost: 0.036077321 Time since last epoch:  13.887794256210327
epoch Accuracy (validation) = 0.943254, five_epoch_moving_average = 0.942617
Epoch: 0034 cost: 0.061254472 Time since last epoch:  17.69301199913025
epoch Accuracy (validation) = 0.939174, five_epoch_moving_average = 0.941545
Epoch: 0035 cost: 0.052312776 Time since last epoch:  16.64895224571228
epoch Accuracy (validation) = 0.949758, five_epoch_moving_average = 0.943407
Epoch: 0036 cost: 0.128661171 Time since last epoch:  16.925968170166016
epoch Accuracy (validation) = 0.942107, five_epoch_moving_average = 0.943382
Epoch: 0037 cost: 0.061073698 Time since last epoch:  15.813904523849487
epoch Accuracy (validation) = 0.941597, five_epoch_moving_average = 0.943178
Epoch: 0038 cost: 0.035135083 Time since last epoch:  13.623779058456421
epoch Accuracy (validation) = 0.947335, five_epoch_moving_average = 0.943994
Epoch: 0039 cost: 0.064237848 Time since last epoch:  16.19192624092102
epoch Accuracy (validation) = 0.949758, five_epoch_moving_average = 0.946111
Epoch: 0040 cost: 0.018748429 Time since last epoch:  17.558004140853882
epoch Accuracy (validation) = 0.947717, five_epoch_moving_average = 0.945703
Epoch: 0041 cost: 0.021935560 Time since last epoch:  15.67189645767212
epoch Accuracy (validation) = 0.948993, five_epoch_moving_average = 0.947080
Epoch: 0042 cost: 0.077892579 Time since last epoch:  18.522059440612793
epoch Accuracy (validation) = 0.948227, five_epoch_moving_average = 0.948406
Epoch: 0043 cost: 0.050296325 Time since last epoch:  14.93485426902771
epoch Accuracy (validation) = 0.945167, five_epoch_moving_average = 0.947972
Epoch: 0044 cost: 0.080875359 Time since last epoch:  17.28298830986023
epoch Accuracy (validation) = 0.94555, five_epoch_moving_average = 0.947131
Epoch: 0045 cost: 0.023909841 Time since last epoch:  18.636066198349
New learning rate:  0.001
epoch Accuracy (validation) = 0.945932, five_epoch_moving_average = 0.946774
Epoch: 0046 cost: 0.011977810 Time since last epoch:  17.705012559890747
epoch Accuracy (validation) = 0.950013, five_epoch_moving_average = 0.946978
Epoch: 0047 cost: 0.035918012 Time since last epoch:  16.86696481704712
epoch Accuracy (validation) = 0.951033, five_epoch_moving_average = 0.947539
Epoch: 0048 cost: 0.030735292 Time since last epoch:  15.067861795425415
epoch Accuracy (validation) = 0.950778, five_epoch_moving_average = 0.948661
Epoch: 0049 cost: 0.075161323 Time since last epoch:  14.397823333740234
epoch Accuracy (validation) = 0.942617, five_epoch_moving_average = 0.948075
Epoch: 0050 cost: 0.219219357 Time since last epoch:  15.411881685256958
epoch Accuracy (validation) = 0.948738, five_epoch_moving_average = 0.948635
Epoch: 0051 cost: 0.160118863 Time since last epoch:  13.764787197113037
epoch Accuracy (validation) = 0.951543, five_epoch_moving_average = 0.948942
Epoch: 0052 cost: 0.025373461 Time since last epoch:  13.86979341506958
epoch Accuracy (validation) = 0.953456, five_epoch_moving_average = 0.949426
Epoch: 0053 cost: 0.142234445 Time since last epoch:  18.92808246612549
epoch Accuracy (validation) = 0.953583, five_epoch_moving_average = 0.949987
Epoch: 0054 cost: 0.006270137 Time since last epoch:  14.187811613082886
```

```
epoch Accuracy (validation) = 0.942489, five_epoch_moving_average = 0.949962
Epoch: 0055 cost: 0.013027457 Time since last epoch:  13.016744613647461
epoch Accuracy (validation) = 0.952563, five_epoch_moving_average = 0.950727
Epoch: 0056 cost: 0.007717252 Time since last epoch:  18.856078386306763
epoch Accuracy (validation) = 0.953456, five_epoch_moving_average = 0.951109
Epoch: 0057 cost: 0.013510614 Time since last epoch:  13.645780563354492
epoch Accuracy (validation) = 0.957791, five_epoch_moving_average = 0.951976
Epoch: 0058 cost: 0.053378772 Time since last epoch:  12.731728076934814
epoch Accuracy (validation) = 0.953583, five_epoch_moving_average = 0.951976
Epoch: 0059 cost: 0.013266009 Time since last epoch:  14.061804294586182
epoch Accuracy (validation) = 0.95167, five_epoch_moving_average = 0.953813
Epoch: 0060 cost: 0.006460203 Time since last epoch:  17.682011365890503
New learning rate:  0.001
epoch Accuracy (validation) = 0.953966, five_epoch_moving_average = 0.954093
Epoch: 0061 cost: 0.028124794 Time since last epoch:  16.862964630126953
epoch Accuracy (validation) = 0.94759, five_epoch_moving_average = 0.952920
Epoch: 0062 cost: 0.008214556 Time since last epoch:  14.188811540603638
epoch Accuracy (validation) = 0.952436, five_epoch_moving_average = 0.951849
Epoch: 0063 cost: 0.029965699 Time since last epoch:  17.841020345687866
epoch Accuracy (validation) = 0.952308, five_epoch_moving_average = 0.951594
Epoch: 0064 cost: 0.021308823 Time since last epoch:  18.544060707092285
epoch Accuracy (validation) = 0.957919, five_epoch_moving_average = 0.952844
Epoch: 0065 cost: 0.097276971 Time since last epoch:  13.874793529510498
epoch Accuracy (validation) = 0.956134, five_epoch_moving_average = 0.953277
Epoch: 0066 cost: 0.023973737 Time since last epoch:  16.21792769432068
epoch Accuracy (validation) = 0.952181, five_epoch_moving_average = 0.954195
Epoch: 0067 cost: 0.003900653 Time since last epoch:  14.923853635787964
epoch Accuracy (validation) = 0.957281, five_epoch_moving_average = 0.955165
Epoch: 0068 cost: 0.032515284 Time since last epoch:  16.268930435180664
epoch Accuracy (validation) = 0.956899, five_epoch_moving_average = 0.956083
Epoch: 0069 cost: 0.037574463 Time since last epoch:  15.240871667861938
epoch Accuracy (validation) = 0.958812, five_epoch_moving_average = 0.956261
Epoch: 0070 cost: 0.027110701 Time since last epoch:  15.586891651153564
epoch Accuracy (validation) = 0.951288, five_epoch_moving_average = 0.955292
Epoch: 0071 cost: 0.005643809 Time since last epoch:  13.558775424957275
epoch Accuracy (validation) = 0.957791, five_epoch_moving_average = 0.956414
Epoch: 0072 cost: 0.018305833 Time since last epoch:  16.64995241165161
epoch Accuracy (validation) = 0.955113, five_epoch_moving_average = 0.955981
Epoch: 0073 cost: 0.008626533 Time since last epoch:  19.33310580253601
epoch Accuracy (validation) = 0.955496, five_epoch_moving_average = 0.955700
Epoch: 0074 cost: 0.006228790 Time since last epoch:  17.943026304244995
epoch Accuracy (validation) = 0.956134, five_epoch_moving_average = 0.955165
Epoch: 0075 cost: 0.005411087 Time since last epoch:  16.28593134880066
New learning rate:  0.001
epoch Accuracy (validation) = 0.953583, five_epoch_moving_average = 0.955624
Epoch: 0076 cost: 0.033696488 Time since last epoch:  13.833791255950928
epoch Accuracy (validation) = 0.952818, five_epoch_moving_average = 0.954629
Epoch: 0077 cost: 0.000915433 Time since last epoch:  15.055861234664917
epoch Accuracy (validation) = 0.956516, five_epoch_moving_average = 0.954909
Epoch: 0078 cost: 0.063065045 Time since last epoch:  17.31099009513855
epoch Accuracy (validation) = 0.954093, five_epoch_moving_average = 0.954629
Epoch: 0079 cost: 0.024593325 Time since last epoch:  17.63200855255127
epoch Accuracy (validation) = 0.959959, five_epoch_moving_average = 0.955394
Epoch: 0080 cost: 0.004226239 Time since last epoch:  18.67706823348999
epoch Accuracy (validation) = 0.956134, five_epoch_moving_average = 0.955904
Epoch: 0081 cost: 0.006296419 Time since last epoch:  17.80301833152771
epoch Accuracy (validation) = 0.954221, five_epoch_moving_average = 0.956185
Epoch: 0082 cost: 0.012124037 Time since last epoch:  14.225813627243042
epoch Accuracy (validation) = 0.957409, five_epoch_moving_average = 0.956363
Epoch: 0083 cost: 0.012157260 Time since last epoch:  12.511715650558472
```

```
epoch Accuracy (validation) = 0.957919, five_epoch_moving_average = 0.957128
Epoch: 0084 cost: 0.007729578 Time since last epoch:  13.89679479598999
epoch Accuracy (validation) = 0.960087, five_epoch_moving_average = 0.957154
Epoch: 0085 cost: 0.010298887 Time since last epoch:  19.049089431762695
epoch Accuracy (validation) = 0.952691, five_epoch_moving_average = 0.956465
Epoch: 0086 cost: 0.039411224 Time since last epoch:  14.343820571899414
epoch Accuracy (validation) = 0.955751, five_epoch_moving_average = 0.956771
Epoch: 0087 cost: 0.003744549 Time since last epoch:  19.541117668151855
epoch Accuracy (validation) = 0.956771, five_epoch_moving_average = 0.956644
Epoch: 0088 cost: 0.033027902 Time since last epoch:  16.803961038589478
epoch Accuracy (validation) = 0.956261, five_epoch_moving_average = 0.956312
Epoch: 0089 cost: 0.006019471 Time since last epoch:  16.569947719573975
epoch Accuracy (validation) = 0.957919, five_epoch_moving_average = 0.955879
Epoch: 0090 cost: 0.022889182 Time since last epoch:  14.480828285217285
New learning rate:  0.001
epoch Accuracy (validation) = 0.959194, five_epoch_moving_average = 0.957179
Epoch: 0091 cost: 0.026669960 Time since last epoch:  13.201755285263062
epoch Accuracy (validation) = 0.958174, five_epoch_moving_average = 0.957664
Epoch: 0092 cost: 0.082925037 Time since last epoch:  14.450826406478882
epoch Accuracy (validation) = 0.963657, five_epoch_moving_average = 0.959041
Epoch: 0093 cost: 0.002972201 Time since last epoch:  13.651780843734741
epoch Accuracy (validation) = 0.959577, five_epoch_moving_average = 0.959704
Epoch: 0094 cost: 0.014854210 Time since last epoch:  13.542774677276611
epoch Accuracy (validation) = 0.951543, five_epoch_moving_average = 0.958429
Epoch: 0095 cost: 0.005970881 Time since last epoch:  16.94896936416626
epoch Accuracy (validation) = 0.959449, five_epoch_moving_average = 0.958480
Epoch: 0096 cost: 0.009269524 Time since last epoch:  17.97902822494507
epoch Accuracy (validation) = 0.959067, five_epoch_moving_average = 0.958659
Epoch: 0097 cost: 0.067570381 Time since last epoch:  18.33704900741577
epoch Accuracy (validation) = 0.954093, five_epoch_moving_average = 0.956746
Epoch: 0098 cost: 0.003921504 Time since last epoch:  19.589120388031006
epoch Accuracy (validation) = 0.959194, five_epoch_moving_average = 0.956669
Epoch: 0099 cost: 0.010301650 Time since last epoch:  15.751900911331177
epoch Accuracy (validation) = 0.957919, five_epoch_moving_average = 0.957944
Epoch: 0100 cost: 0.015973015 Time since last epoch:  17.355992794036865
epoch Accuracy (validation) = 0.956134, five_epoch_moving_average = 0.957281
Epoch: 0101 cost: 0.118168965 Time since last epoch:  15.991914510726929
epoch Accuracy (validation) = 0.958812, five_epoch_moving_average = 0.957230
Epoch: 0102 cost: 0.069796160 Time since last epoch:  18.433054447174072
epoch Accuracy (validation) = 0.963402, five_epoch_moving_average = 0.959092
Epoch: 0103 cost: 0.093538344 Time since last epoch:  19.242100477218628
epoch Accuracy (validation) = 0.959067, five_epoch_moving_average = 0.959067
Epoch: 0104 cost: 0.011780025 Time since last epoch:  20.184154510498047
epoch Accuracy (validation) = 0.958939, five_epoch_moving_average = 0.959271
Epoch: 0105 cost: 0.006619848 Time since last epoch:  20.11715054512024
New learning rate:  0.001
epoch Accuracy (validation) = 0.960342, five_epoch_moving_average = 0.960112
Optimization Finished!
```



4.4 Test Model

In [19]:

```
# Test model
correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

with sess.as_default():
    print("Accuracy (test):", accuracy.eval({x_unflattened: X_test, y_rawlabels:
y_test}))

# Print parameters for reference
print("\nParameters:")
print("Learning rate (initial): ", initial_learning_rate)
print("Anneal learning rate every ", anneal_mod_frequency, " epochs by ", 1 -
annealing_rate)
print("Learning rate (final): ", learning_rate)
print("Training epochs: ", training_epochs)
print("Batch size: ", batch_size)
print("Dropout (conv): ", dropout_conv)
print("Dropout (fc): ", dropout_fc)
print("Padding: ", padding)
print("weights_mean: ", weights_mean)
print("weights_stddev: ", weights_stddev)
print("biases_mean: ", biases_mean)
```

```
Accuracy (test): 0.795249

Parameters:
Learning rate (initial):  0.001
Anneal learning rate every  15  epochs by  0
```

```
Learning rate (final):  0.001
Training epochs:  150
Batch size:  100
Dropout (conv):  0.9
Dropout (fc):  0.9
Padding:  VALID
weights_mean:  0.0
weights_stddev:  0.1
biases_mean:  0.0
```

In [ ]:

```
```

In [ ]:

```
```

Question 1

Describe how you preprocessed the data. Why did you choose that technique?

Answer:

See Section 2.1 for Preprocessing steps

1 // no Padding done as image was 32x32

2 // data normallized to avoid high variance and improve classifier performance

3 // features randomized to avoid overfitting

4 // Added Extra data for low frequency Sign classes

5 // labels one hot encoded

Question 2

Describe how you set up the training, validation and testing data for your model. ***Optional****: If you generated additional data, how did you generate the data? Why did you generate the data? What are the differences in the new dataset (with generated data) from the original dataset?*

Answer:

See Section 2.1 for Splitting Data

1 // Training and test data were already separated (downloaded pickled files train.p and test.p).

2 // shuffled the training data because they were arranged in ascending order by label. If I don't shuffle the training data, the first series of batches will all be the first type of sign followed by the second type and so on. This will distort the learning process.

3 // I further split the training data into test and valdiation sets so the model wouldn't be cheating when we optimised it.

4 // Improvement: Generate additional data. The number of examples in the training data for each class is uneven, so the model may be biased towards predicting an unknown sign belongs to a class where there is abundant training data since we are minimising the training loss.

Question 3

What does your final architecture look like? (Type of model, layers, sizes, connectivity, etc.) For reference on how to build a deep neural network using TensorFlow, see Deep Neural Network in TensorFlow from the classroom.

Answer:

See 3.3 for Model / Network Defination

- 3-layer Convolutional Neural Network.
- It consists of one convolution layer (feature extraction) followed by two fully connected layers (ReLU activation) and a single fully connected linear classifier.

  1 : Convolution layer Input: (32, 32, 3)
  Output: (5, 5, 32) 'VALID' padding Filters: 32 Stride: 3 ReLU Activation 2D Max Pooling (down-sampling) layer Dropout: 0.9

  2 : Reshape Layer Input: (5, 5, 32) Output: 800

  3 : Fully connected layer
  Input: 800 Output: 512 ReLU Activation Dropout: 0.9

  4 : Fully connected layer
  Input: 512 Output: 128 ReLU Activation Dropout: 0.9

  5 : output layer
  Input: 128 Output: 43
- The network uses full colour information (all three channels) and normalised data.

Question 4

How did you train your model? (Type of optimizer, batch size, epochs, hyperparameters, etc.)

See Section 4.3

Type of optimiser: AdamOptimizer
- Batch size: 100
- Training Epochs: 105(with ELU)
- Learning rate: 0.001

Network Parameters:

- Dropout (conv layer): 0.9
- Dropout (fully connected layers): 0.9
- Padding: VALID

tf.train.AdamOptimizer ref http://stats.stackexchange.com/questions/184448/difference-between-gradientdescentoptimizer-and-adamoptimizer-tensorflow

•Main advantage of Adam over the simple tf.train.GradientDescentOptimizer: Uses moving averages of the parameters (momentum) -> enables Adam to use a larger effective step size, and the algorithm will converge to this step size without fine tuning. A simple tf.train.GradientDescentOptimizer would require more hyperparameter tuning before it would converge as quickly.

•Disadvantage: Adam requires more computation to be performed for each parameter in each training step (to maintain the moving averages and variance, and calculate the scaled gradient) and more state to be retained for each parameter (approximately tripling the size of the model to store the average and variance for each parameter).

Question 5

What approach did you take in coming up with a solution to this problem? It may have been a process of trial and error, in which case, outline the steps you took to get to the final solution and why you chose those steps. Perhaps your solution involved an already well known implementation or architecture. In this case, discuss why you think this is suitable for the current problem.

Answer:

1. First attempt: building a minimum viable model and debugging

- I wanted to get a working model first. I started with a basic multilayer perceptron which I adapted from `TensorFlow-Examples`. I trained it for 15 epochs, which had an accuracy of 6% on the training and test sets. I then trained a two-layer convolutional neural network for 15 epochs which had an accuracy of 5-6% on the training and test sets.
  - The accuracy was lower than I expected and the cost seemed high (of order 10^6 in the first epoch, 10^5 in the second and third and in the hundreds in the tenth epoch), so I adjusted parameters hoping to improve it before training for longer.
  - The cost reduced significantly (to single digits by the second epoch as opposed to order 10^5) after I added a small positive bias to the initial weights and biases. Strangely, the accuracy did not increase, but remained at 5-6%. The cost did not decrease significantly over the next 10 epochs either.
  - I went on Slack to see what results people were getting to get a feel for how wrong I was. I saw that people often trained their networks for hundreds of epochs so I thought it would be good to train my network for e.g. 100 epochs.

- I rewrote the multilayer perceptron in a Python Script and it worked fine, returning an accuracy of over 70% accuracy within 2 epochs.

2. Improvements to the model

- I then added a convolution layer before the two fully connected layers and the output layer.
- This new model produced a validation accuracy of above 90% after 15 epochs (parameters not tuned), which was higher than that for the two-layer feedforward network. So I chose this model with a convolution layer.

3. Tuning Parameters

- I altered the model code (replaced hard-coded numbers with variables) so I could tweak parameters easily.
- I tested models with different values or settings for
  - dropout for the fully connected layers,
  - dropout for the convolution layer,
  - padding (SAME vs VALID),
  - weight and bias initialisation
  - maxpool vs no max pool
- I used Keras to implement comparisons so I could get full figures on training and validation loss and accuracy easily.
- I stopped when the model reached a validation accuracy of over 95% within 100 epochs.
  - This figure is strange because my models implemented in Keras reach validation accuracy of over 99% within 15 epochs.

Step 3: Test a Model on New Images

Take several pictures of traffic signs that you find on the web or around you (at least five), and run them through your classifier on your computer to produce example results. The classifier might not recognize some local signs but it could prove interesting nonetheless.

You may find `signnames.csv` useful as it contains mappings from the class id (integer) to the actual sign name.

Implementation

Use the code cell (or multiple code cells, if necessary) to implement the first step of your project. Once you have completed your implementation and are satisfied with the results, be sure to thoroughly answer the questions that follow.

In [24]:

```
def read_image_and_print_dims(image_path):
    image = mpimg.imread(image_path)
    print('This image is:', type(image), 'with dimensions:', image.shape)
    plt.imshow(image)
    return image
```

```
def process_image_file(name):
    image = cv2.imread(name)
    image = cv2.resize(image,(32,32))
    image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
    image = image/255.-.5
    return image

def plot_image_grid(n_row,n_col,X):
    plt.figure(figsize = (8,6))
    gs1 = gridspec.GridSpec(n_row,n_row)
    gs1.update(wspace=0.01, hspace=0.02) # set the spacing between axes.

    for i in range(n_row*n_col):
        # i = i + 1 # grid spec indexes from 0
        ax1 = plt.subplot(gs1[i])
        plt.axis('on')
        ax1.set_xticklabels([])
        ax1.set_yticklabels([])
        ax1.set_aspect('equal')
        #plt.subplot(4,11,i+1)
        ind_plot = i
        plt.imshow(X[ind_plot])
        plt.axis('off')
    plt.show()

def predict(img):
    classification = sess.run(tf.argmax(pred, 1), feed_dict={x_unflattened: [img]})
    print('NN predicted', classification[0])
    plt.imshow(sign_dict[classification[0]])
    plt.show()

def top_5_predictions(img):
    #Return model's top five choices for what traffic sign this image is and its
confidence in its predictions.
    top_five_certainties = sess.run(tf.nn.top_k(tf.nn.softmax(pred), k=5),
feed_dict={x_unflattened: [img]})
    print("Top five: ", top_five_certainties)
    plot_certainty_arrays(top_five_certainties[0][0], top_five_certainties[1][0])
    return top_five_certainties

def plot_certainty_arrays(probabilities, labels):
    # Plot model's probabilities (y) and traffic sign labels (x) in a bar chart.
    y_pos = np.arange(len(labels))
    plt.bar(y_pos, probabilities, align='center', alpha=0.5)
    plt.xticks(y_pos, labels)
    plt.ylabel('Probability')
    plt.xlabel('Traffic sign')
    plt.title('Model\'s certainty of its predictions')
    plt.show()
    print("Traffic Sign Key")
    for label in labels:
        print(label, ": ", data_pd.loc[label]['SignName'])
```

Question 6

Choose five candidate images of traffic signs and provide them in the report. Are there any particular qualities of the image(s) that might make classification difficult? It could be helpful to plot the images in the notebook.

Answer:

In [21]:

```
newdata = [process_image_file("./new_signs/"+name) for name in
os.listdir("./new_signs/")]
namenewdata = [name for name in os.listdir("./new_signs/")]
newdata = np.array(newdata ,dtype = np.float32)
plot_image_grid(9,5,newdata+.5)
```

Question 7

Is your model able to perform equally well on captured pictures when compared to testing on the dataset? The simplest way to do this check the accuracy of the predictions. For example, if the model predicted 1 out of 5 signs correctly, it's 20% accurate.

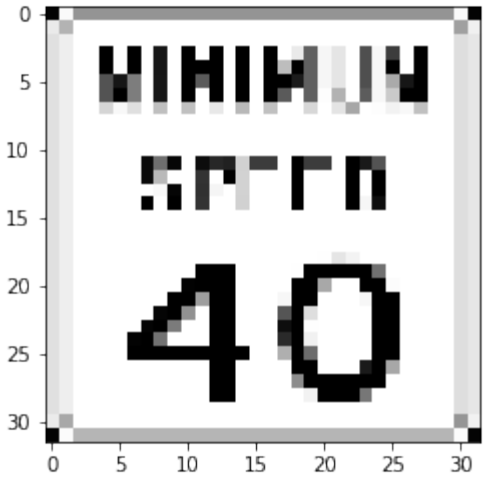
***NOTE:** You could check the accuracy manually by using `signnames.csv` (same directory). This file has a mapping from the class id (0-42) to the corresponding sign name. So, you could take the class id the model outputs, lookup the name in `signnames.csv` and see if it matches the sign from the image.*

Answer:

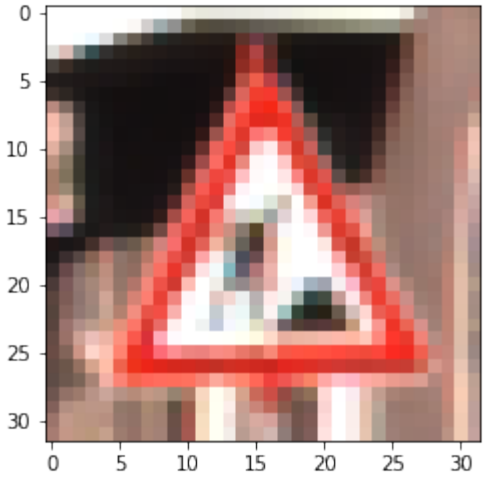
In [22]:

```
for i in range(len(namenewdata)):
    print('\n\n\n\n')
    print(namenewdata[i])
    plt.imshow(newdata[i]+.5)
    plt.show()
    predict(newdata[i])
```

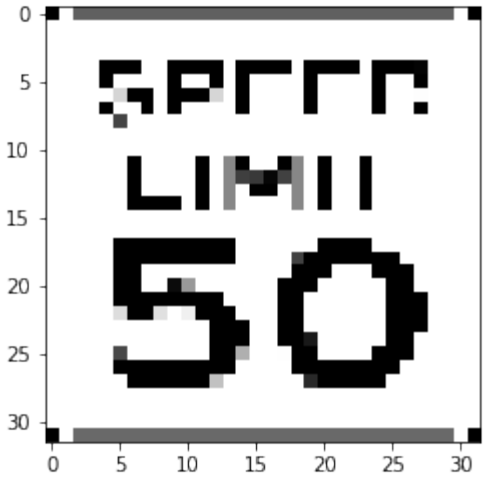
```
40mph.png
```



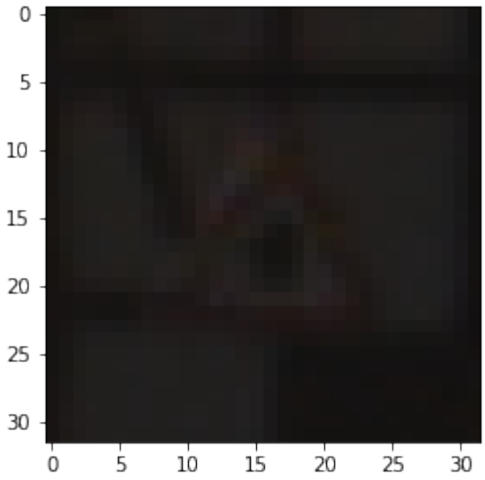
NN predicted 25



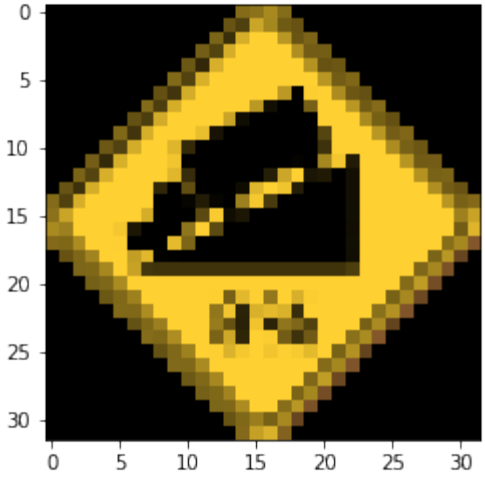
50mph.png



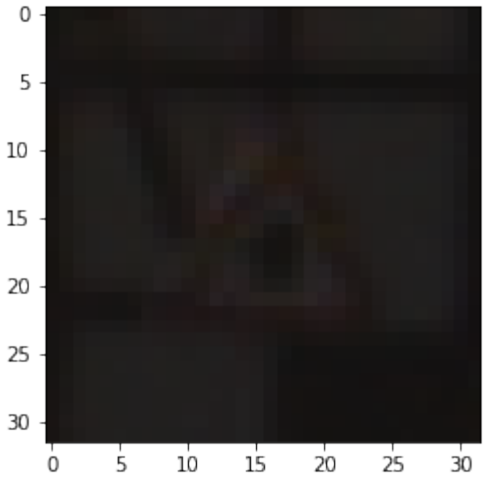
NN predicted 11



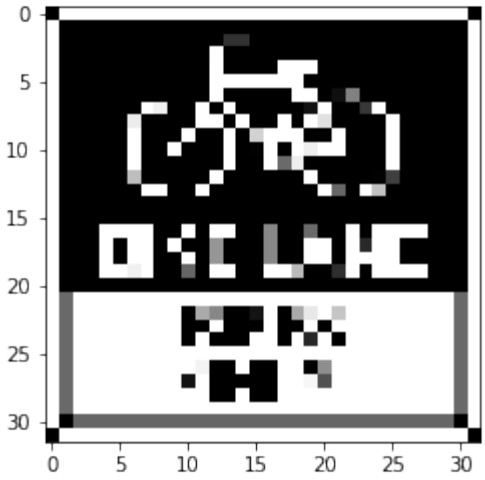
8pcGrade.png



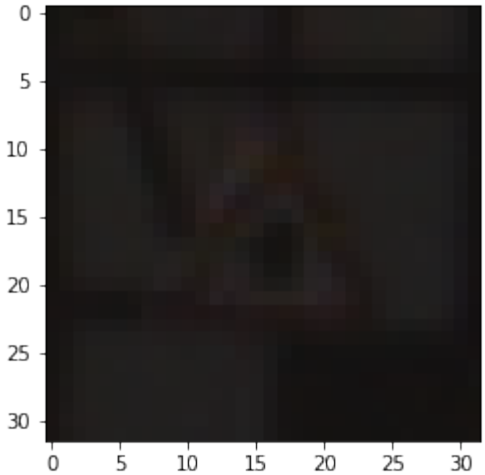
NN predicted 11



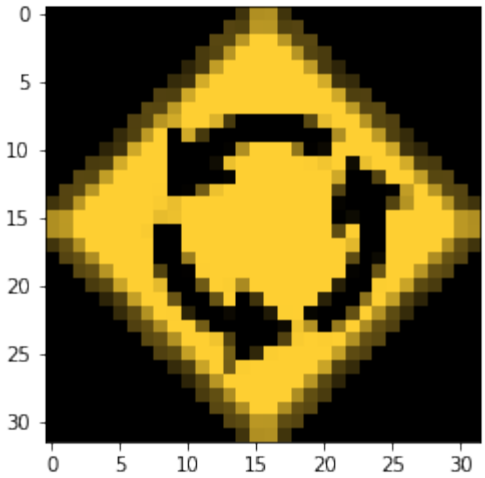
bikelane.png



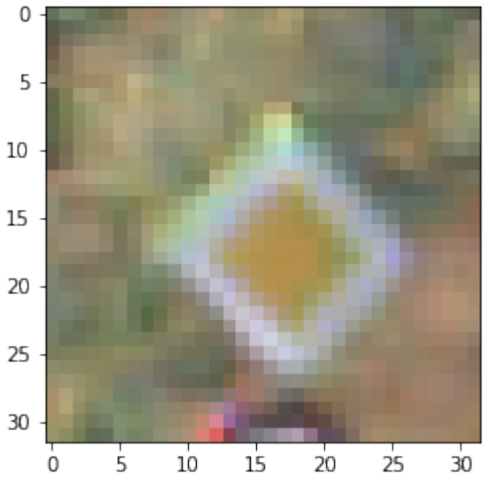
NN predicted 11



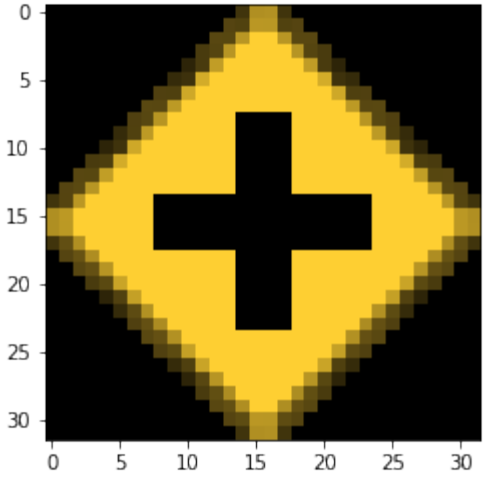
circle.png



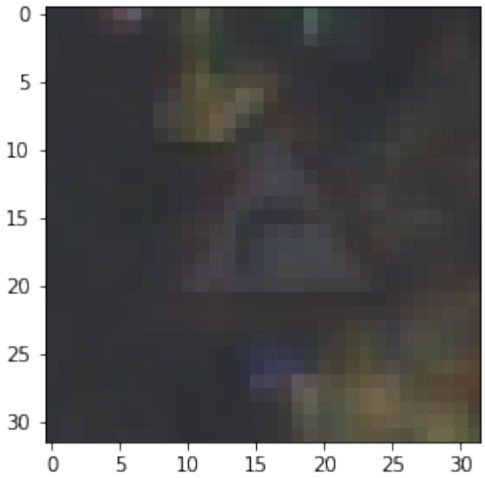
NN predicted 12



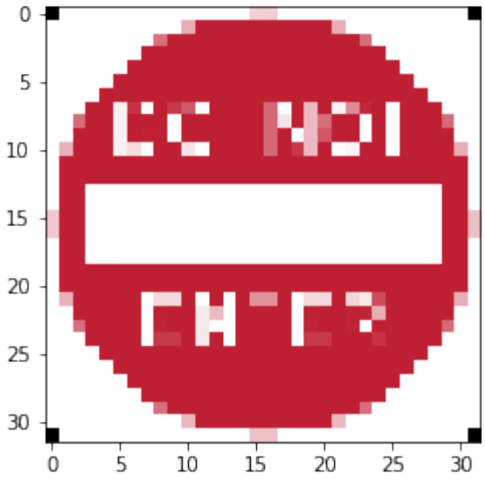
CrossRoad.png



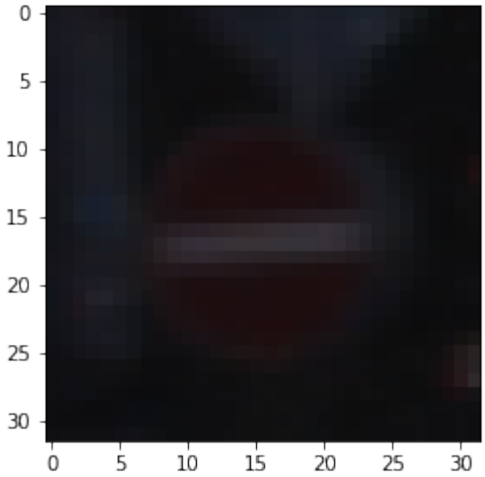
NN predicted 20



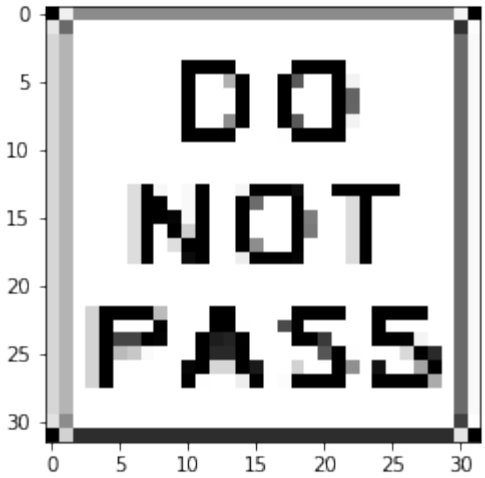
DonotEnter.png



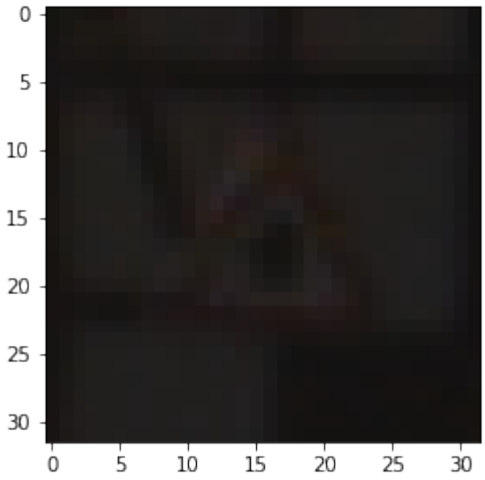
NN predicted 17



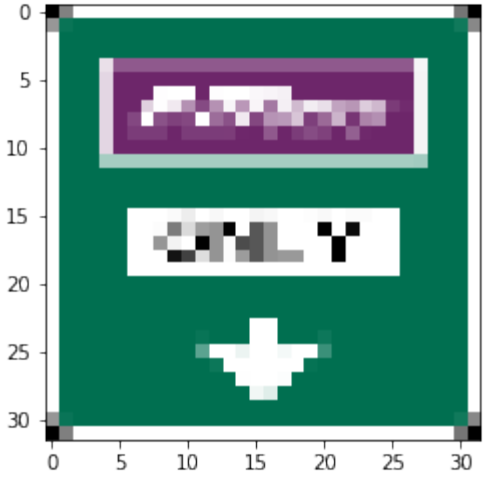
DoNotPass.png



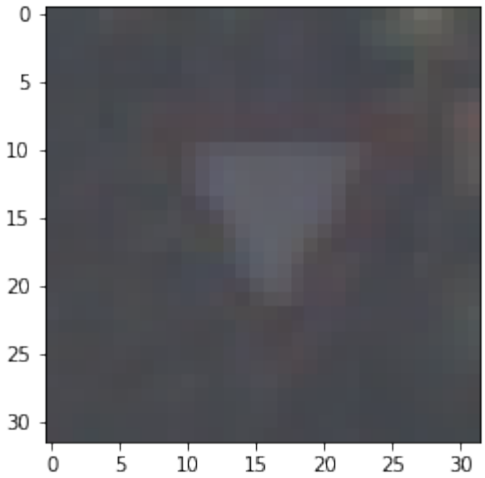
NN predicted 11



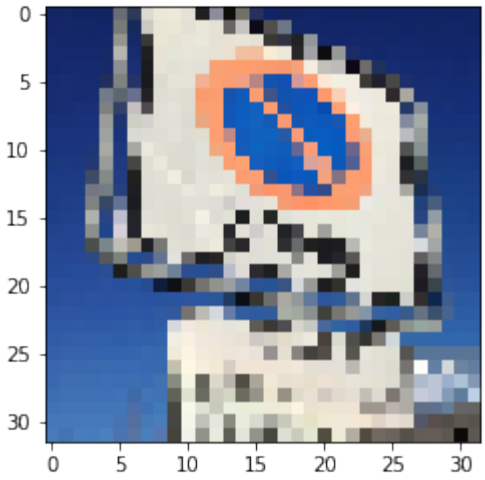
EZPass.png



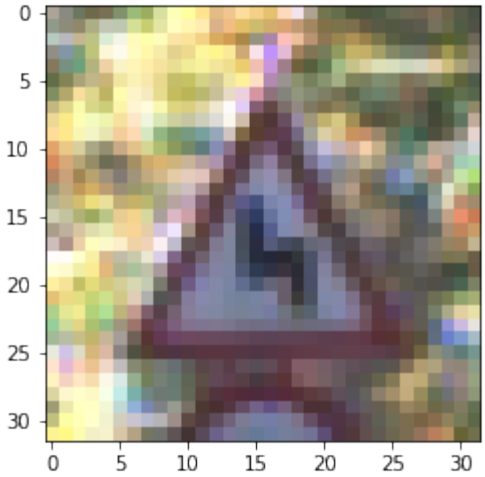
NN predicted 13



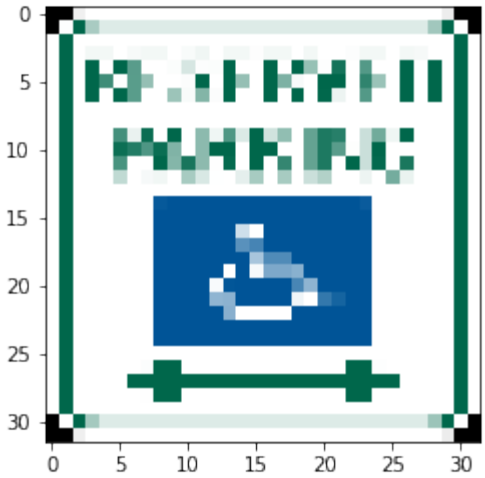
german_sign.png



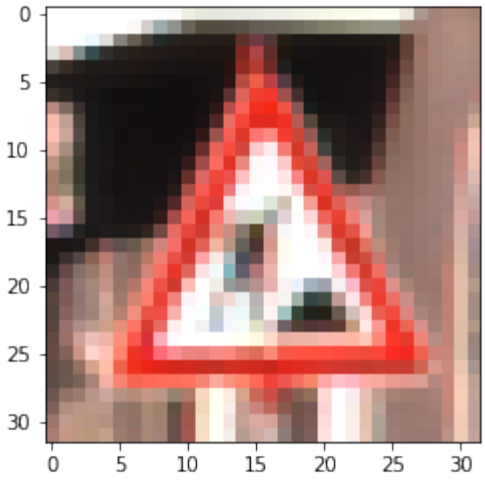
```
NN predicted 21
```



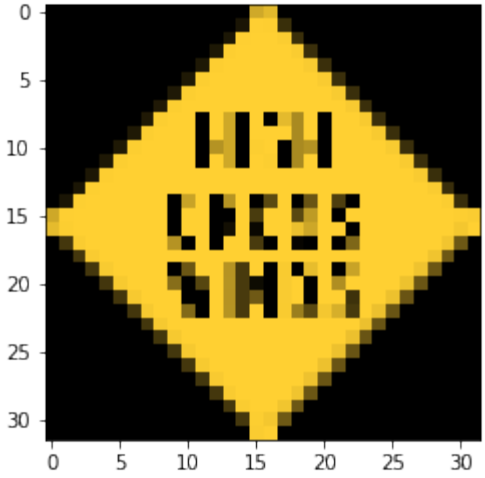
```
HCparking.png
```



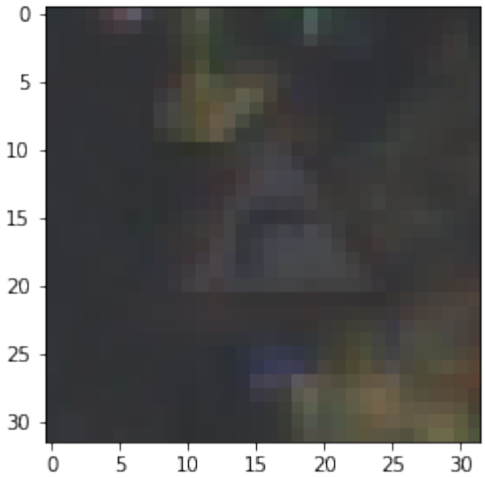
```
NN predicted 25
```



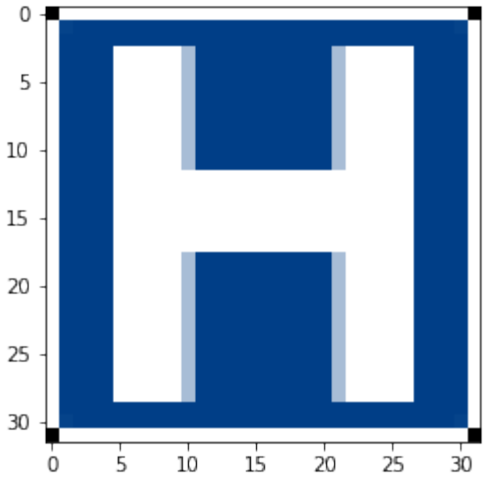
HighXWinds.png



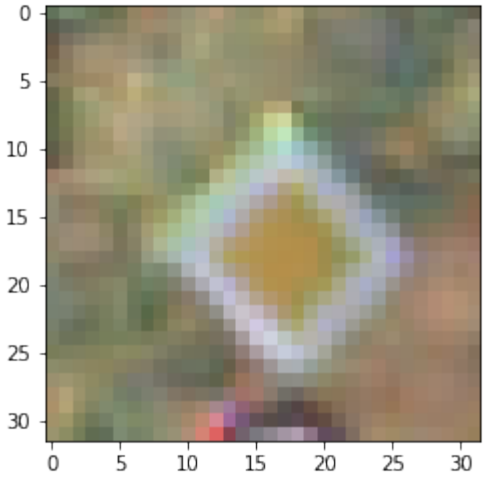
NN predicted 20



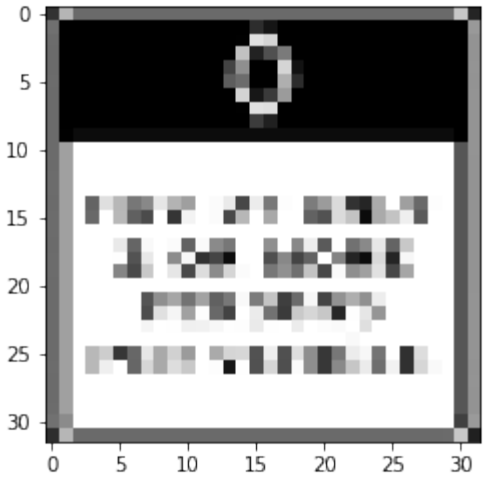
Hospital.png



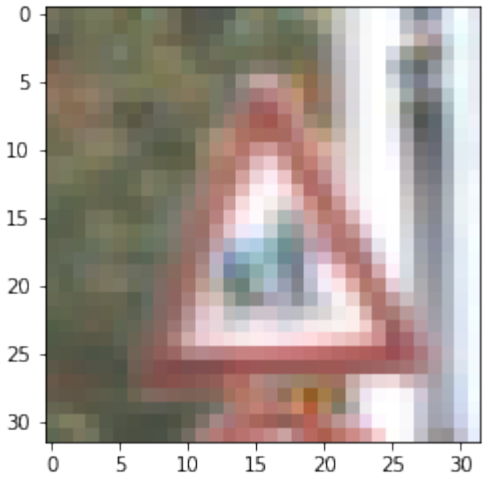
```
NN predicted 12
```



```
HOW.png
```

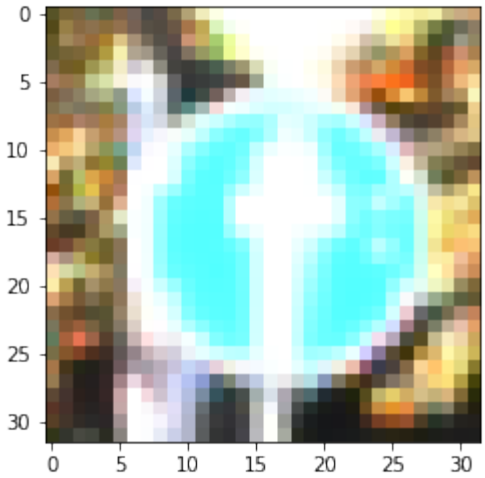


```
NN predicted 28
```



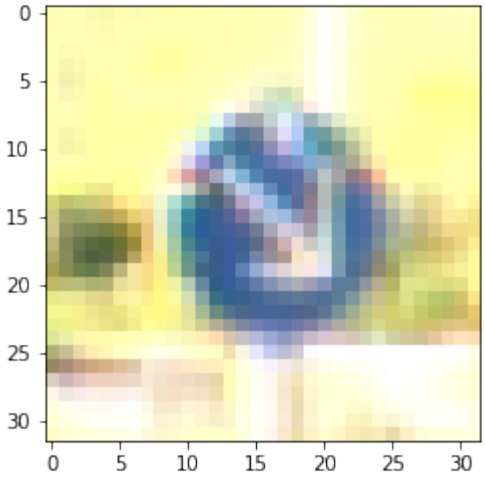
i1.png

NN predicted 35

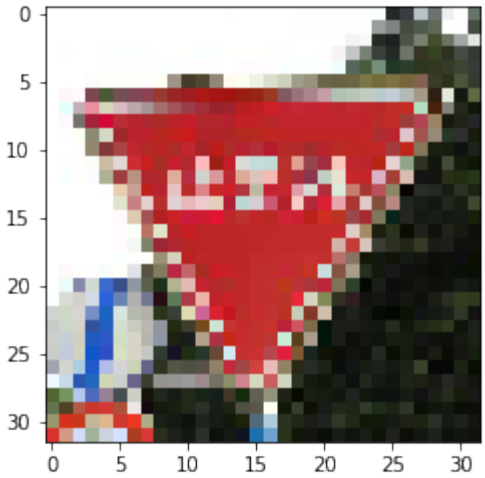


i22.png

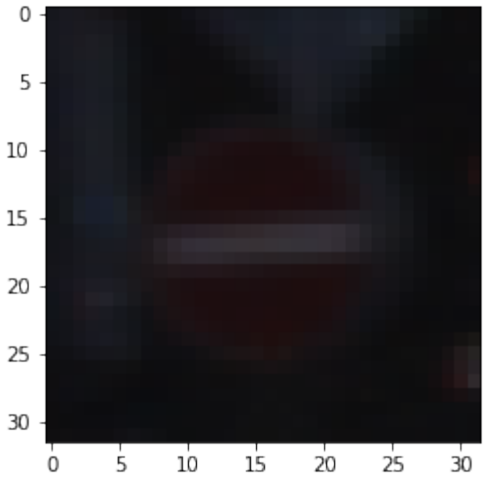
```
NN predicted 38
```



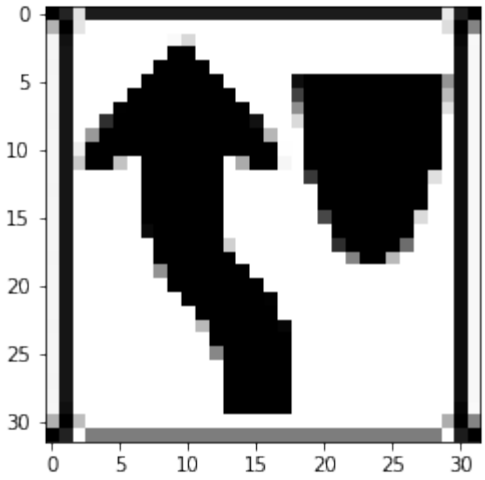
```
japanese_sign_resized.png
```



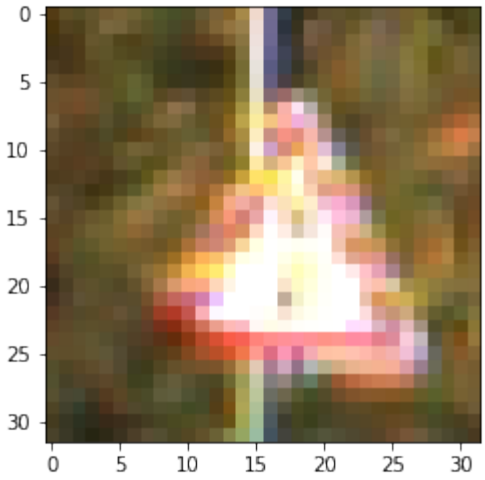
```
NN predicted 17
```



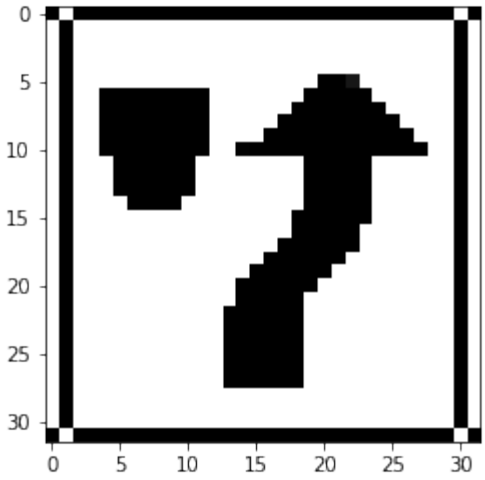
keepleft.png



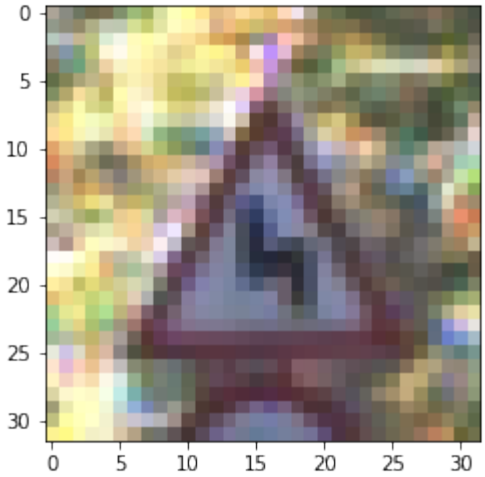
NN predicted 26



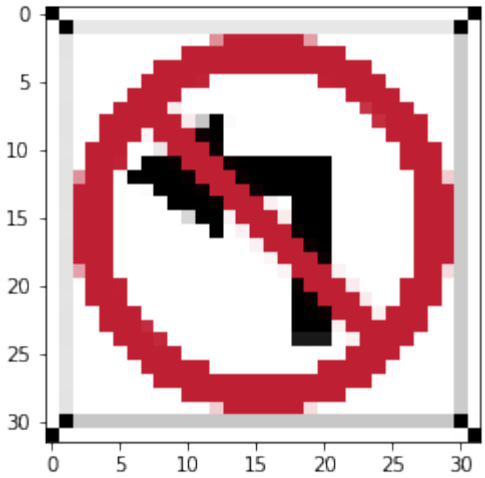
keepright.png



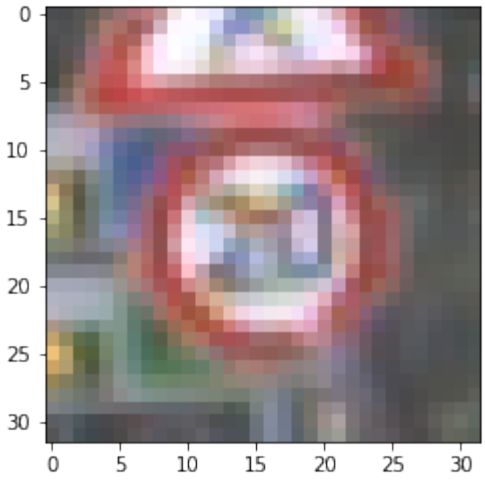
```
NN predicted 21
```



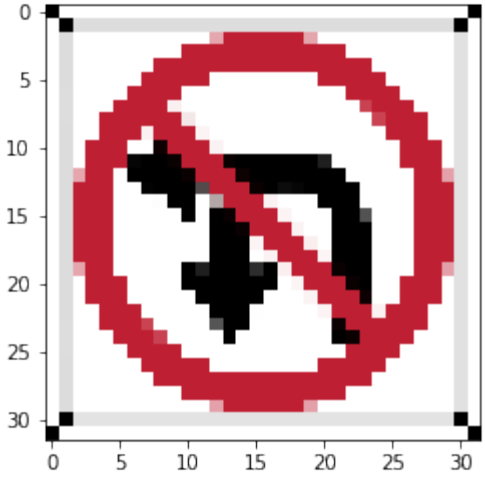
```
noleft.png
```



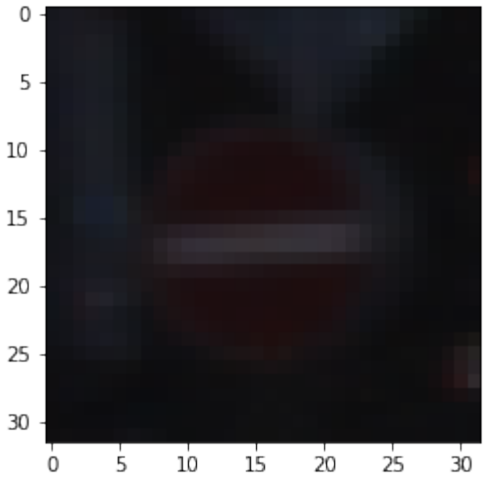
```
NN predicted 0
```



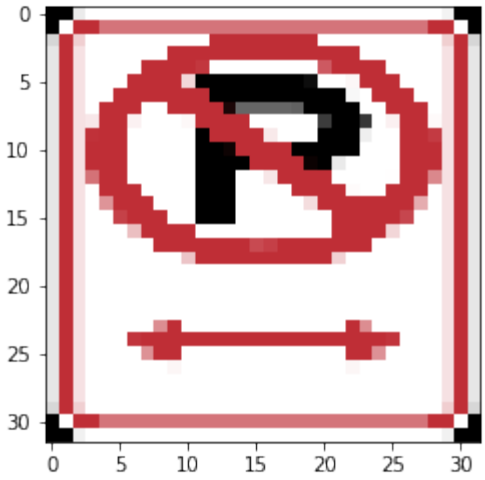
noleftUturn.png



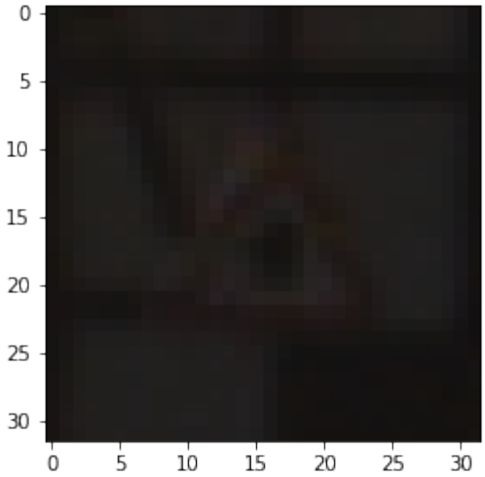
NN predicted 17



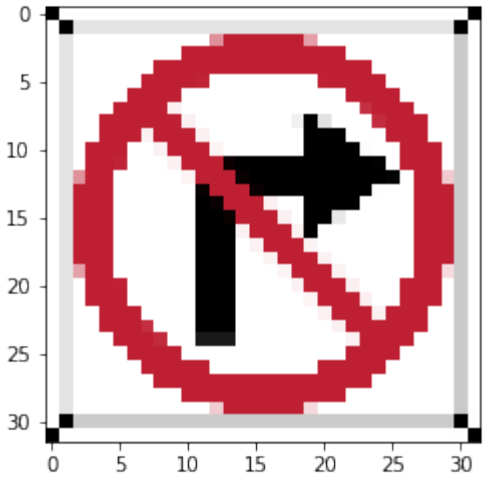
NoParking.png



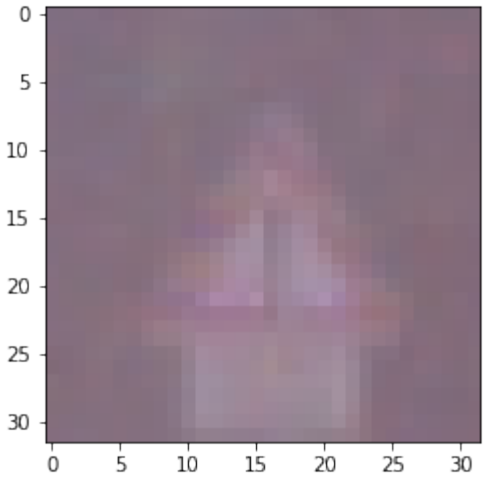
NN predicted 11



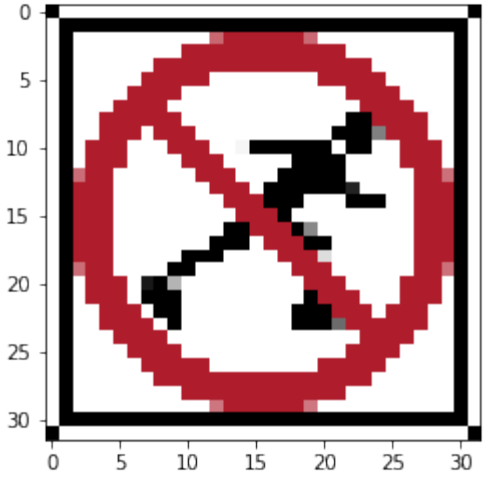
noright.png



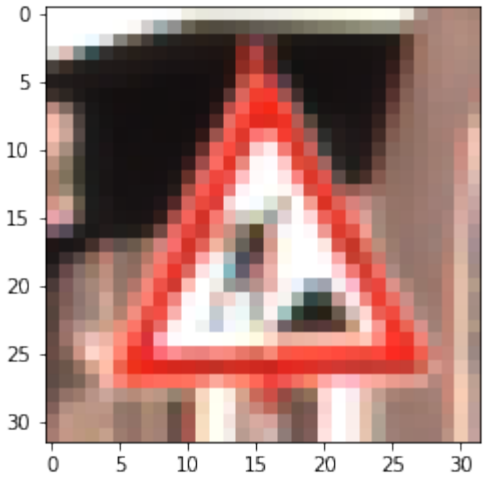
NN predicted 18



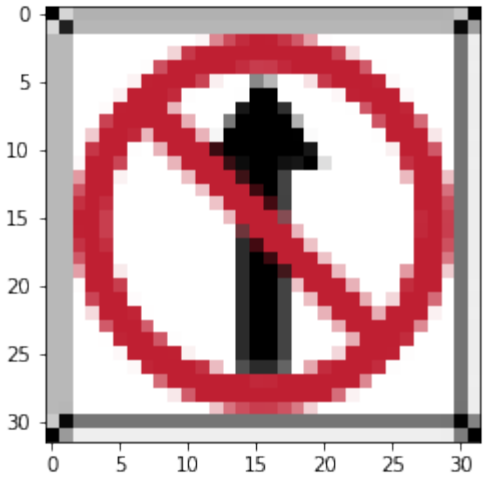
noRoller.png



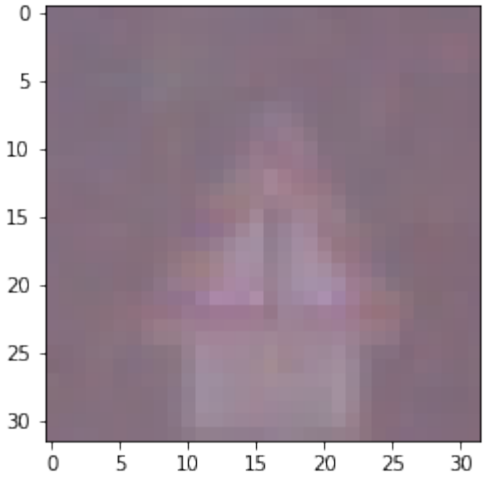
NN predicted 25



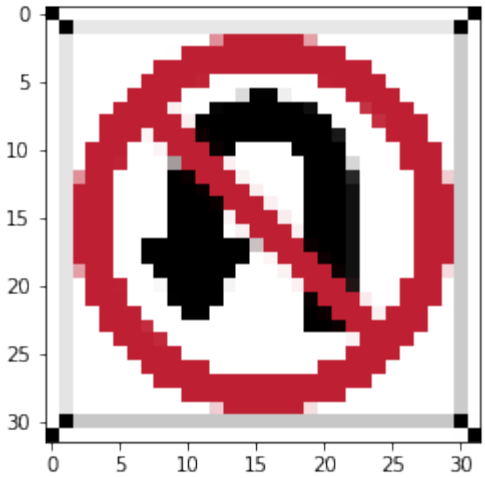
nostraight.png



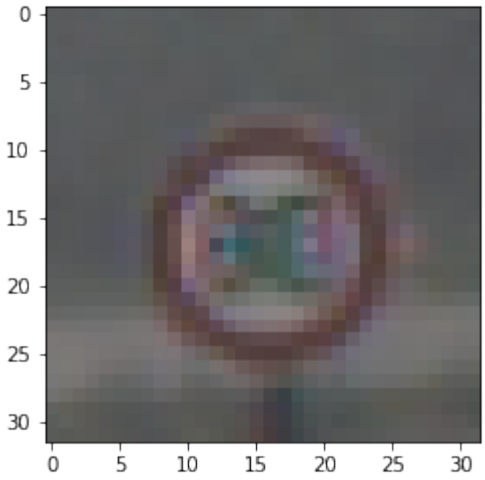
```
NN predicted 18
```



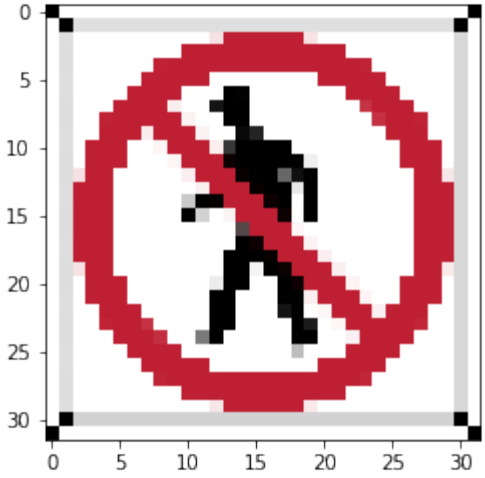
```
noUturn.png
```



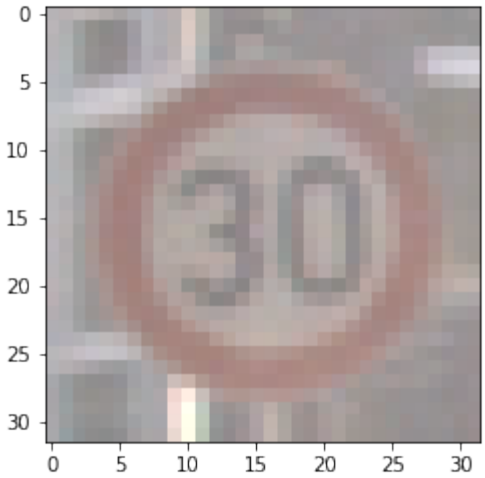
```
NN predicted 5
```



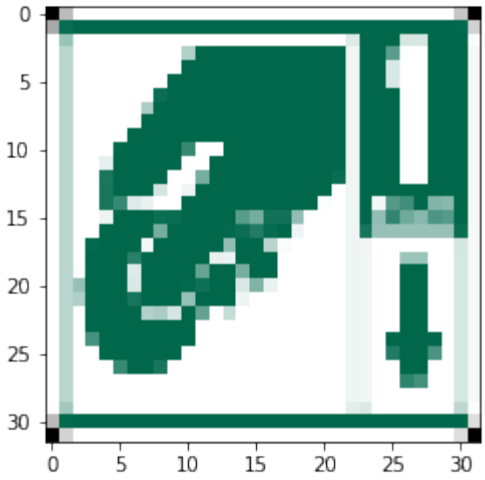
noWalking.png



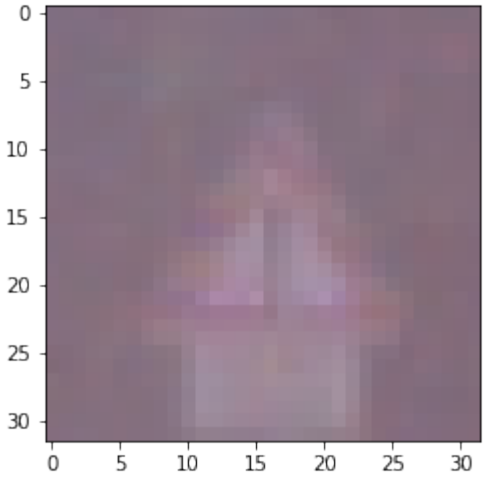
NN predicted 1



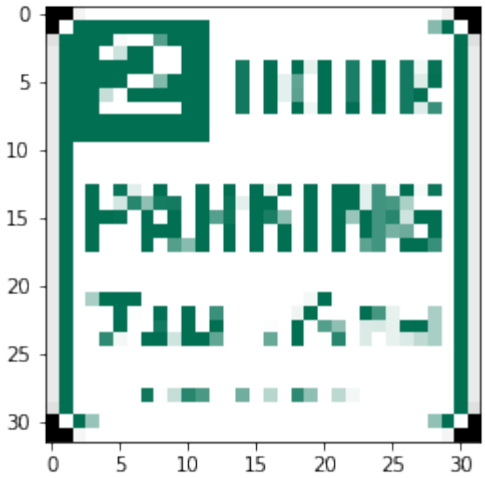
paidParking.png



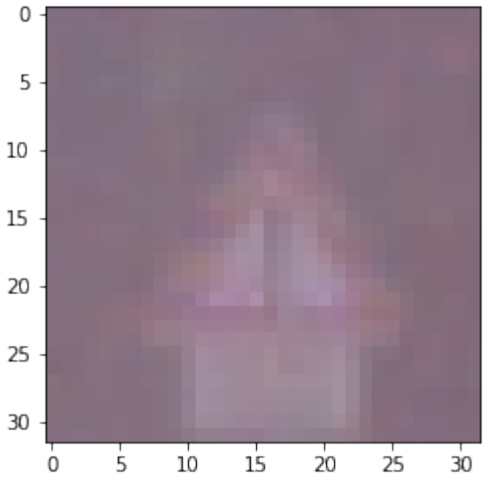
```
NN predicted 18
```



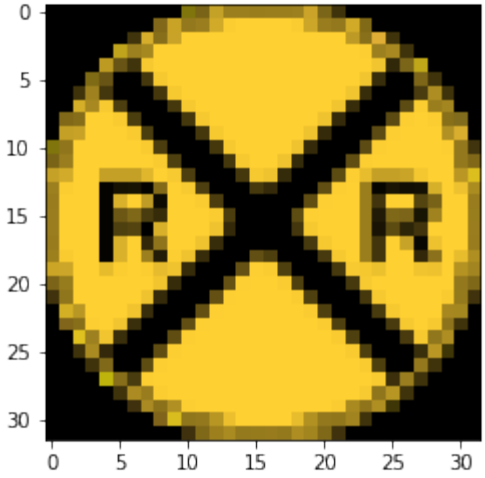
```
parking2hr.png
```



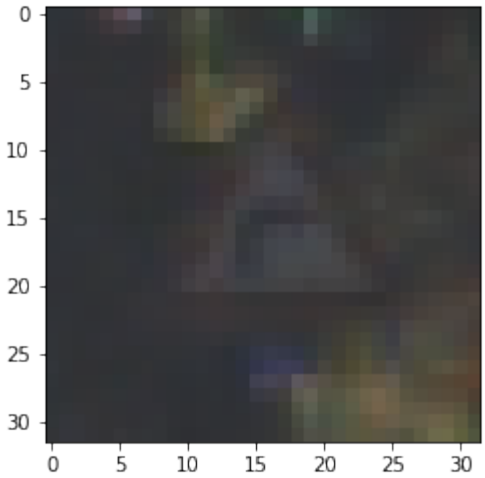
```
NN predicted 18
```



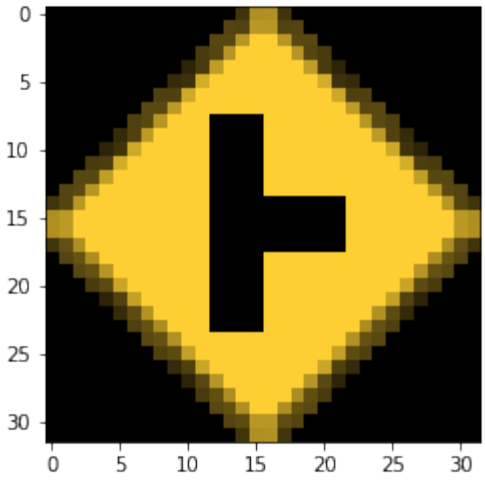
RailRoad.png



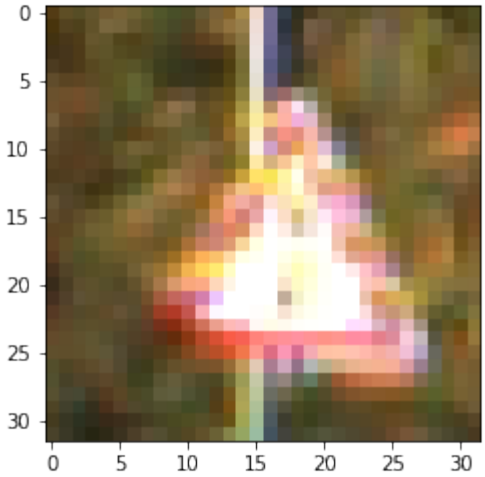
NN predicted 20



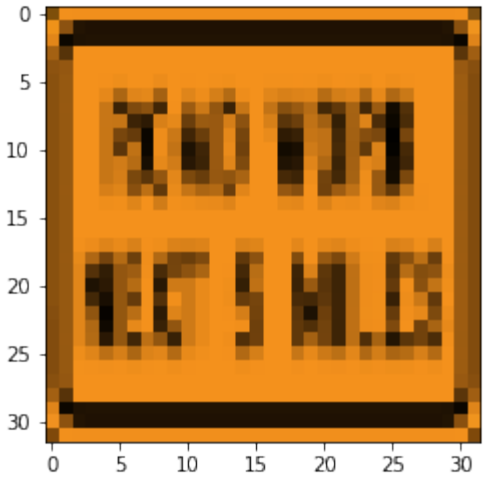
RightCrossing.png



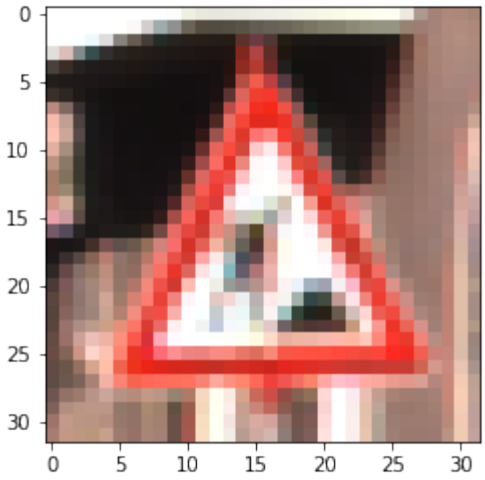
```
NN predicted 26
```



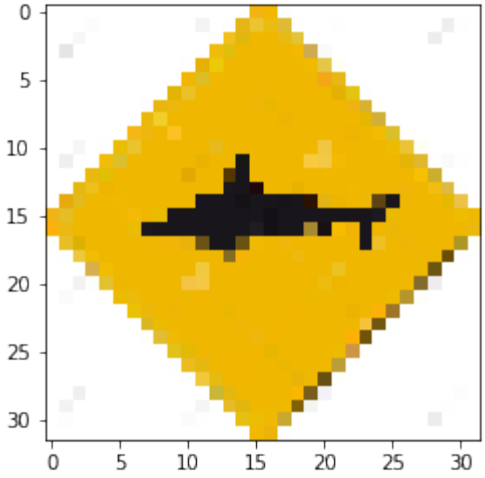
```
roadwork.png
```



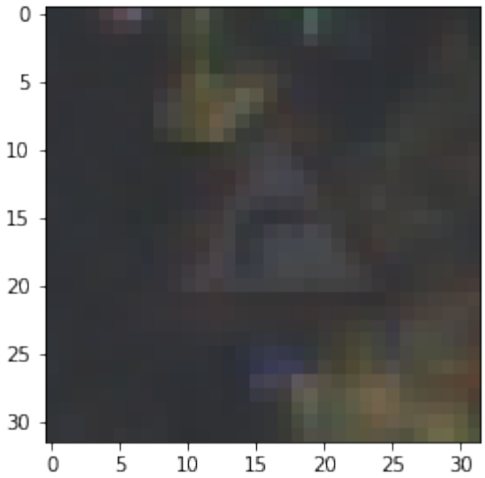
```
NN predicted 25
```



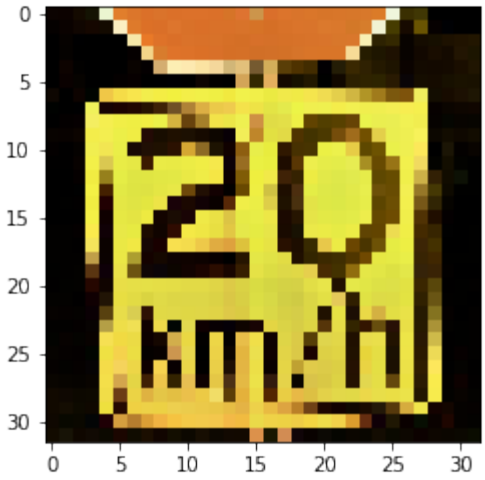
shark_sign.png



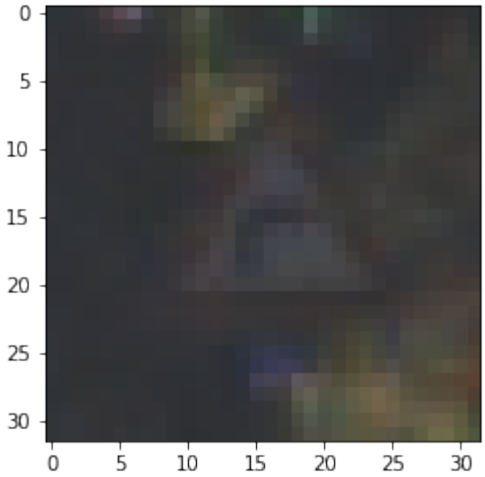
NN predicted 20



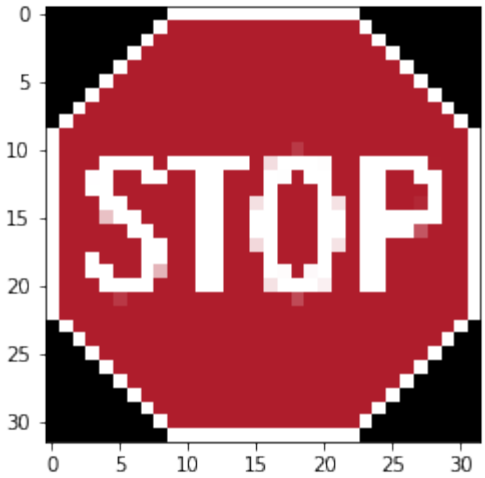
speed_limit_stop.png



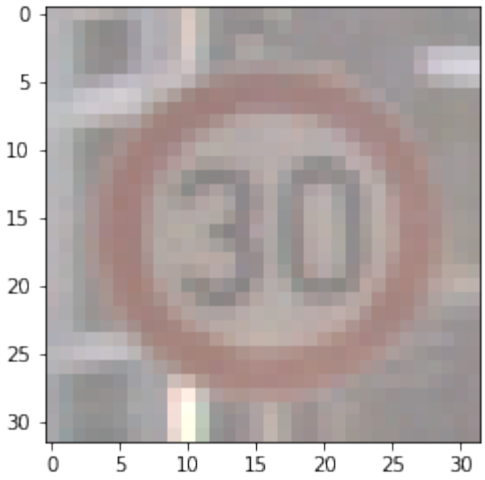

```
NN predicted 20
```



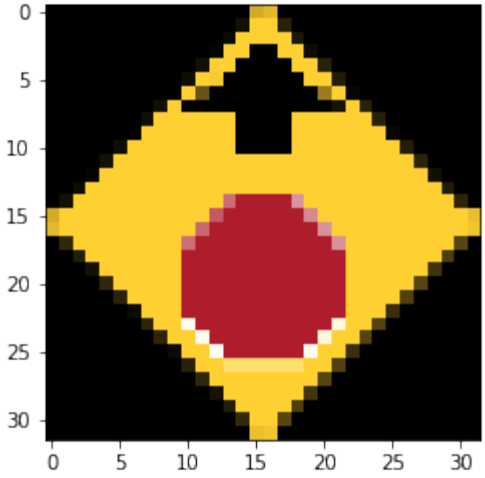
```
stop.png
```



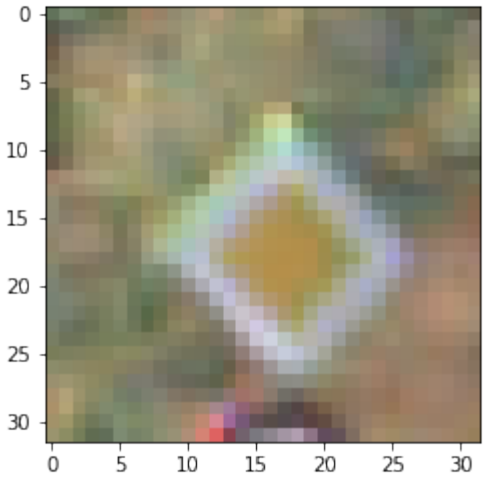

```
NN predicted 1
```



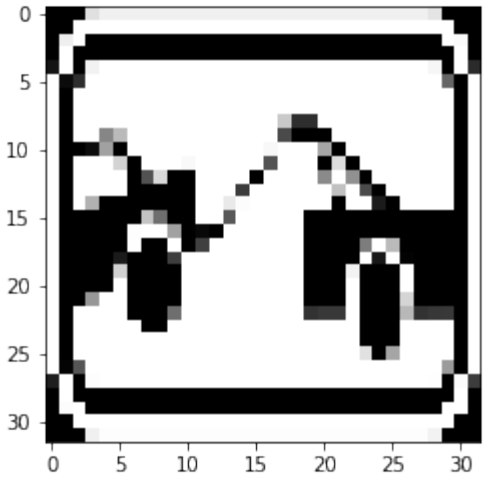
StopAhead.png



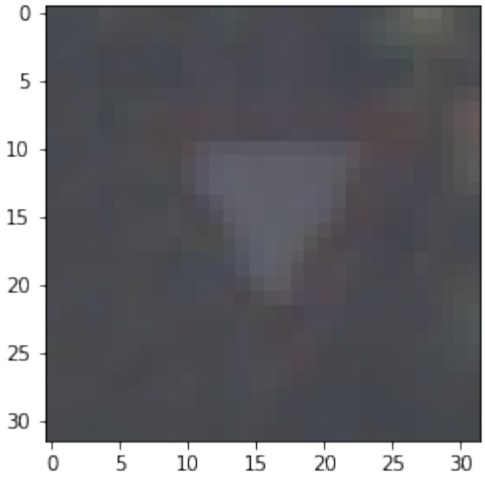
NN predicted 12



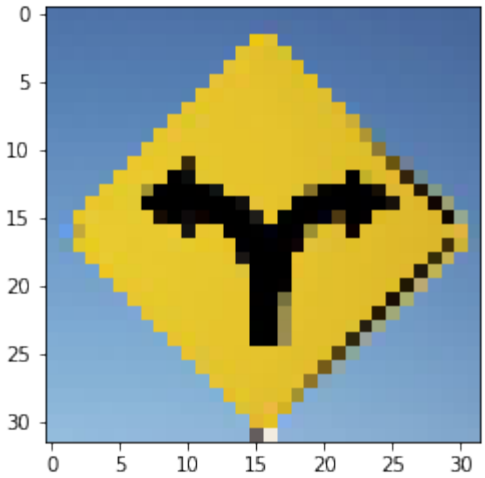
TowAway.png



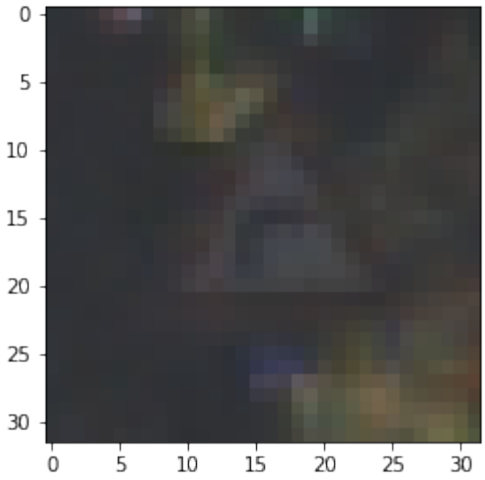
```
NN predicted 13
```



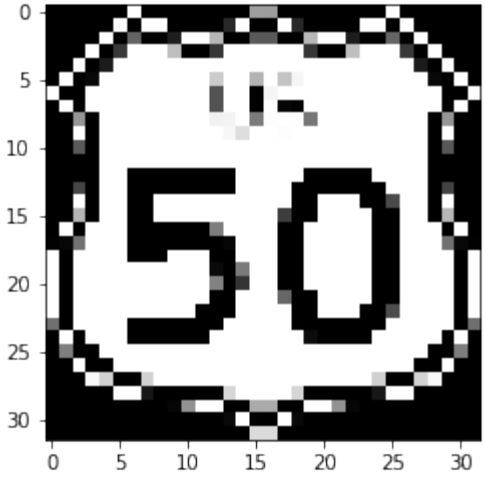
```
two_way_sign.png
```



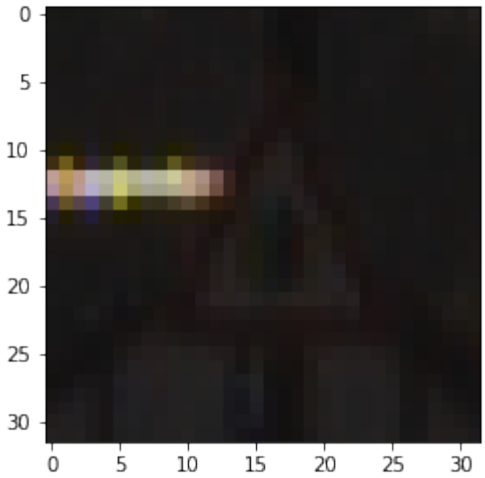
```
NN predicted 20
```



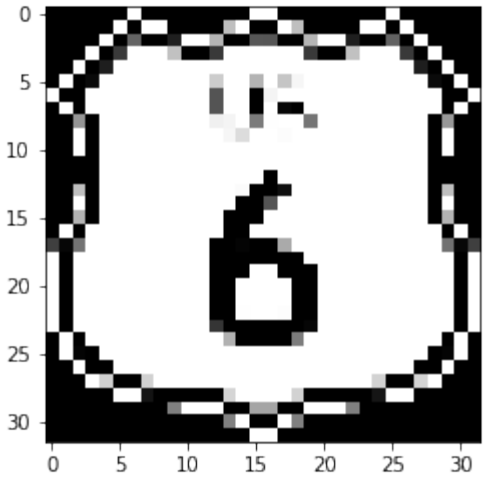
US50.png



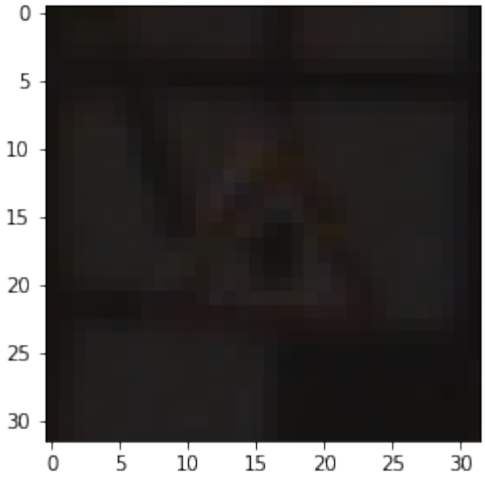
NN predicted 30



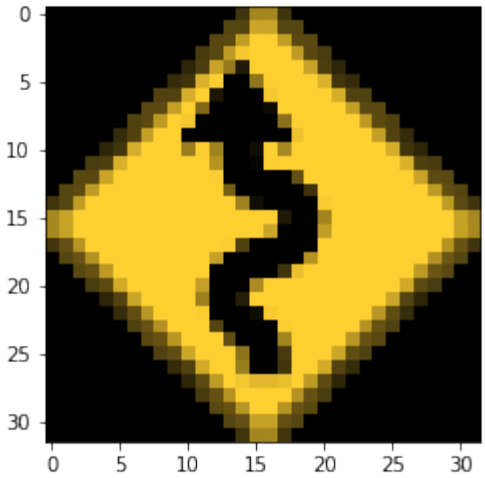
US6.png



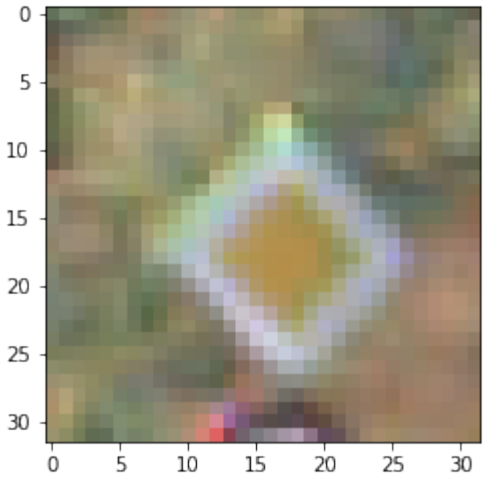
NN predicted 11



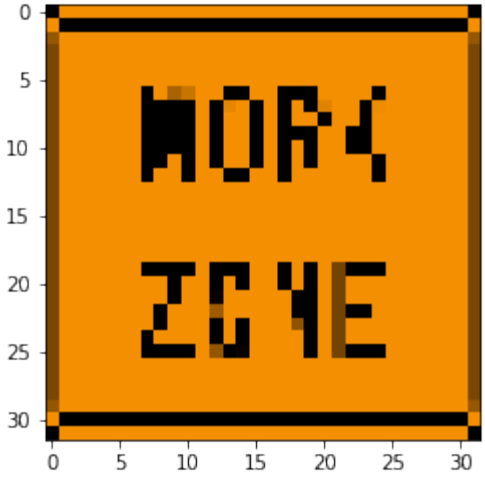
windyRight.png



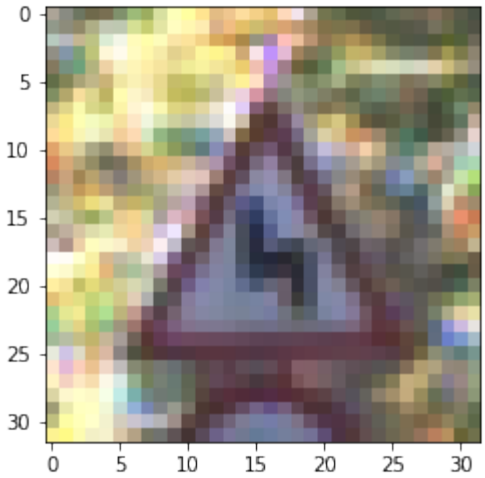
NN predicted 12



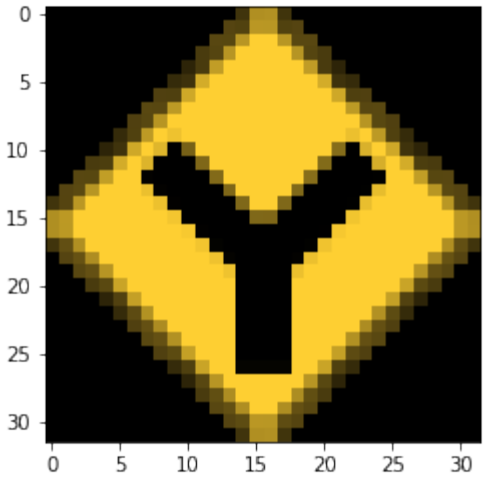
workzone.png



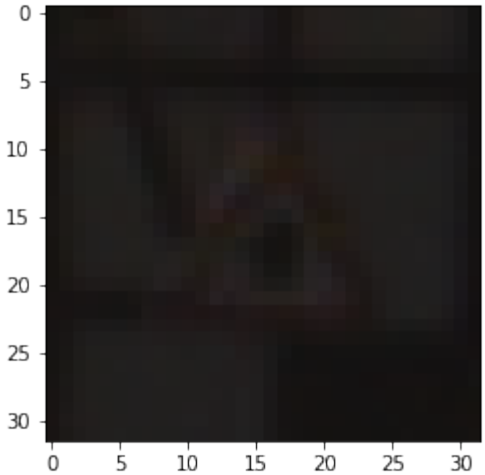
NN predicted 21



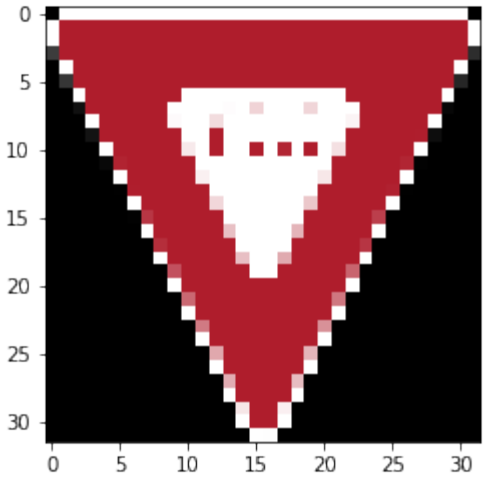
YCrossing.png



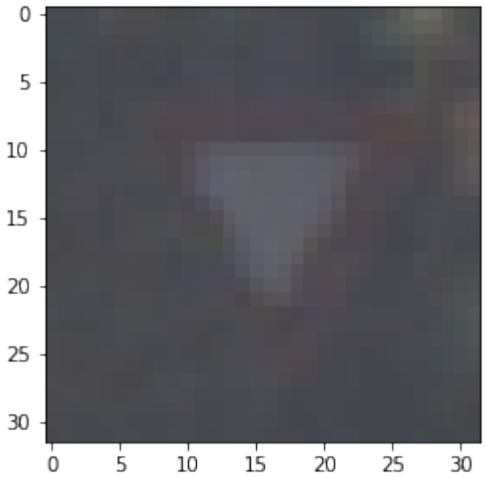
NN predicted 11



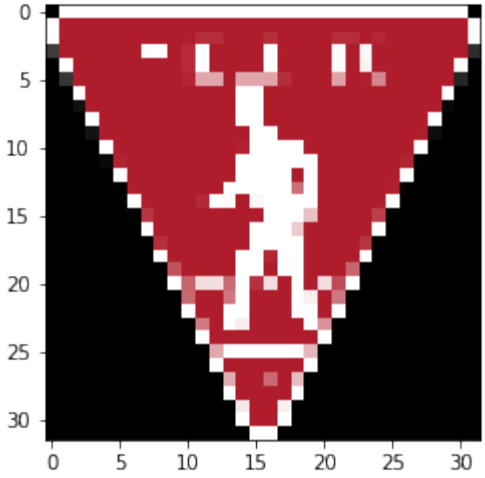
yeild.png



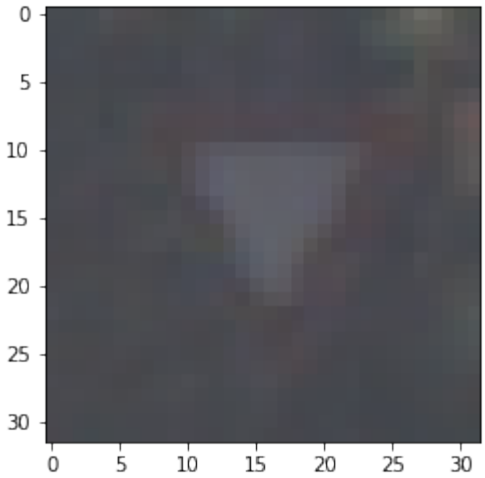
NN predicted 13



yield_pedestrian.png



NN predicted 13



No, it does not perform equally well on captured images. It has a performance of 0% accuracy on captured images as opposed to 79% on the test set. •The images not included in the dataset are not exactly the same road signs so there is additional difficulty because the model needs to generalise well to classify these new signs correctly. The

•Some road signs such as the shark sign may not even be included in the 43 categories. •The images are also processed (e.g. cropped) differently.

It seems that the model is classifying 'unknown signs' as Roundabout Mandatory signs.

Question 8

*Use the model's softmax probabilities to visualize the **certainty** of its predictions, `tf.nn.top_k` could prove helpful here. Which predictions is the model certain of? Uncertain? If the model was incorrect in its initial prediction, does the correct prediction appear in the top k? (k should be 5 at most)*

`tf.nn.top_k` will return the values and indices (class ids) of the top k predictions. So if k=3, for each sign, it'll return the 3 largest probabilities (out of a possible 43) and the correspoding class ids.

Take this numpy array as an example:

```
# (5, 6) array
a = np.array([[ 0.24879643,  0.07032244,  0.12641572,  0.34763842,
0.07893497,
         0.12789202],
       [ 0.28086119,  0.27569815,  0.08594638,  0.0178669 ,  0.18063401,
         0.15899337],
       [ 0.26076848,  0.23664738,  0.08020603,  0.07001922,  0.1134371 ,
         0.23892179],
       [ 0.11943333,  0.29198961,  0.02605103,  0.26234032,  0.1351348 ,
         0.16505091],
       [ 0.09561176,  0.34396535,  0.0643941 ,  0.16240774,  0.24206137,
         0.09155967]])
```

Running it through `sess.run(tf.nn.top_k(tf.constant(a), k=3))` produces:

```
TopKV2(values=array([[ 0.34763842,  0.24879643,  0.12789202],
       [ 0.28086119,  0.27569815,  0.18063401],
       [ 0.26076848,  0.23892179,  0.23664738],
       [ 0.29198961,  0.26234032,  0.16505091],
       [ 0.34396535,  0.24206137,  0.16240774]]), indices=array([[3, 0, 5],
       [0, 1, 4],
       [0, 5, 1],
       [1, 3, 5],
       [1, 4, 3]], dtype=int32))
```

Looking just at the first row we get `[ 0.34763842,  0.24879643,  0.12789202]`, you can confirm these are the 3 largest probabilities in a. You'll also notice `[3, 0, 5]` are the corresponding indices.

Answer:

•The model is certain of all of its predictions even though some are wrong. •The model also predicts different outcomes confidently for the two times I ran the predictions on each sign.

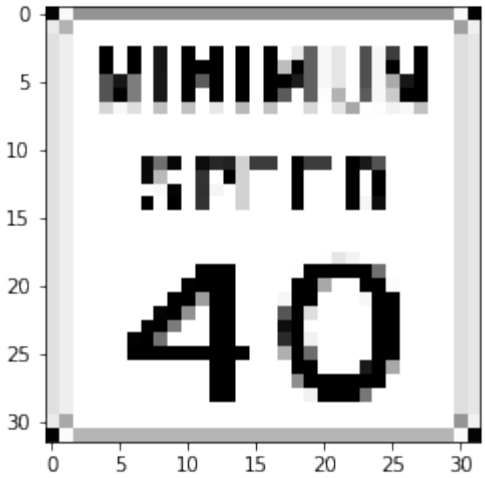
These are both strange outcomes.

> **Note**: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to \n", "**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

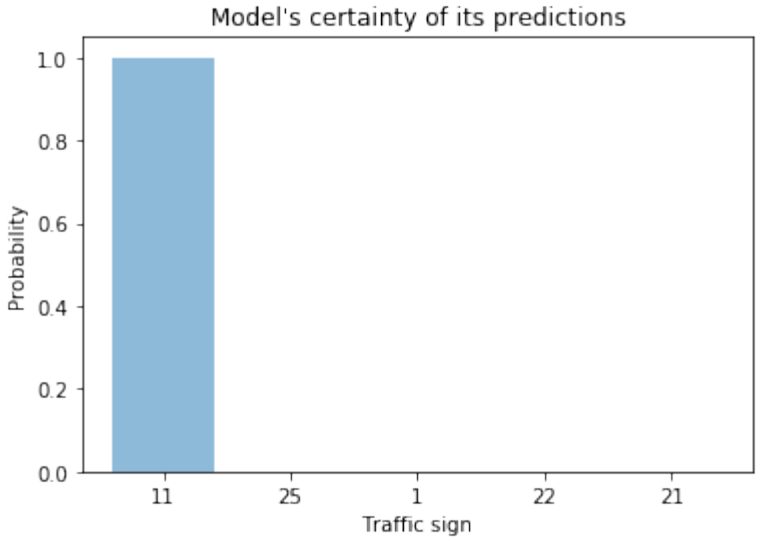
In [25]:

```
for i in range(len(namenewdata)):
    print('\n\n\n\n')
    print(namenewdata[i])
    plt.imshow(newdata[i]+.5)
    plt.show()
    top_5_predictions(newdata[i])
```

40mph.png

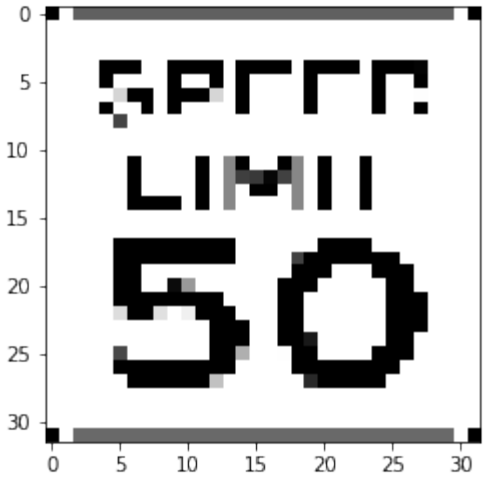



```
Top five:  TopKV2(values=array([[  9.99937534e-01,   6.24219028e-05,   1.94342125e-08,
          5.18475931e-12,   1.91675446e-15]], dtype=float32), indices=array([[11, 25, 1, 22, 21]]))
```

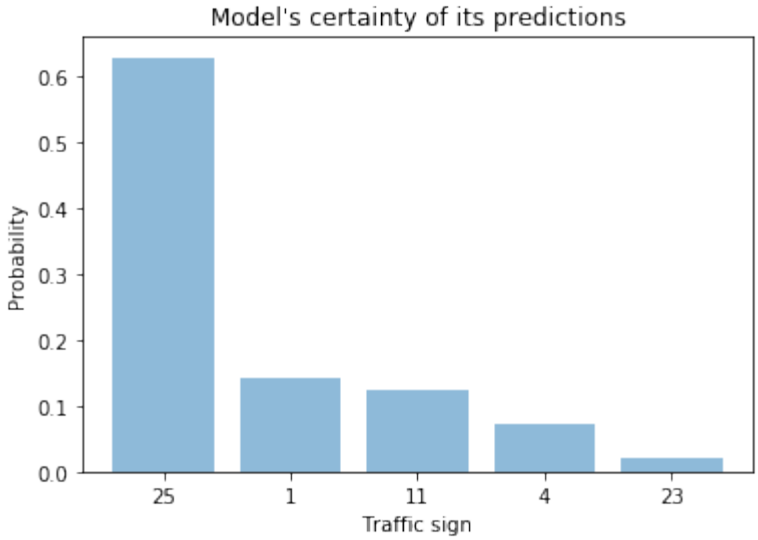



```
Traffic Sign Key
11 :  Right-of-way at the next intersection
25 :  Road work
1 :  Speed limit (30km/h)
22 :  Bumpy road
21 :  Double curve
```

```
50mph.png
```

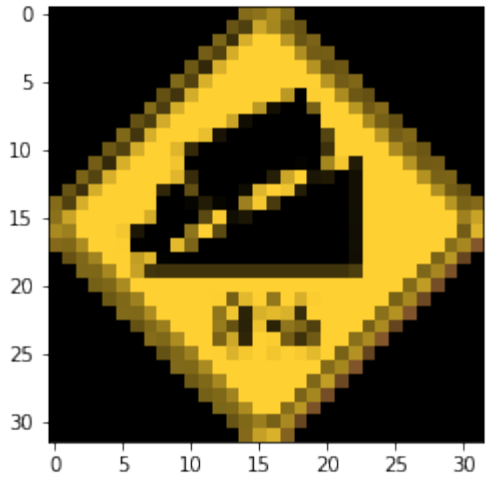



```
Top five:  TopKV2(values=array([[ 0.62737113,  0.14048479,  0.12503   ,  0.07073754,
0.01937948]], dtype=float32), indices=array([[25,  1, 11,  4, 23]]))
```

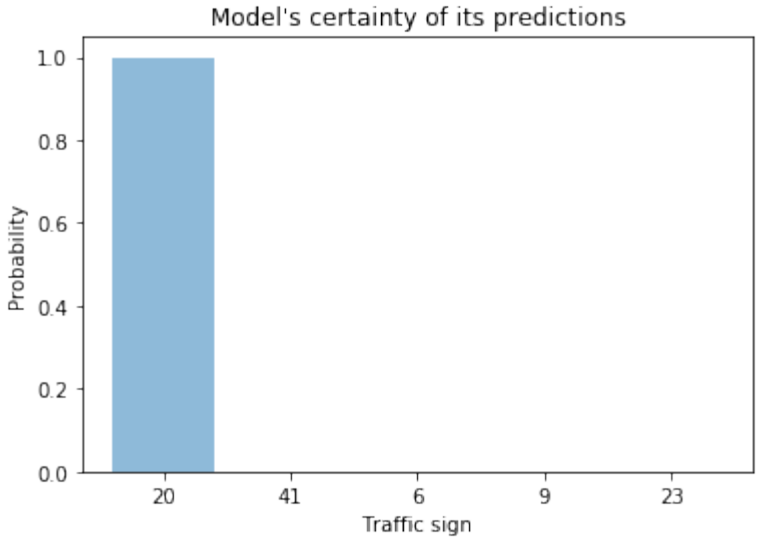



```
Traffic Sign Key
25 :  Road work
1 :  Speed limit (30km/h)
11 :  Right-of-way at the next intersection
4 :  Speed limit (70km/h)
23 :  Slippery road
```

```
8pcGrade.png
```

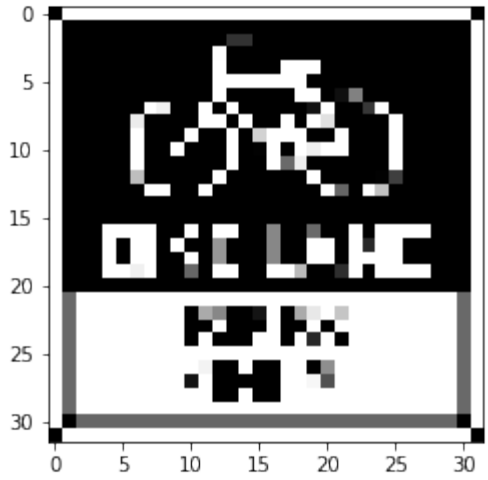



```
Top five:  TopKV2(values=array([[  9.99295354e-01,   5.64003130e-04,   7.08036314e-
05,
          5.88282855e-05,   8.32758360e-06]], dtype=float32), indices=array([[20, 41,
6,  9, 23]]))
```

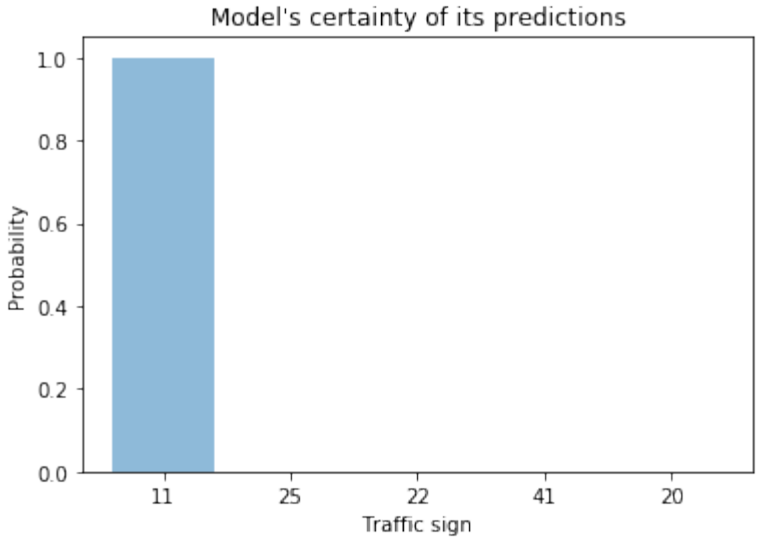



```
Traffic Sign Key
20 :  Dangerous curve to the right
41 :  End of no passing
6 :  End of speed limit (80km/h)
9 :  No passing
23 :  Slippery road
```

```
bikelane.png
```

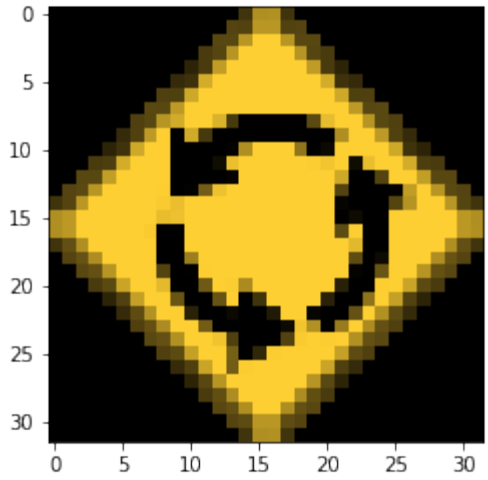



```
Top five:  TopKV2(values=array([[  1.00000000e+00,   1.81508251e-14,   1.56764940e-
15,
          5.16322747e-18,   3.22120266e-21]], dtype=float32), indices=array([[11, 25,
22, 41, 20]]))
```

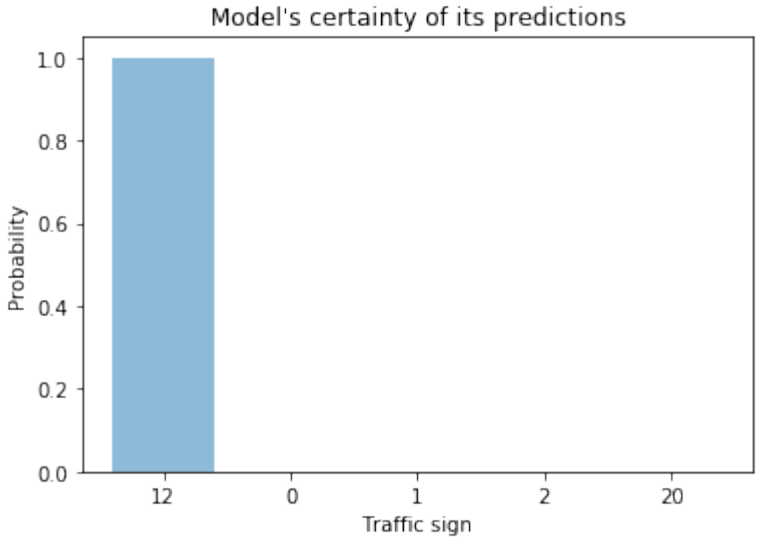



```
Traffic Sign Key
11 :  Right-of-way at the next intersection
25 :  Road work
22 :  Bumpy road
41 :  End of no passing
20 :  Dangerous curve to the right
```

```
circle.png
```

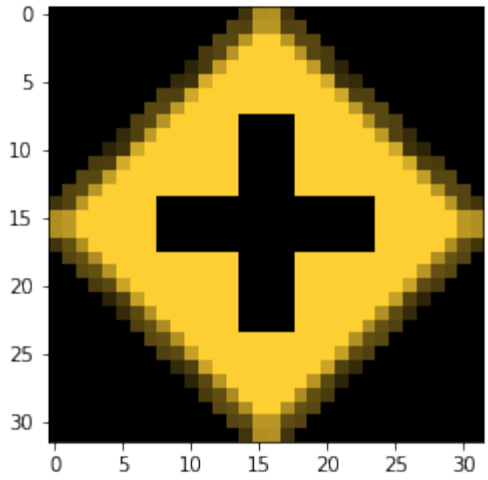



```
Top five:  TopKV2(values=array([[  9.99999881e-01,   9.32753466e-08,   1.92791526e-
14,
          4.12247296e-17,   1.18911002e-17]], dtype=float32), indices=array([[12,  0,
1,  2, 20]]))
```

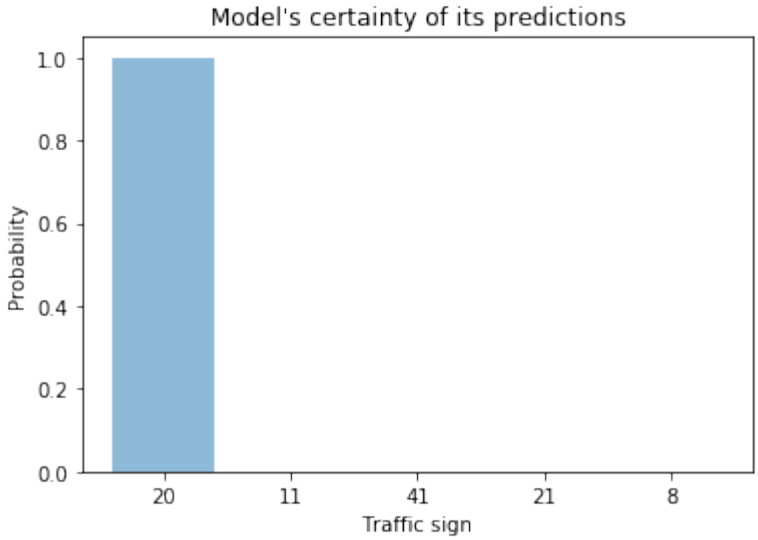



```
Traffic Sign Key
12 :  Priority road
0 :  Speed limit (20km/h)
1 :  Speed limit (30km/h)
2 :  Speed limit (50km/h)
20 :  Dangerous curve to the right
```

```
CrossRoad.png
```

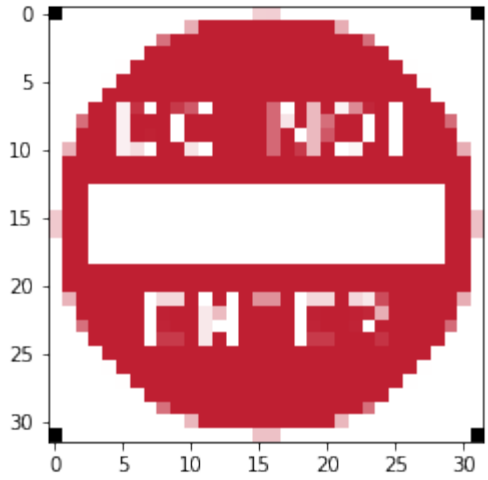



```
Top five:  TopKV2(values=array([[  9.99999642e-01,   3.53175523e-07,   1.71302143e-
12,
          1.12421140e-12,   5.13325751e-14]], dtype=float32), indices=array([[20, 11,
41, 21,  8]]))
```

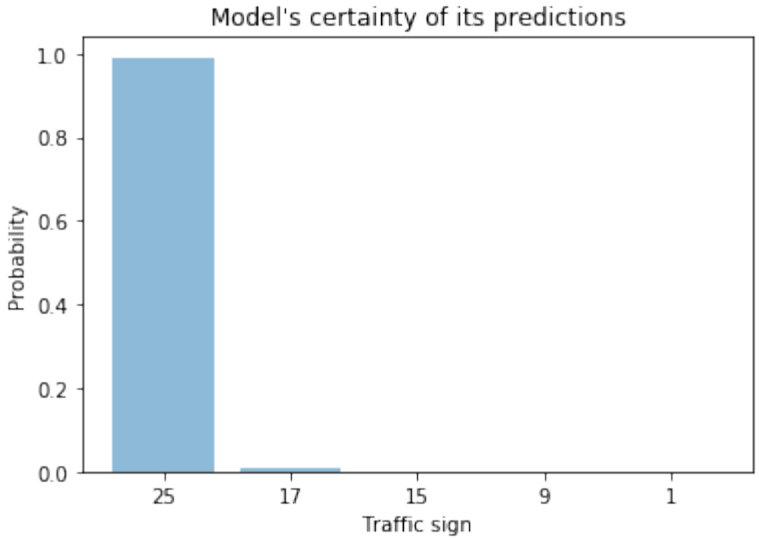



```
Traffic Sign Key
20 :  Dangerous curve to the right
11 :  Right-of-way at the next intersection
41 :  End of no passing
21 :  Double curve
8 :  Speed limit (120km/h)
```

```
DonotEnter.png
```

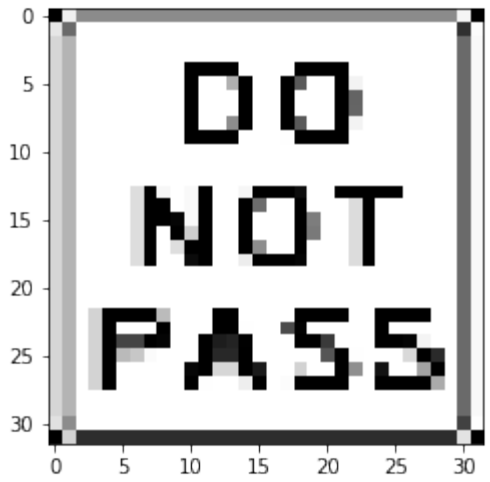


```
Top five:  TopKV2(values=array([[  9.91163492e-01,   8.83655716e-03,   6.28953251e-
16,
          1.94010580e-17,   3.82024965e-19]], dtype=float32), indices=array([[25, 17,
15,  9,  1]]))
```

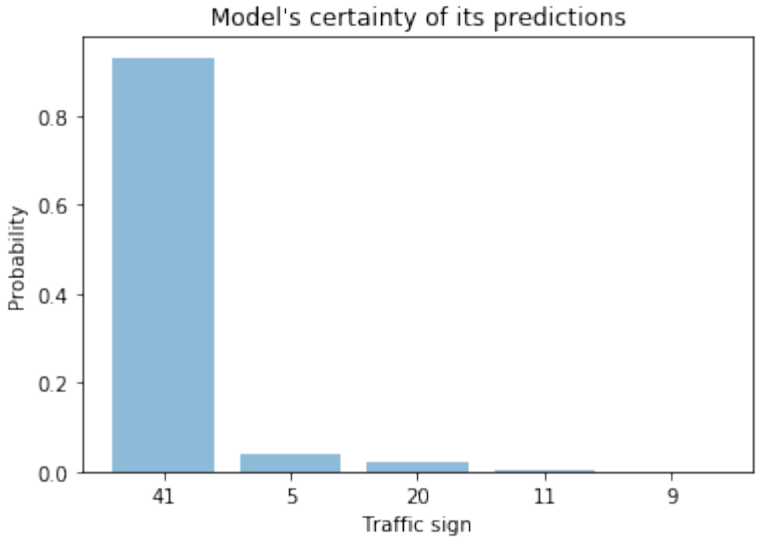



```
Traffic Sign Key
25 :  Road work
17 :  No entry
15 :  No vehicles
9 :  No passing
1 :  Speed limit (30km/h)
```

```
DoNotPass.png
```

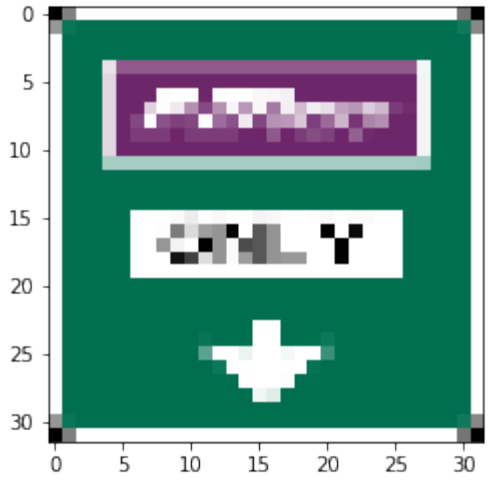



```
Top five:  TopKV2(values=array([[  9.31805491e-01,   4.08876874e-02,   2.15184521e-
02,
          5.67048974e-03,   1.08646913e-04]], dtype=float32), indices=array([[41,  5,
20, 11,  9]]))
```

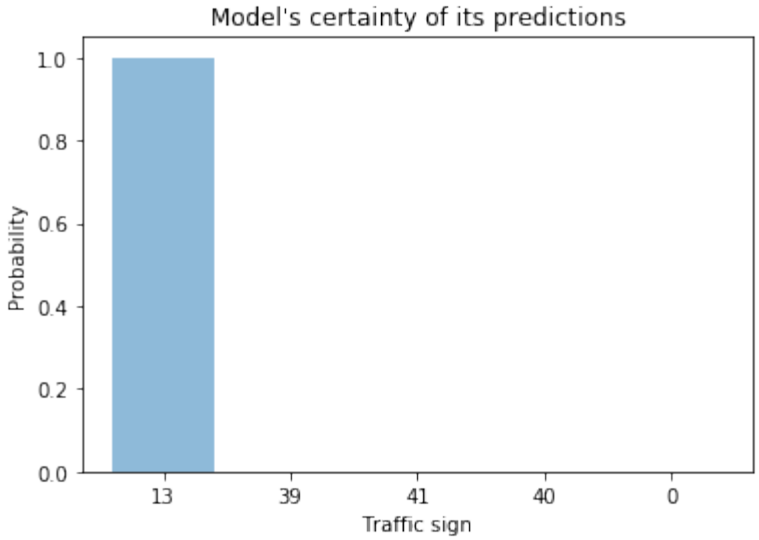



```
Traffic Sign Key
41 :  End of no passing
5 :  Speed limit (80km/h)
20 :  Dangerous curve to the right
11 :  Right-of-way at the next intersection
9 :  No passing
```

```
EZPass.png
```

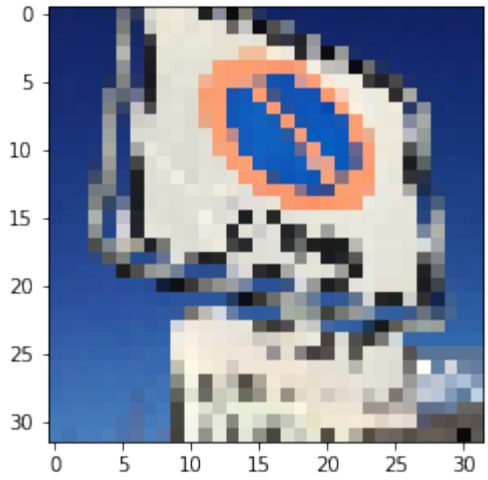



```
Top five:  TopKV2(values=array([[  1.00000000e+00,   1.96948377e-32,   3.05492874e-
40,
          1.76913931e-41,   0.00000000e+00]], dtype=float32), indices=array([[13, 39,
41, 40,  0]]))
```

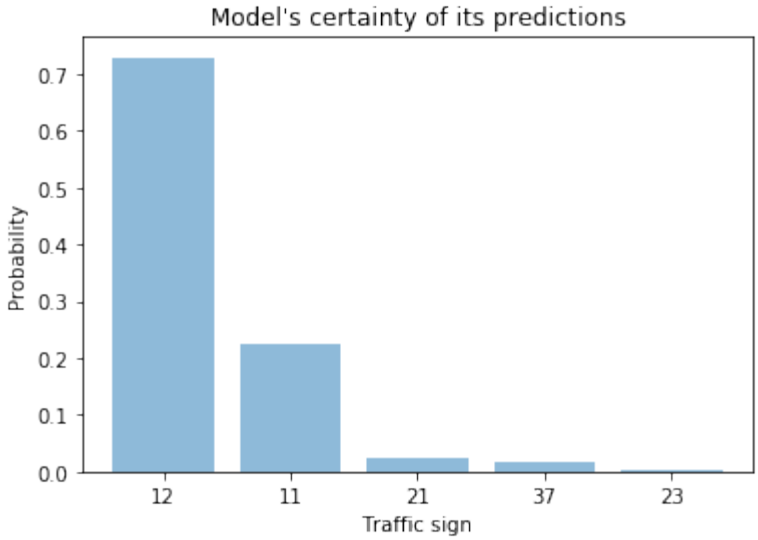



```
Traffic Sign Key
13 :  Yield
39 :  Keep left
41 :  End of no passing
40 :  Roundabout mandatory
0 :  Speed limit (20km/h)
```

```
german_sign.png
```

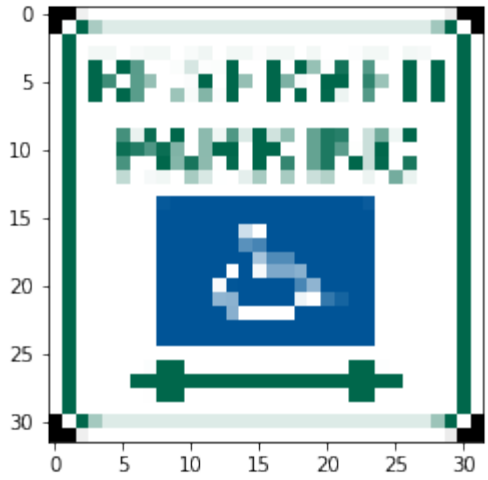



```
Top five:  TopKV2(values=array([[ 0.72874165,  0.22346744,  0.02429088,  0.01657704,
0.00366458]], dtype=float32), indices=array([[12, 11, 21, 37, 23]]))
```

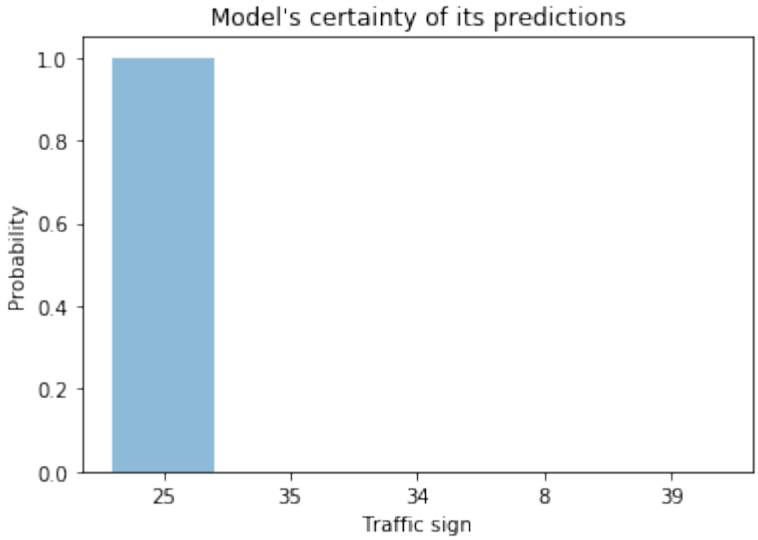



```
Traffic Sign Key
12 :  Priority road
11 :  Right-of-way at the next intersection
21 :  Double curve
37 :  Go straight or left
23 :  Slippery road
```

```
HCparking.png
```

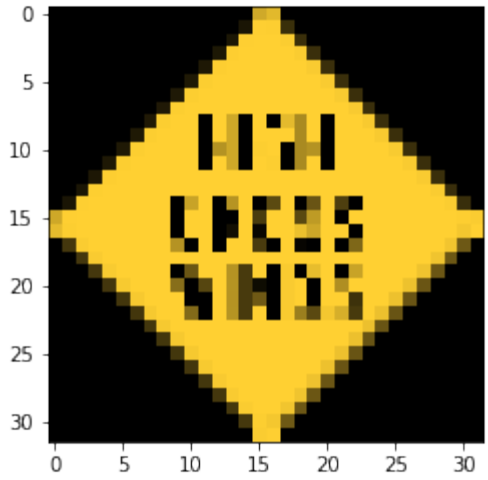



```
Top five:  TopKV2(values=array([[  1.00000000e+00,   2.26646132e-12,   1.98798484e-
13,
          3.26556739e-15,   5.62976809e-17]], dtype=float32), indices=array([[25, 35,
34,  8, 39]]))
```

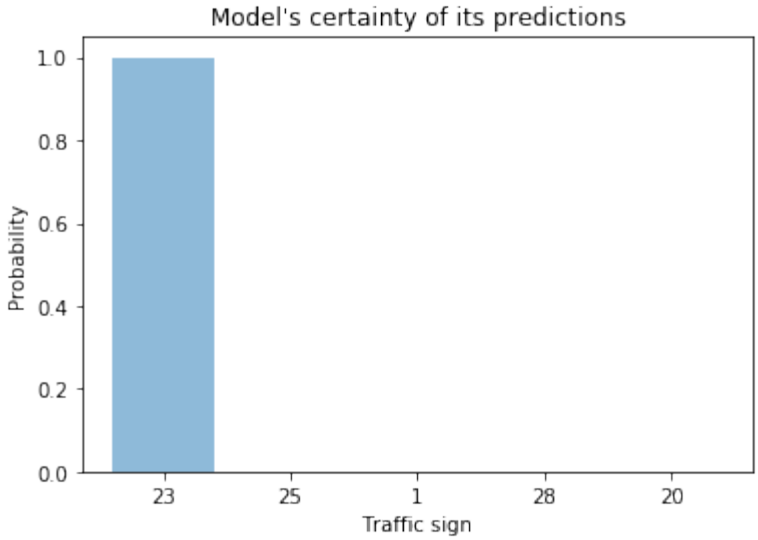



```
Traffic Sign Key
25 :  Road work
35 :  Ahead only
34 :  Turn left ahead
8 :  Speed limit (120km/h)
39 :  Keep left
```

```
HighXWinds.png
```

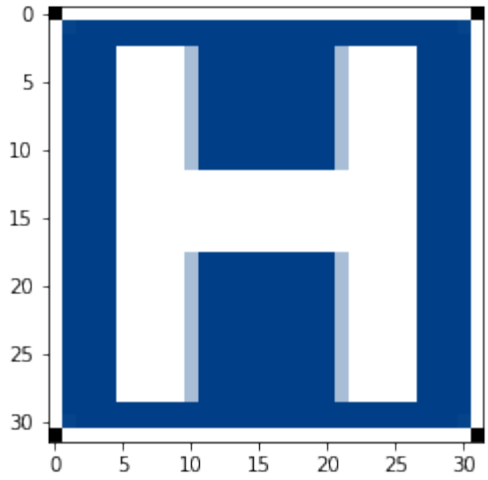


```
Top five:  TopKV2(values=array([[  9.99704182e-01,   1.67899881e-04,   8.09978083e-
05,
          4.62095595e-05,   4.30413706e-07]], dtype=float32), indices=array([[23, 25,
1, 28, 20]]))
```

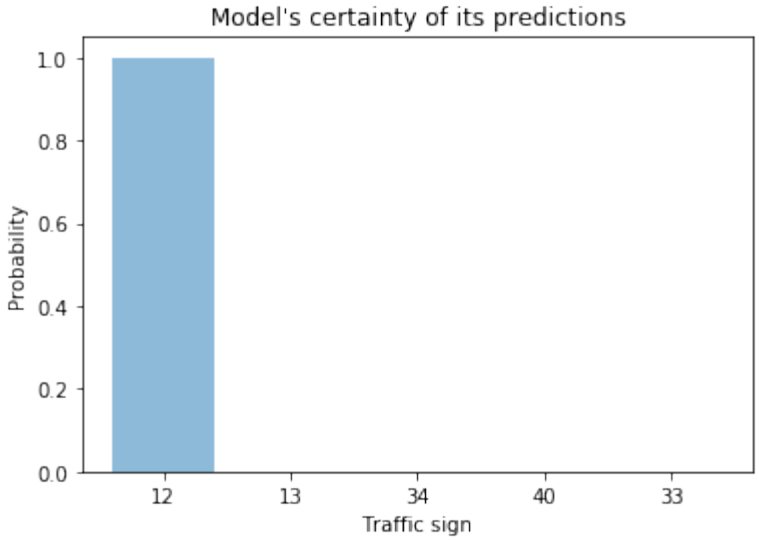



```
Traffic Sign Key
23 :  Slippery road
25 :  Road work
1 :  Speed limit (30km/h)
28 :  Children crossing
20 :  Dangerous curve to the right
```

```
Hospital.png
```

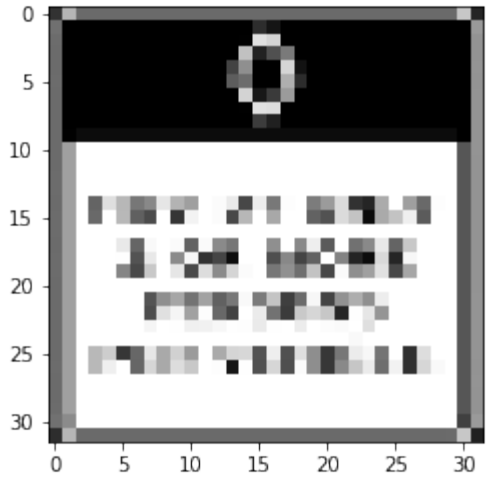



```
Top five:  TopKV2(values=array([[  1.00000000e+00,   1.38411309e-12,   1.35214458e-
17,
          3.13487416e-31,   2.00177247e-33]], dtype=float32), indices=array([[12, 13,
34, 40, 33]]))
```

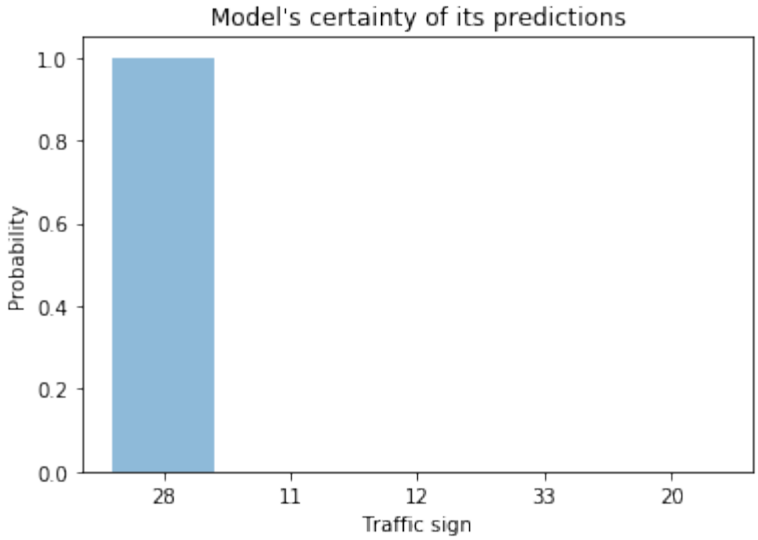



```
Traffic Sign Key
12 :  Priority road
13 :  Yield
34 :  Turn left ahead
40 :  Roundabout mandatory
33 :  Turn right ahead
```

HOW.png



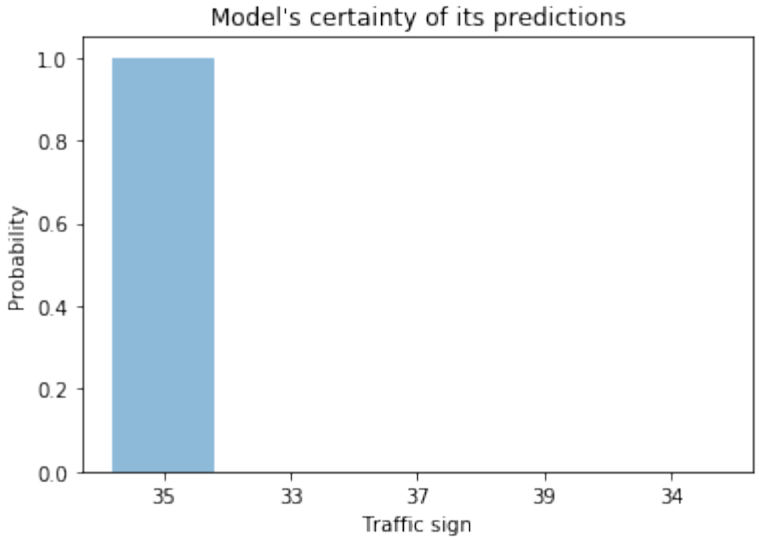

```
Top five:  TopKV2(values=array([[  9.99999881e-01,   7.25822602e-08,   3.54673513e-11,
          1.42352406e-12,   8.91214619e-13]], dtype=float32), indices=array([[28, 11, 12, 33, 20]]))
```




```
Traffic Sign Key
28 :  Children crossing
11 :  Right-of-way at the next intersection
12 :  Priority road
33 :  Turn right ahead
20 :  Dangerous curve to the right
```

```
i1.png
```

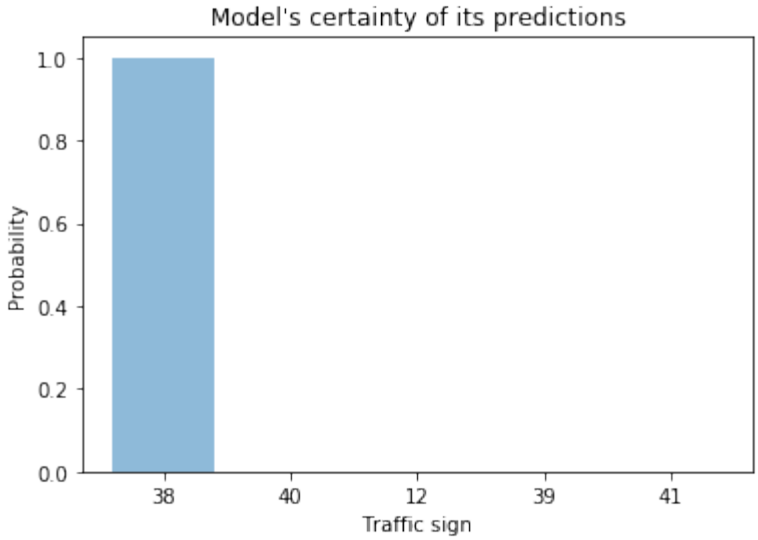

```
Top five:  TopKV2(values=array([[  9.99998927e-01,   1.07158735e-06,   1.74728336e-
23,
          3.17016235e-26,   9.61646553e-29]], dtype=float32), indices=array([[35, 33,
37, 39, 34]]))
```




```
Traffic Sign Key
35 :  Ahead only
33 :  Turn right ahead
37 :  Go straight or left
39 :  Keep left
34 :  Turn left ahead
```

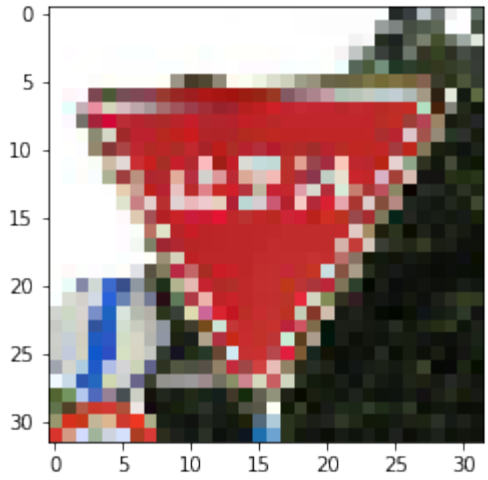
```
i22.png
```


```
Top five:  TopKV2(values=array([[  9.99999881e-01,   9.80420864e-08,   5.27918209e-
10,
          1.35520490e-12,   2.83475879e-13]], dtype=float32), indices=array([[38, 40,
12, 39, 41]]))
```

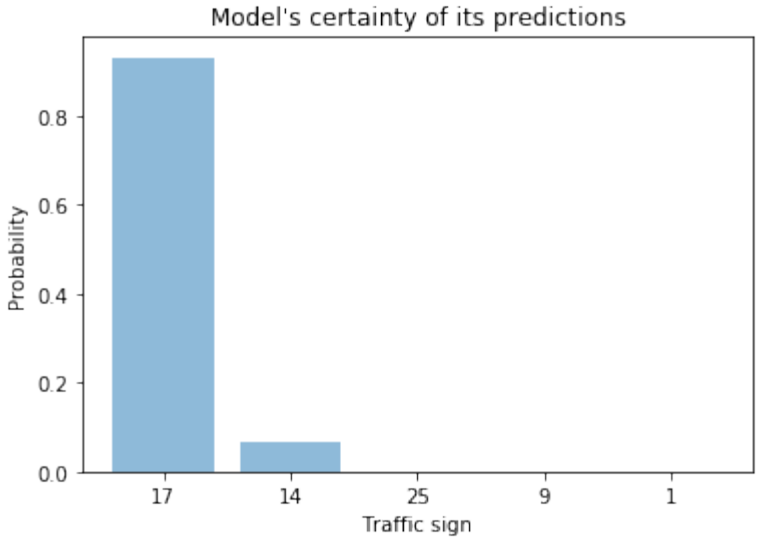



```
Traffic Sign Key
38 :  Keep right
40 :  Roundabout mandatory
12 :  Priority road
39 :  Keep left
41 :  End of no passing
```

```
japanese_sign_resized.png
```

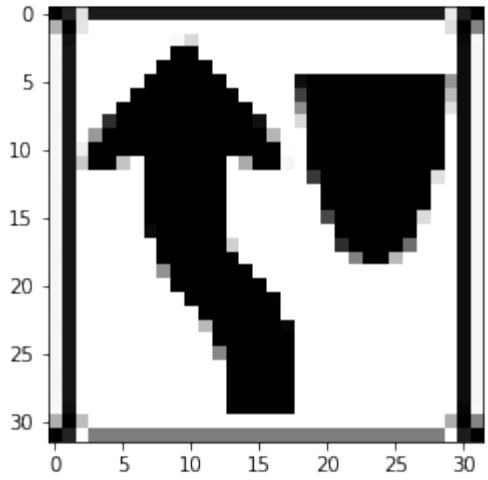



```
Top five:  TopKV2(values=array([[  9.31641877e-01,   6.83574528e-02,   6.23555081e-07,
          1.02306319e-09,   8.05791459e-18]], dtype=float32), indices=array([[17, 14, 25,  9,  1]]))
```

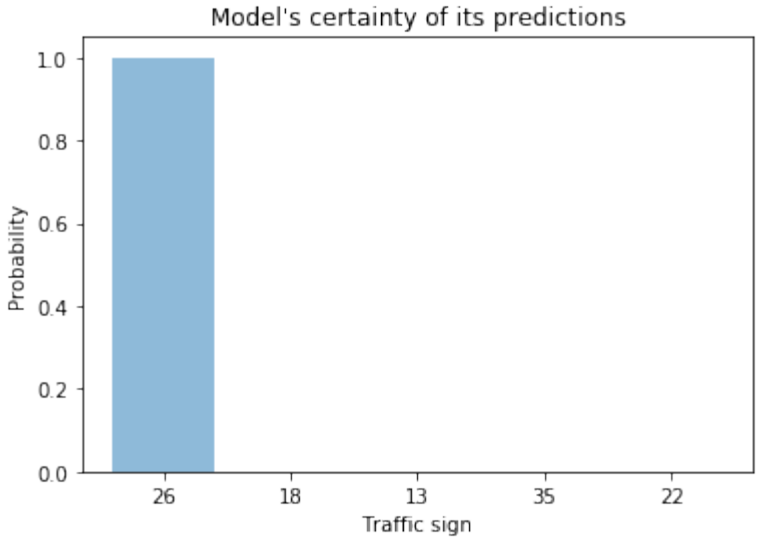



```
Traffic Sign Key
17 :  No entry
14 :  Stop
25 :  Road work
9 :  No passing
1 :  Speed limit (30km/h)
```

```
keepleft.png
```

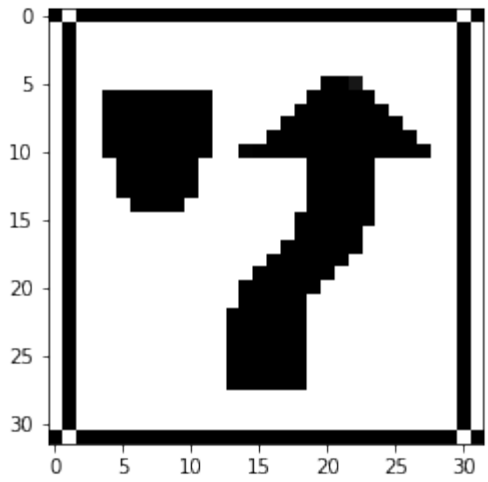



```
Top five:  TopKV2(values=array([[  1.00000000e+00,   2.89994908e-08,   4.59864959e-
11,
          1.59074845e-19,   1.87632279e-31]], dtype=float32), indices=array([[26, 18,
13, 35, 22]]))
```

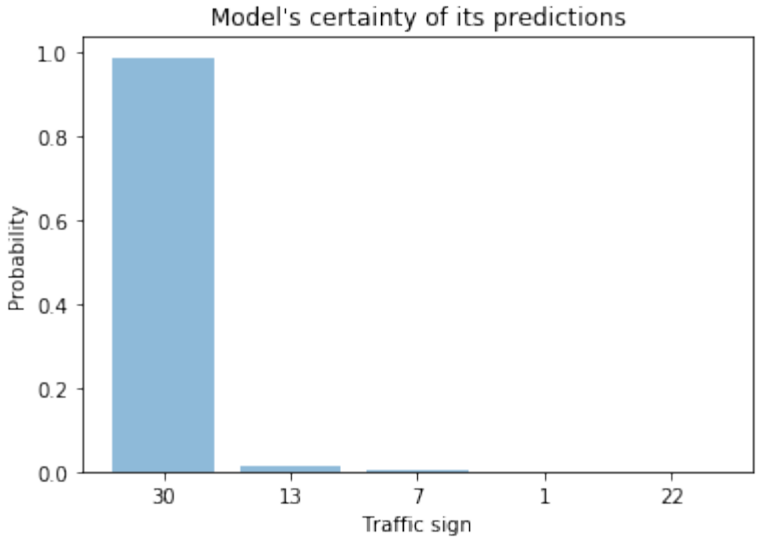



```
Traffic Sign Key
26 :  Traffic signals
18 :  General caution
13 :  Yield
35 :  Ahead only
22 :  Bumpy road
```

```
keepright.png
```

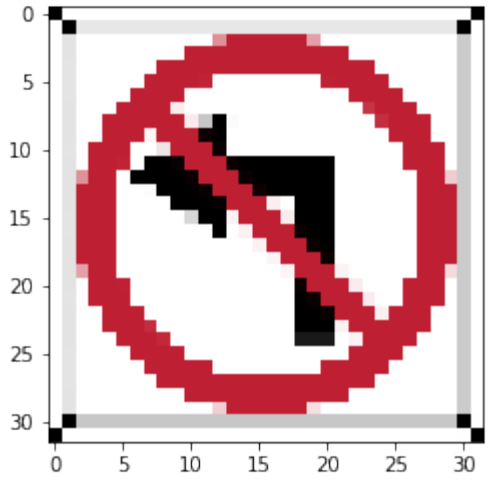



```
Top five:  TopKV2(values=array([[  9.85710144e-01,   1.26199946e-02,   1.64768752e-
03,
          1.17552827e-05,   7.69333565e-06]], dtype=float32), indices=array([[30, 13,
7,  1, 22]]))
```

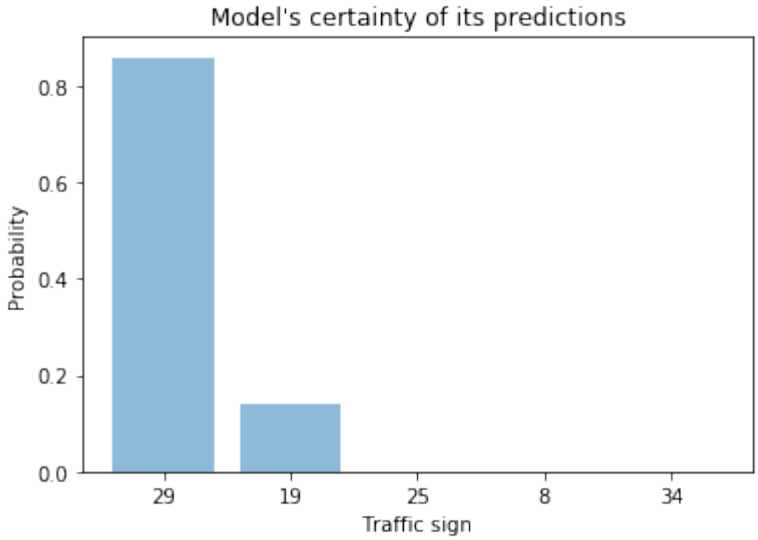



```
Traffic Sign Key
30 :  Beware of ice/snow
13 :  Yield
7 :  Speed limit (100km/h)
1 :  Speed limit (30km/h)
22 :  Bumpy road
```

```
noleft.png
```

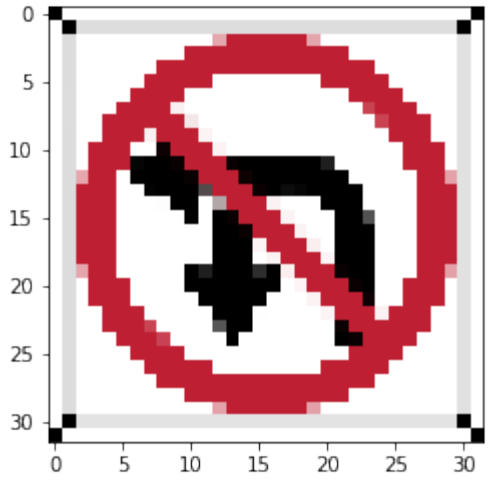


```
Top five:  TopKV2(values=array([[  8.59790504e-01,   1.40073180e-01,   1.08606255e-
04,
          2.41286325e-05,   3.59543537e-06]], dtype=float32), indices=array([[29, 19,
25,  8, 34]]))
```

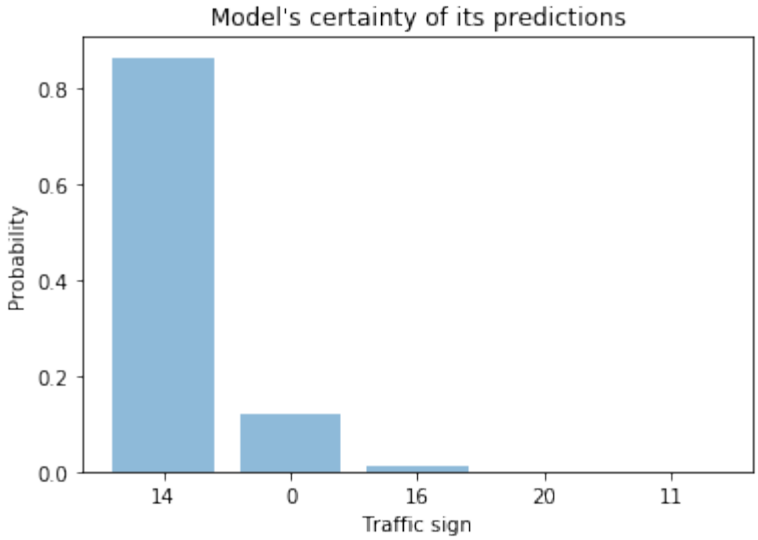



```
Traffic Sign Key
29 :  Bicycles crossing
19 :  Dangerous curve to the left
25 :  Road work
8 :  Speed limit (120km/h)
34 :  Turn left ahead
```

```
noleftUturn.png
```

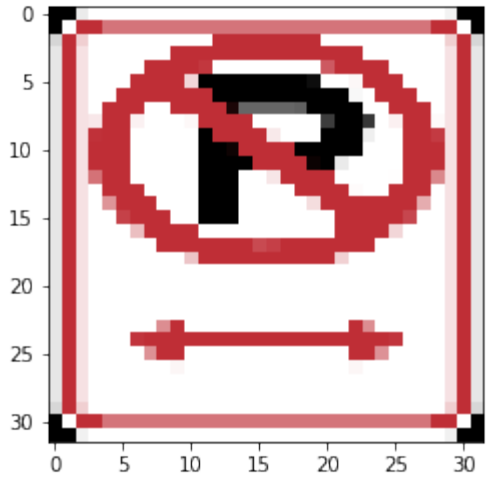



```
Top five:  TopKV2(values=array([[  8.65572274e-01,   1.22248799e-01,   1.21747050e-
02,
          3.75544801e-06,   3.02048818e-07]], dtype=float32), indices=array([[14,  0,
16, 20, 11]]))
```

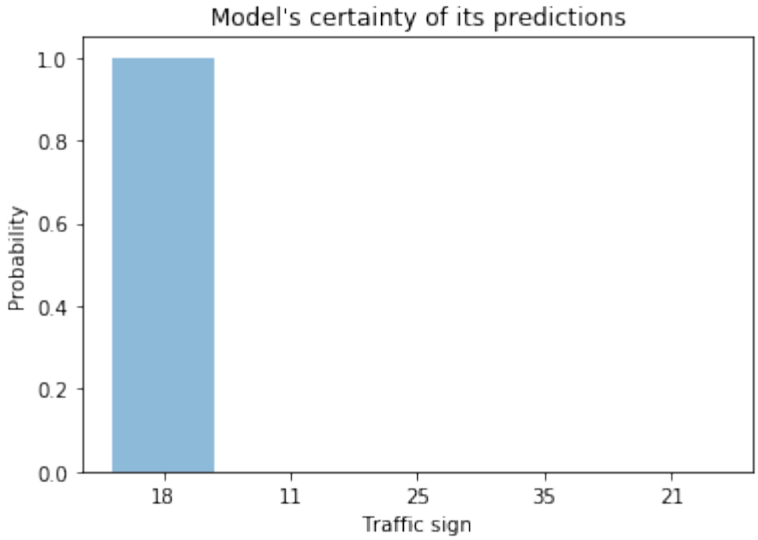



```
Traffic Sign Key
14 :  Stop
0 :  Speed limit (20km/h)
16 :  Vehicles over 3.5 metric tons prohibited
20 :  Dangerous curve to the right
11 :  Right-of-way at the next intersection
```

```
NoParking.png
```

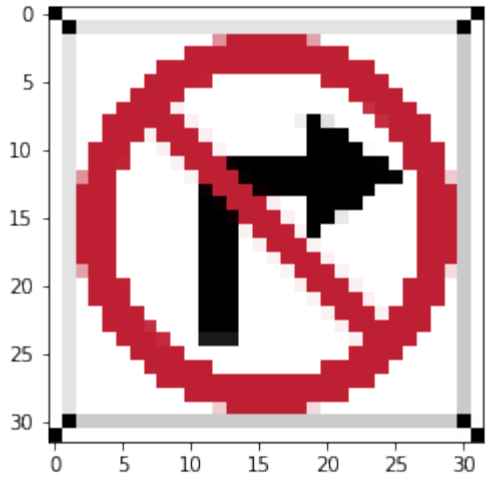


```
Top five:  TopKV2(values=array([[  9.99998808e-01,   1.16614706e-06,   1.52146430e-
14,
          7.06774720e-16,   6.13336249e-21]], dtype=float32), indices=array([[18, 11,
25, 35, 21]]))
```

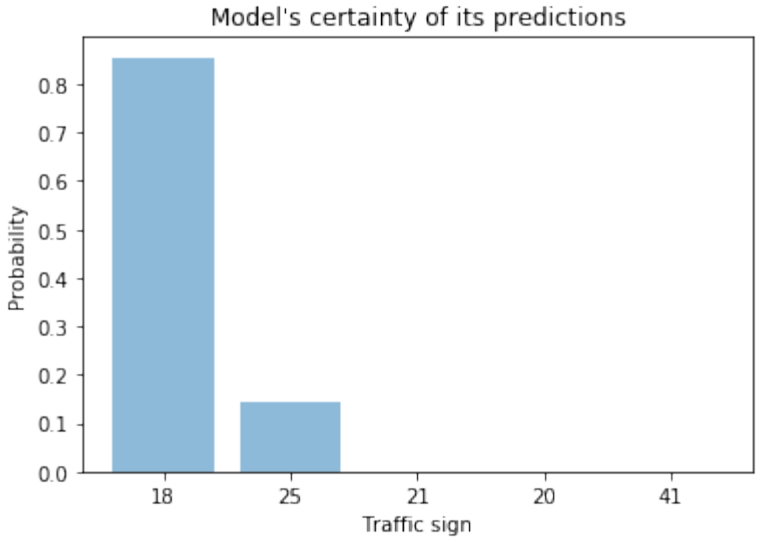



```
Traffic Sign Key
18 :  General caution
11 :  Right-of-way at the next intersection
25 :  Road work
35 :  Ahead only
21 :  Double curve
```

noright.png

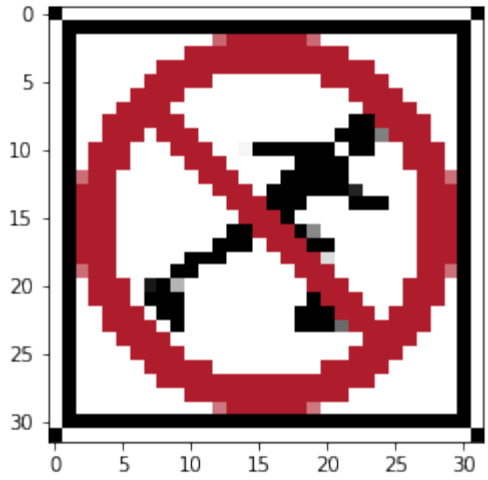



```
Top five:  TopKV2(values=array([[  8.54965270e-01,   1.43706620e-01,   1.20997732e-
03,
          8.99951337e-05,   2.77828185e-05]], dtype=float32), indices=array([[18, 25,
21, 20, 41]]))
```

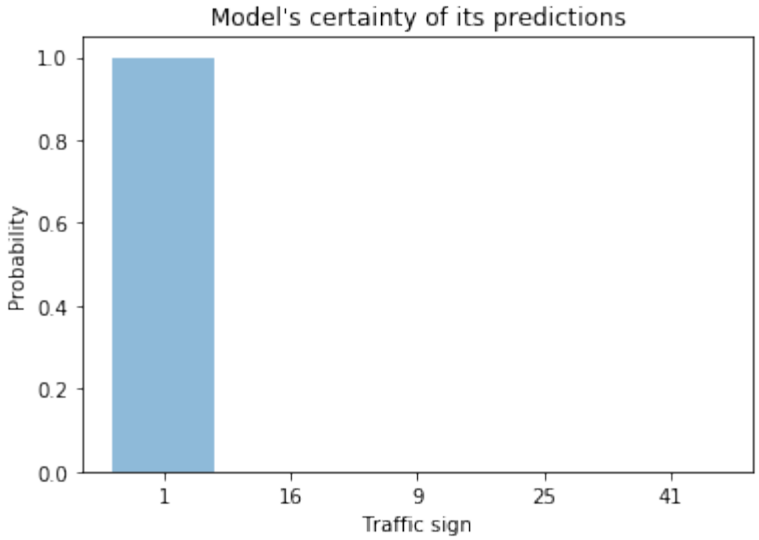



```
Traffic Sign Key
18 :  General caution
25 :  Road work
21 :  Double curve
20 :  Dangerous curve to the right
41 :  End of no passing
```

noRoller.png

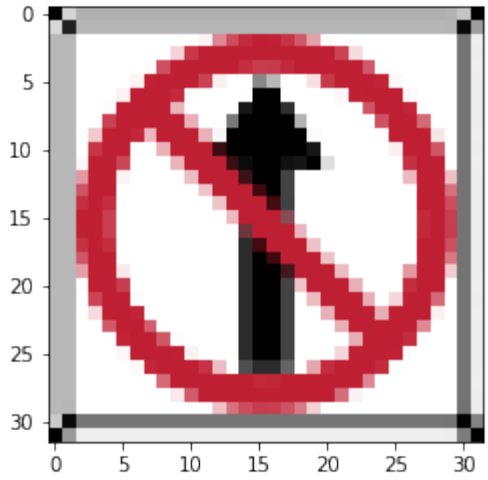



```
Top five:  TopKV2(values=array([[  9.98967528e-01,   1.03249855e-03,   6.14924292e-11,
          8.42476175e-18,   4.34662201e-23]], dtype=float32), indices=array([[ 1, 16, 9, 25, 41]]))
```

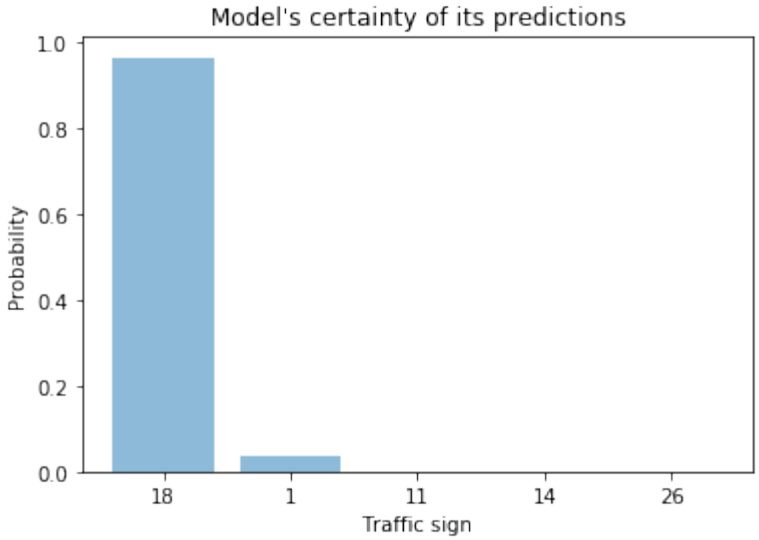



```
Traffic Sign Key
1 :  Speed limit (30km/h)
16 :  Vehicles over 3.5 metric tons prohibited
9 :  No passing
25 :  Road work
41 :  End of no passing
```

```
nostraight.png
```

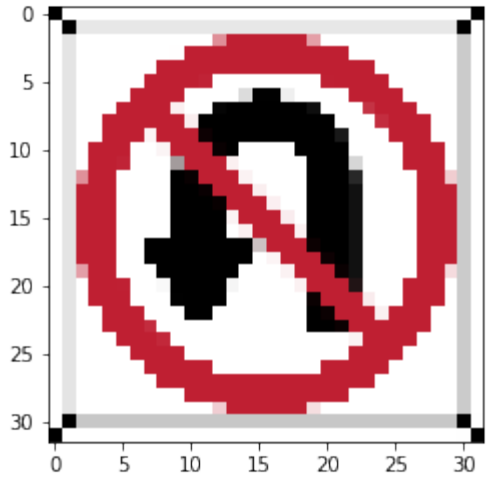



```
Top five:  TopKV2(values=array([[  9.63583469e-01,   3.60164940e-02,   2.96370679e-
04,
          1.03270388e-04,   2.97301000e-07]], dtype=float32), indices=array([[18,  1,
11, 14, 26]]))
```

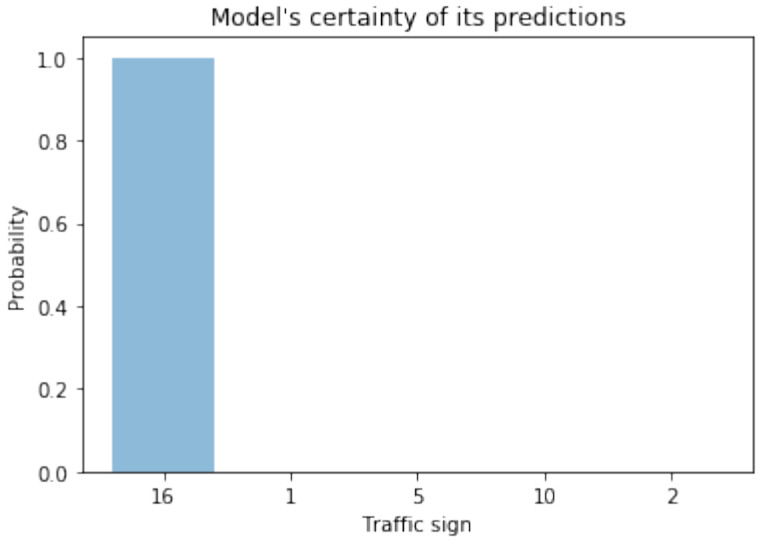



```
Traffic Sign Key
18 :  General caution
1 :  Speed limit (30km/h)
11 :  Right-of-way at the next intersection
14 :  Stop
26 :  Traffic signals
```

```
noUturn.png
```

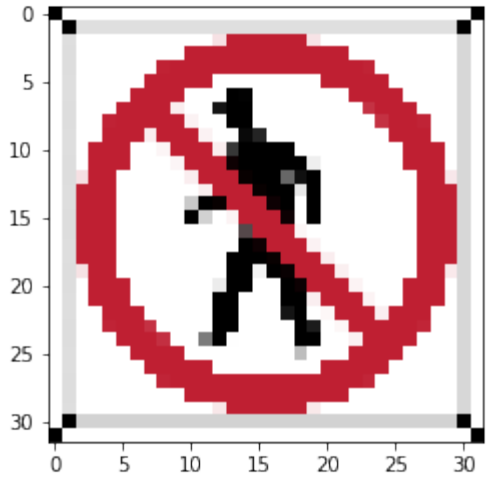


```
Top five:  TopKV2(values=array([[  9.99892592e-01,   8.52377416e-05,   1.33486665e-
05,
          8.44372062e-06,   1.79658656e-07]], dtype=float32), indices=array([[16,  1,
5, 10,  2]]))
```

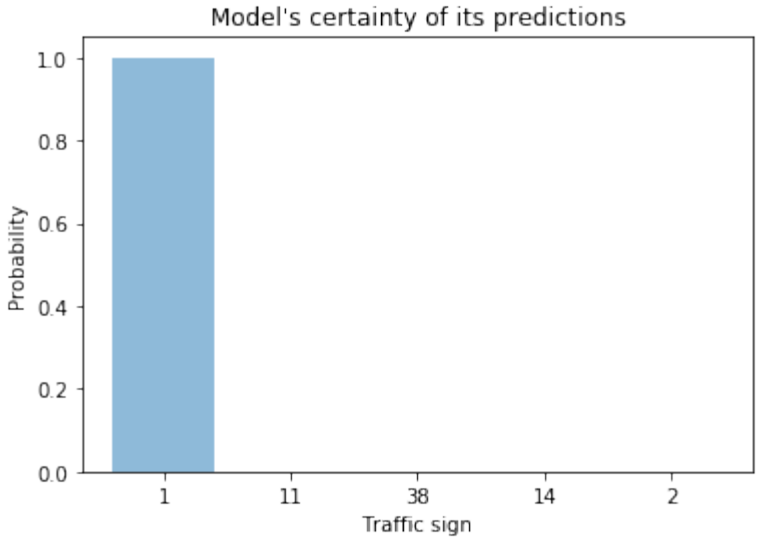



```
Traffic Sign Key
16 :  Vehicles over 3.5 metric tons prohibited
1 :  Speed limit (30km/h)
5 :  Speed limit (80km/h)
10 :  No passing for vehicles over 3.5 metric tons
2 :  Speed limit (50km/h)
```

```
noWalking.png
```

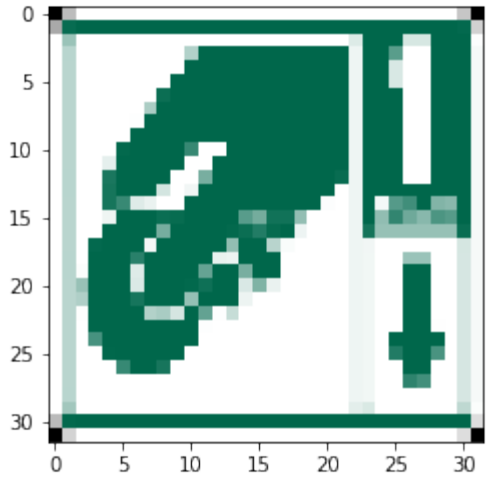


```
Top five:  TopKV2(values=array([[  1.00000000e+00,   1.07912868e-09,   9.49583412e-10,
          1.88979518e-12,   1.46053232e-12]], dtype=float32), indices=array([[ 1, 11, 38, 14,  2]]))
```

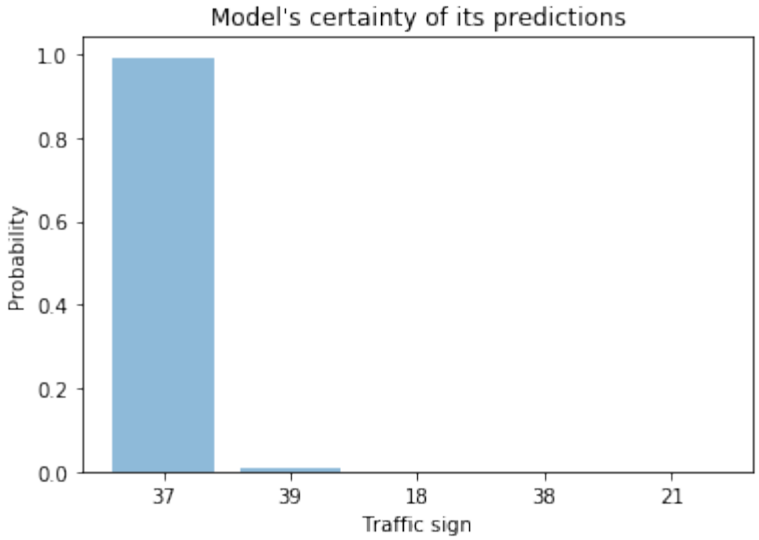



```
Traffic Sign Key
1 :  Speed limit (30km/h)
11 :  Right-of-way at the next intersection
38 :  Keep right
14 :  Stop
2 :  Speed limit (50km/h)
```

paidParking.png

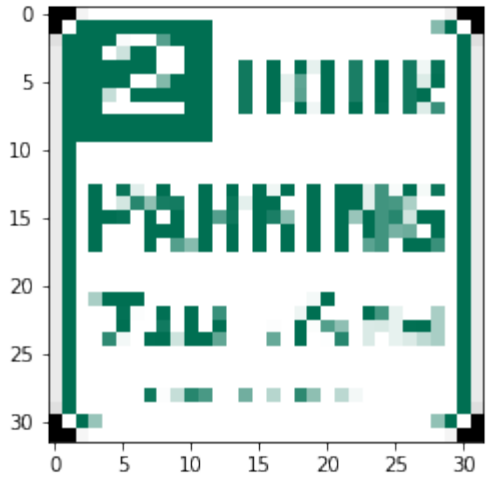



```
Top five:  TopKV2(values=array([[  9.92227435e-01,   7.76312919e-03,   5.95440270e-
06,
          3.55945826e-06,   1.03596054e-09]], dtype=float32), indices=array([[37, 39,
18, 38, 21]]))
```

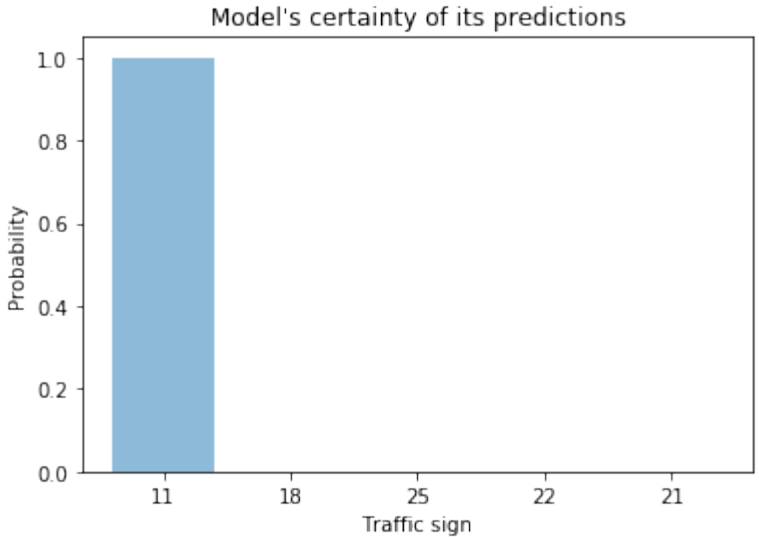



```
Traffic Sign Key
37 :  Go straight or left
39 :  Keep left
18 :  General caution
38 :  Keep right
21 :  Double curve
```

parking2hr.png

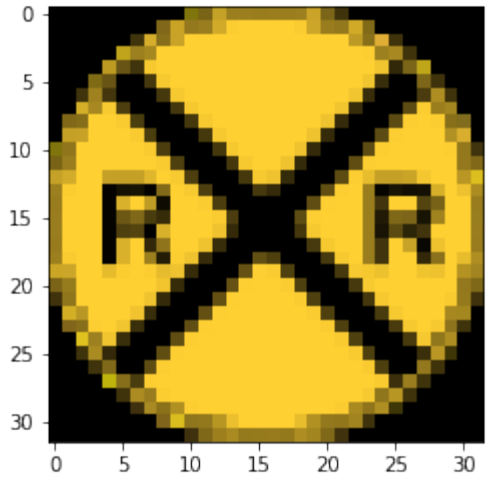



```
Top five:  TopKV2(values=array([[  9.99866128e-01,   1.33880836e-04,   2.20385602e-11,
          1.44560175e-16,   2.86904397e-20]], dtype=float32), indices=array([[11, 18, 25, 22, 21]]))
```

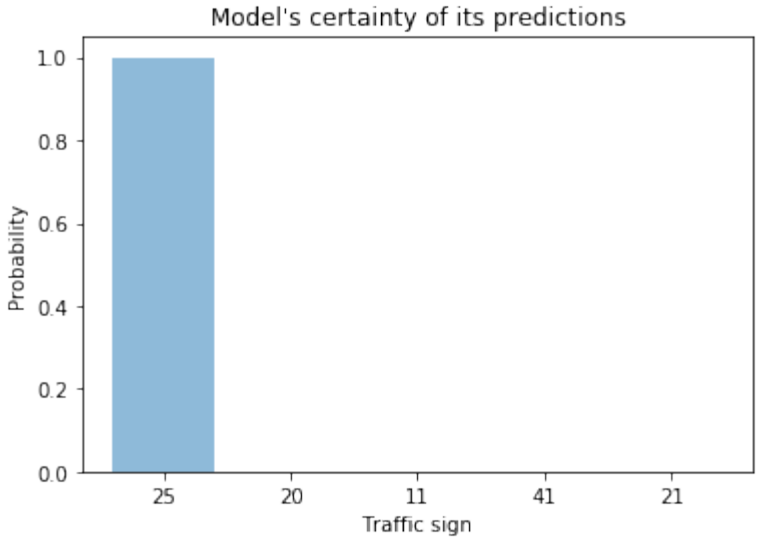



```
Traffic Sign Key
11 :  Right-of-way at the next intersection
18 :  General caution
25 :  Road work
22 :  Bumpy road
21 :  Double curve
```

```
RailRoad.png
```

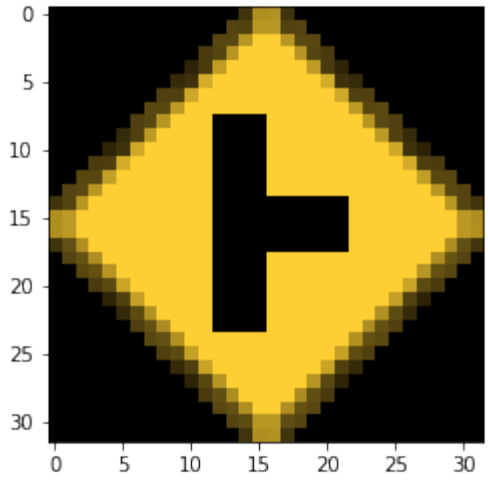



```
Top five:  TopKV2(values=array([[  9.99696136e-01,   3.03885230e-04,   2.96920578e-
11,
          4.97436073e-19,   8.93556428e-20]], dtype=float32), indices=array([[25, 20,
11, 41, 21]]))
```

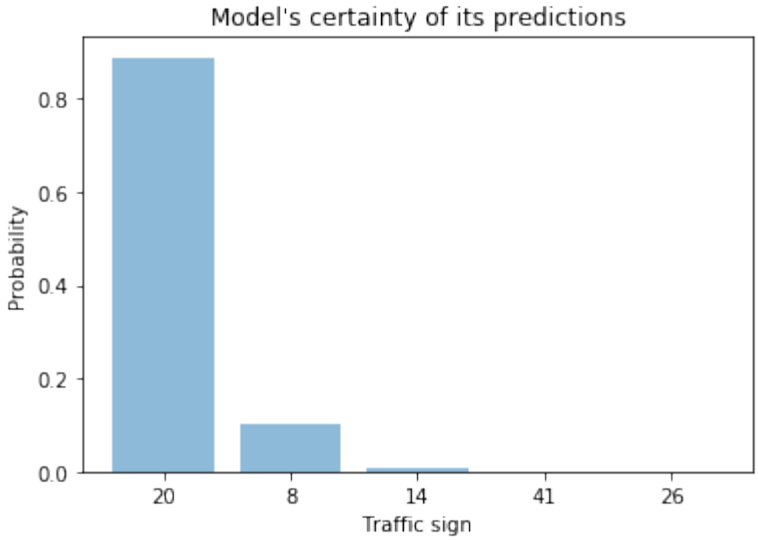



```
Traffic Sign Key
25 :  Road work
20 :  Dangerous curve to the right
11 :  Right-of-way at the next intersection
41 :  End of no passing
21 :  Double curve
```

```
RightCrossing.png
```

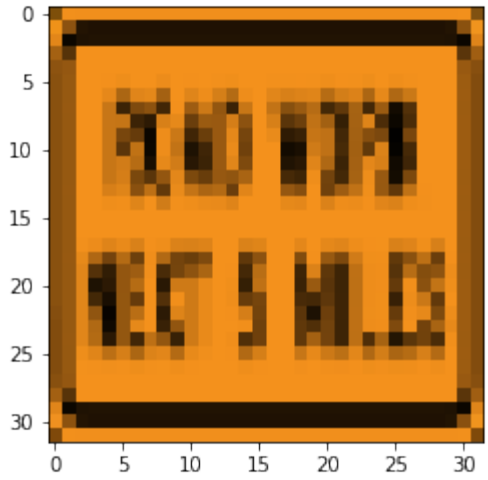



```
Top five:  TopKV2(values=array([[  8.89002383e-01,   1.01664796e-01,   9.33088269e-
03,
          1.86784382e-06,   3.12237738e-08]], dtype=float32), indices=array([[20,  8,
14, 41, 26]]))
```

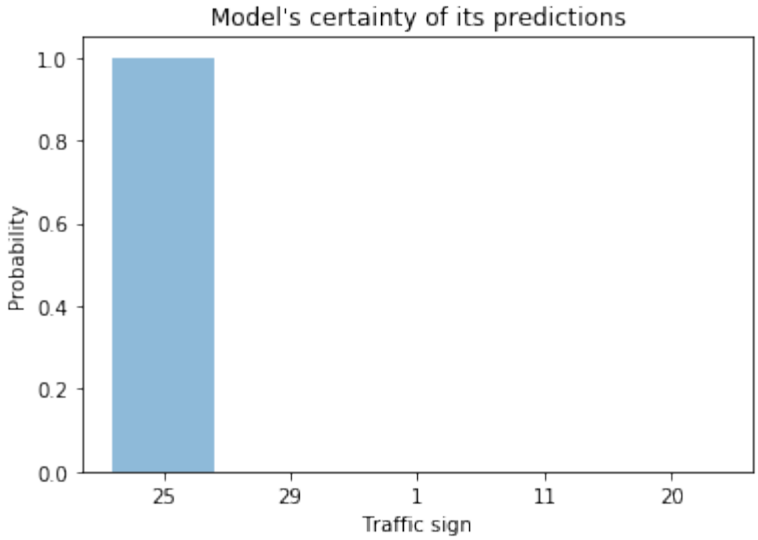



```
Traffic Sign Key
20 :  Dangerous curve to the right
8 :  Speed limit (120km/h)
14 :  Stop
41 :  End of no passing
26 :  Traffic signals
```

```
roadwork.png
```

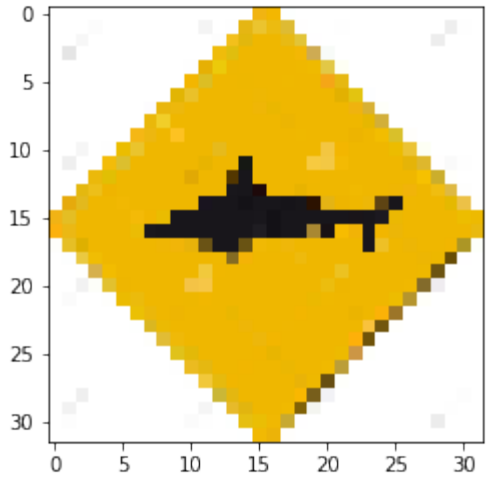



```
Top five:  TopKV2(values=array([[  9.99986649e-01,   1.33217263e-05,   4.55496033e-
14,
          6.06305421e-15,   4.63272034e-22]], dtype=float32), indices=array([[25, 29,
1, 11, 20]]))
```

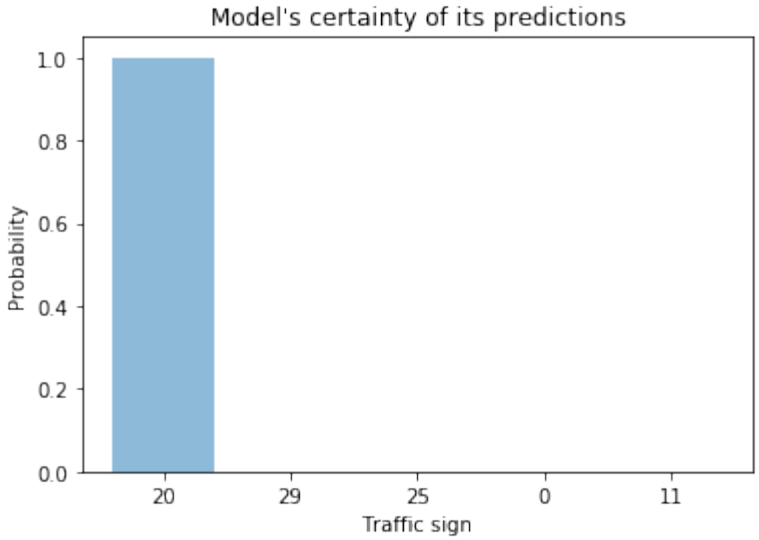



```
Traffic Sign Key
25 :  Road work
29 :  Bicycles crossing
1 :  Speed limit (30km/h)
11 :  Right-of-way at the next intersection
20 :  Dangerous curve to the right
```

```
shark_sign.png
```

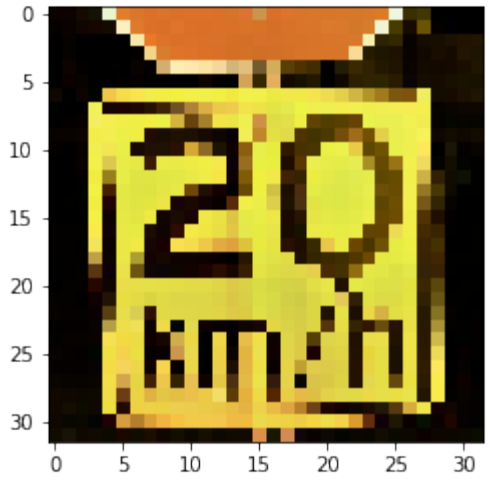



```
Top five:  TopKV2(values=array([[  1.00000000e+00,   2.92875021e-18,   1.11846080e-
23,
          3.85432385e-26,   4.66108342e-33]], dtype=float32), indices=array([[20, 29,
25,  0, 11]]))
```

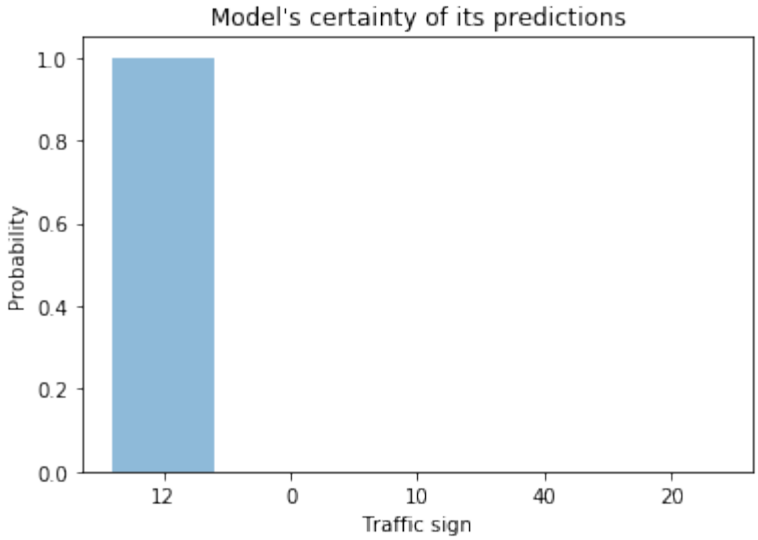



```
Traffic Sign Key
20 :  Dangerous curve to the right
29 :  Bicycles crossing
25 :  Road work
0 :  Speed limit (20km/h)
11 :  Right-of-way at the next intersection
```

```
speed_limit_stop.png
```

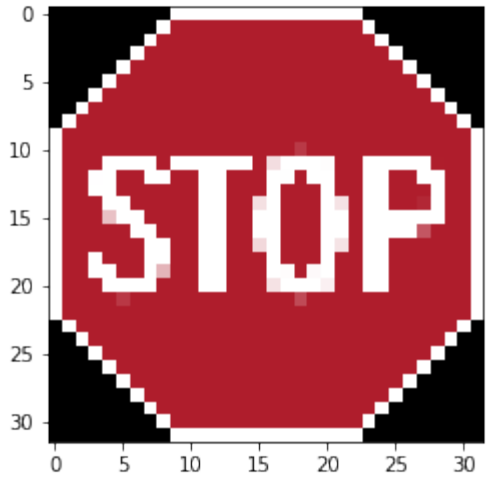



```
Top five:  TopKV2(values=array([[  9.99999285e-01,   5.49591334e-07,   1.83094855e-
07,
          1.06923874e-08,   2.48394882e-09]], dtype=float32), indices=array([[12,  0,
10, 40, 20]]))
```

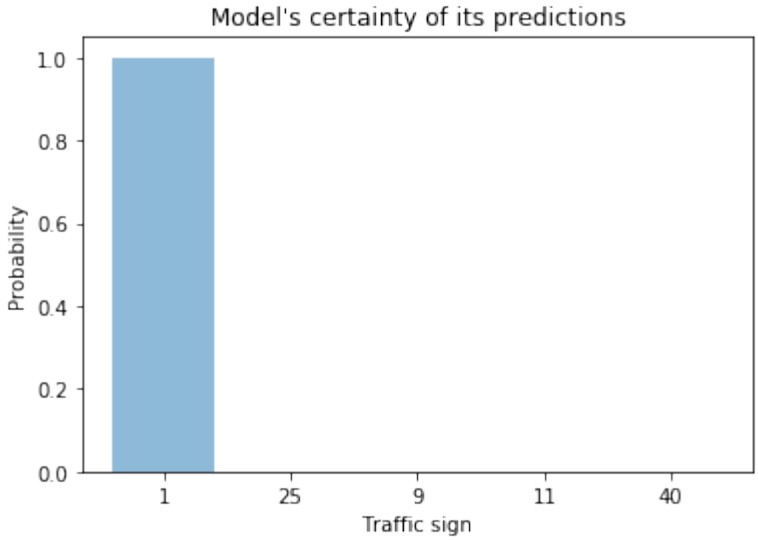



```
Traffic Sign Key
12 :  Priority road
0 :  Speed limit (20km/h)
10 :  No passing for vehicles over 3.5 metric tons
40 :  Roundabout mandatory
20 :  Dangerous curve to the right
```

```
stop.png
```

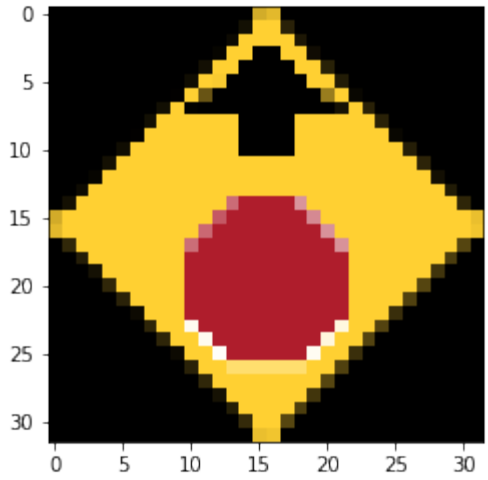



```
Top five:  TopKV2(values=array([[  1.00000000e+00,   1.59572325e-15,   8.96946990e-19,
          7.52545839e-22,   1.30733259e-25]], dtype=float32), indices=array([[ 1, 25, 9, 11, 40]]))
```

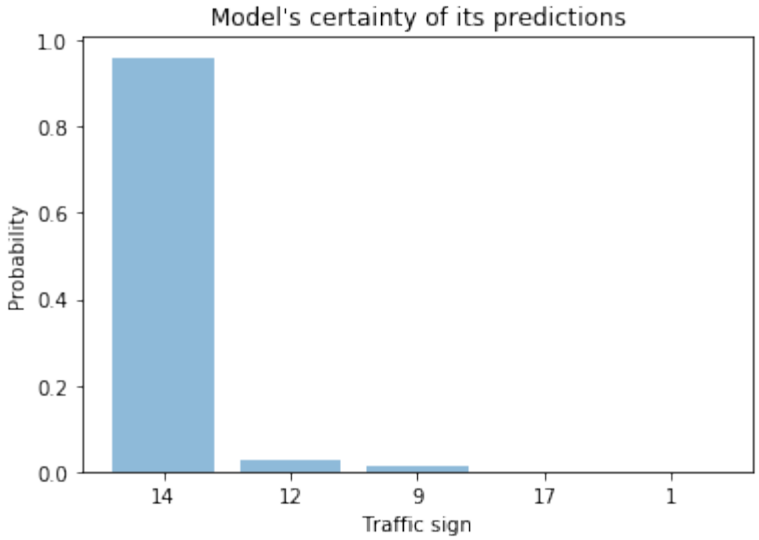



```
Traffic Sign Key
1 :  Speed limit (30km/h)
25 :  Road work
9 :  No passing
11 :  Right-of-way at the next intersection
40 :  Roundabout mandatory
```

StopAhead.png

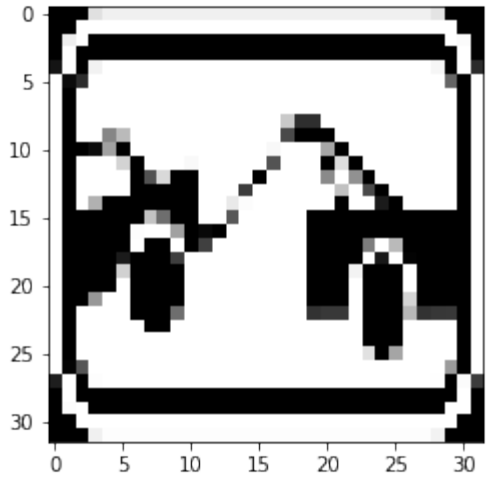



```
Top five:  TopKV2(values=array([[  9.60852146e-01,   2.60129459e-02,   1.29385469e-
02,
          1.96388020e-04,   3.94786027e-12]], dtype=float32), indices=array([[14, 12,
9, 17,  1]]))
```

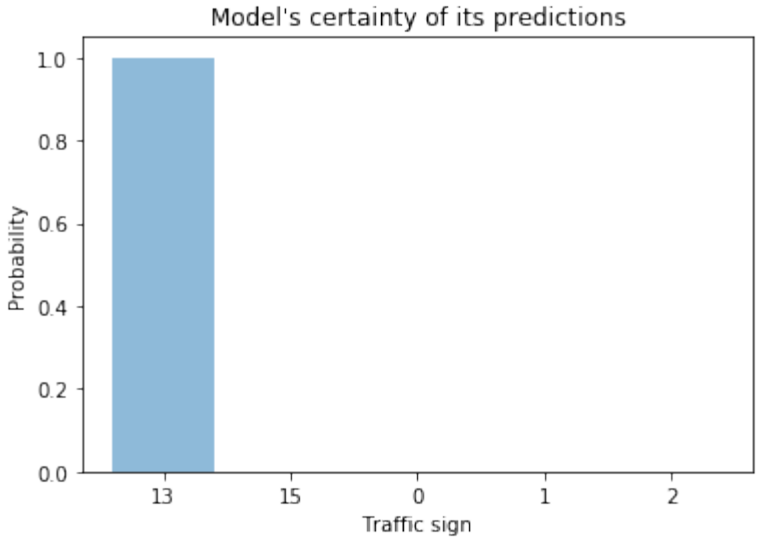



```
Traffic Sign Key
14 :  Stop
12 :  Priority road
9 :  No passing
17 :  No entry
1 :  Speed limit (30km/h)
```

TowAway.png

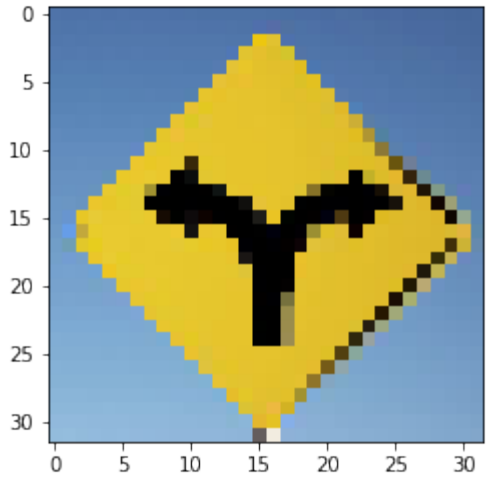



```
Top five:  TopKV2(values=array([[  1.00000000e+00,   6.32482721e-37,
0.00000000e+00,
          0.00000000e+00,   0.00000000e+00]], dtype=float32), indices=array([[13, 15,
0,  1,  2]]))
```

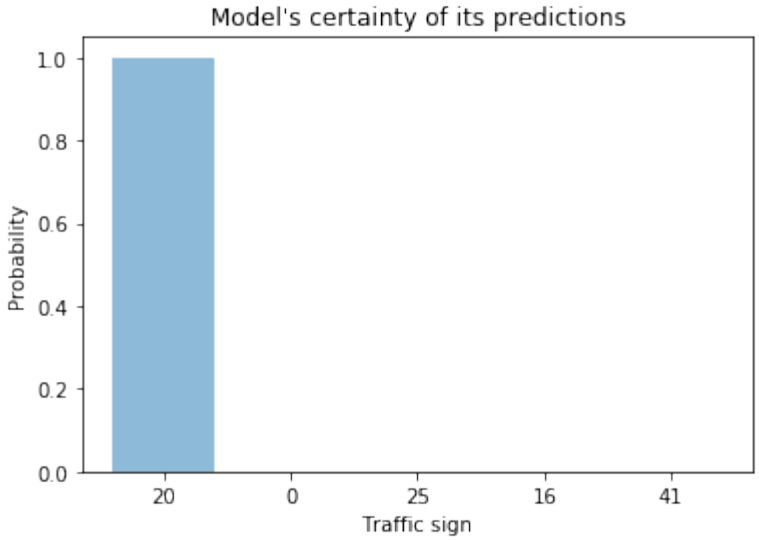



```
Traffic Sign Key
13 :  Yield
15 :  No vehicles
0 :  Speed limit (20km/h)
1 :  Speed limit (30km/h)
2 :  Speed limit (50km/h)
```

two_way_sign.png

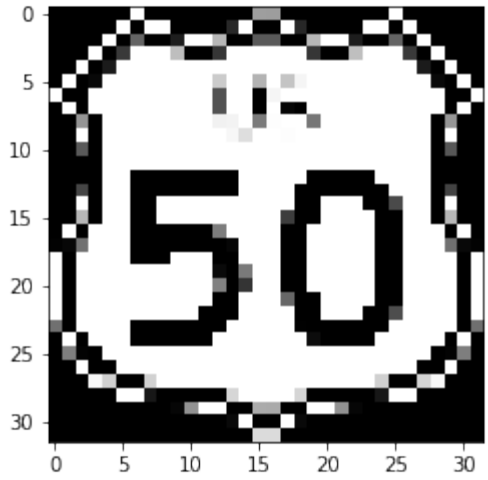


```
Top five:  TopKV2(values=array([[  1.00000000e+00,   3.43044402e-21,   1.90769681e-
23,
          3.68719504e-24,   7.46995626e-26]], dtype=float32), indices=array([[20,  0,
25, 16, 41]]))
```

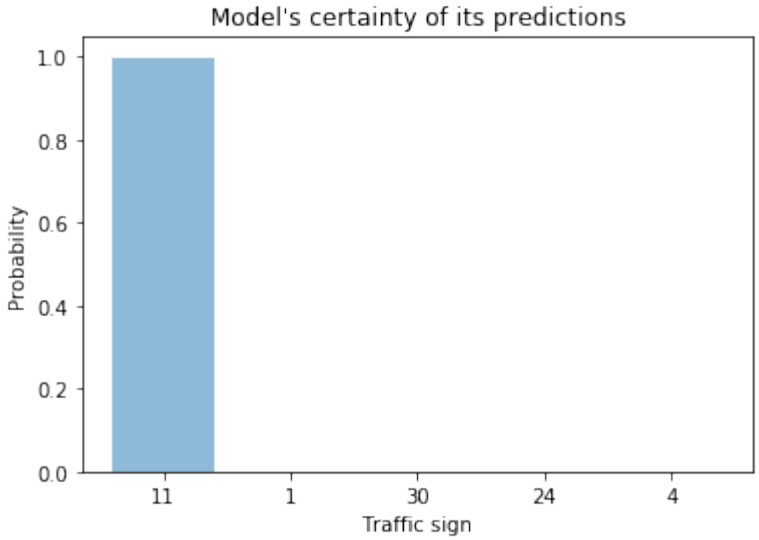



```
Traffic Sign Key
20 :  Dangerous curve to the right
0 :  Speed limit (20km/h)
25 :  Road work
16 :  Vehicles over 3.5 metric tons prohibited
41 :  End of no passing
```

US50.png

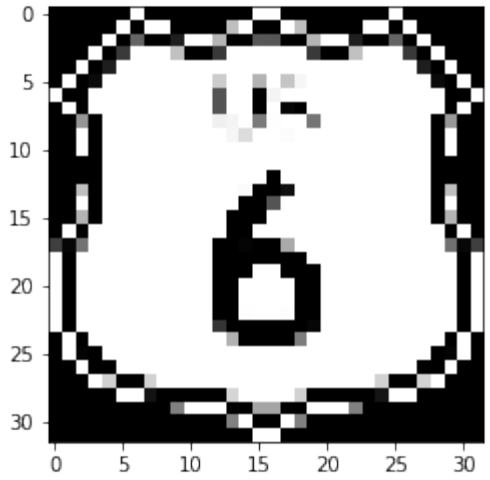


```
Top five:  TopKV2(values=array([[  9.98767912e-01,   1.00342918e-03,   2.28566598e-04,
          1.18376342e-11,   2.48309132e-13]], dtype=float32), indices=array([[11,  1, 30, 24,  4]]))
```

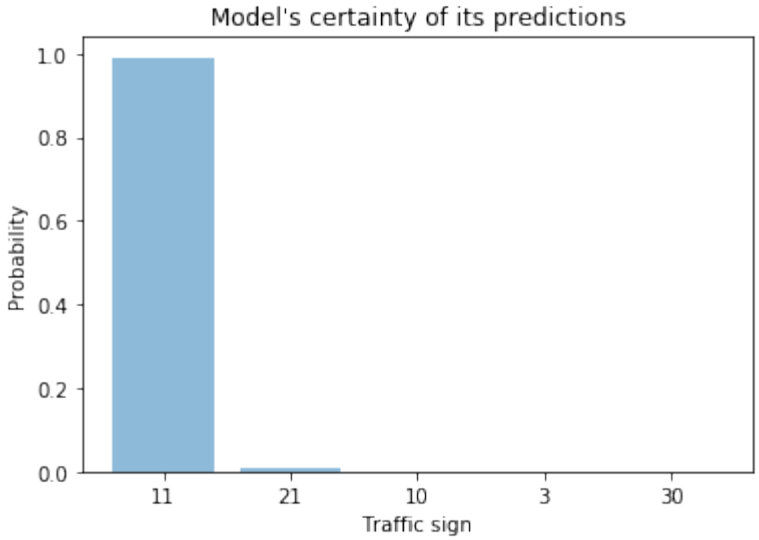



```
Traffic Sign Key
11 :  Right-of-way at the next intersection
1 :  Speed limit (30km/h)
30 :  Beware of ice/snow
24 :  Road narrows on the right
4 :  Speed limit (70km/h)
```

```
US6.png
```

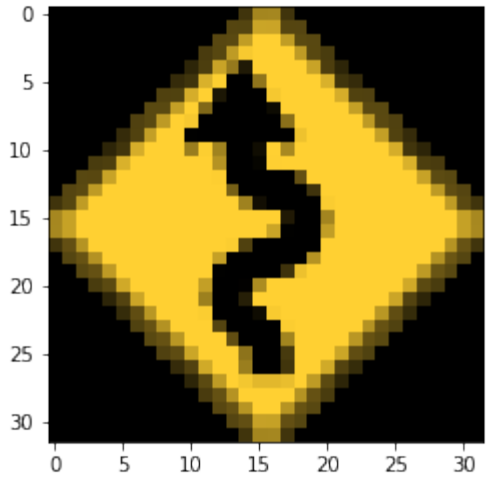


```
Top five:  TopKV2(values=array([[  9.91104007e-01,   8.89599230e-03,   1.25855781e-
09,
          2.04973061e-12,   1.93882263e-15]], dtype=float32), indices=array([[11, 21,
10,  3, 30]]))
```

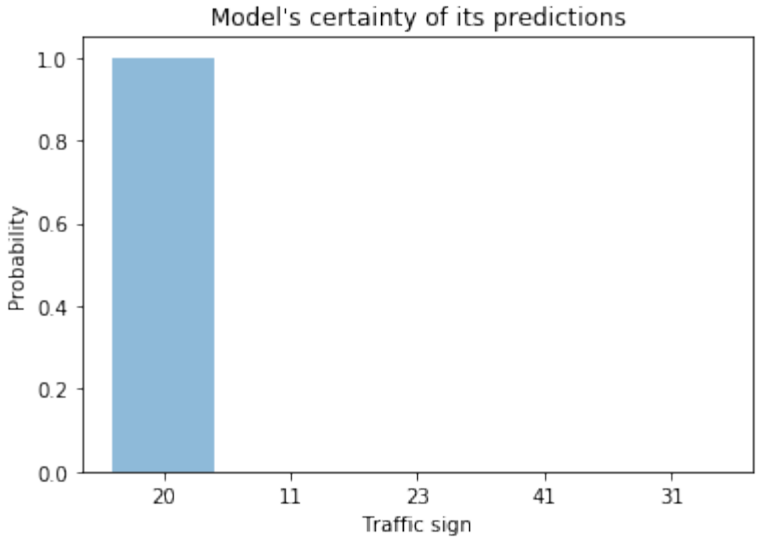



```
Traffic Sign Key
11 :  Right-of-way at the next intersection
21 :  Double curve
10 :  No passing for vehicles over 3.5 metric tons
3 :  Speed limit (60km/h)
30 :  Beware of ice/snow
```

```
windyRight.png
```

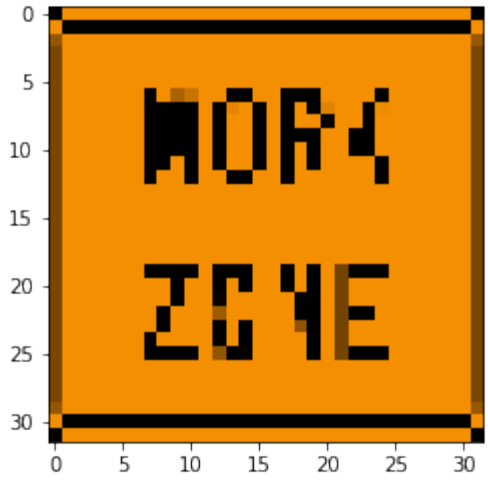


```
Top five:  TopKV2(values=array([[  9.99993920e-01,   6.12419126e-06,   3.21461989e-
11,
          2.85378180e-18,   2.52966121e-18]], dtype=float32), indices=array([[20, 11,
23, 41, 31]]))
```

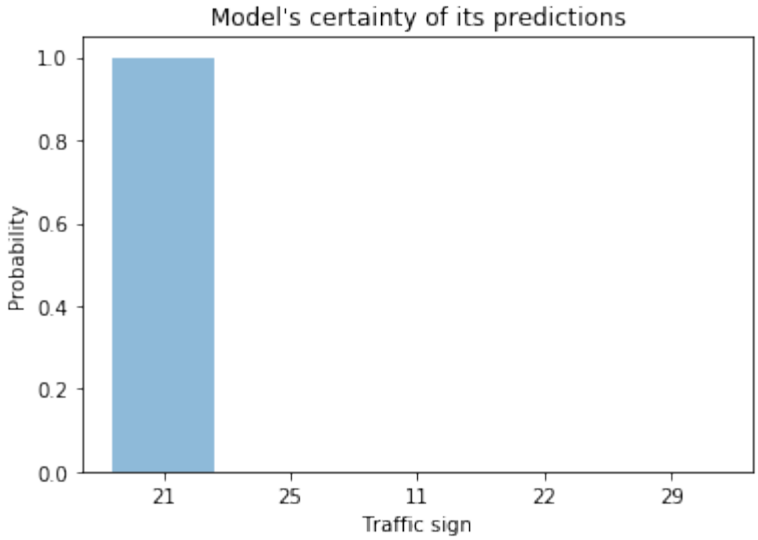



```
Traffic Sign Key
20 :  Dangerous curve to the right
11 :  Right-of-way at the next intersection
23 :  Slippery road
41 :  End of no passing
31 :  Wild animals crossing
```

workzone.png

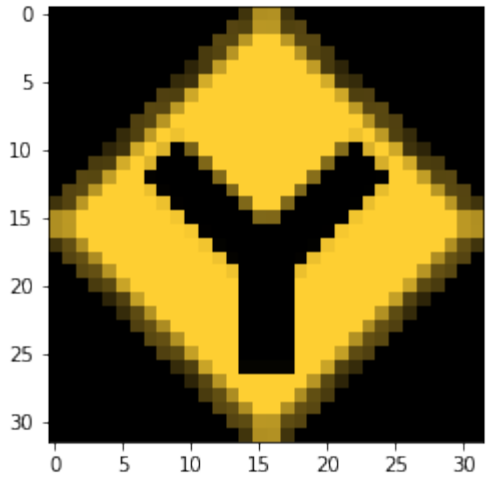



```
Top five:  TopKV2(values=array([[  9.99693274e-01,   3.06694506e-04,   5.39248646e-09,
          9.11828165e-12,   4.05209347e-14]], dtype=float32), indices=array([[21, 25, 11, 22, 29]]))
```

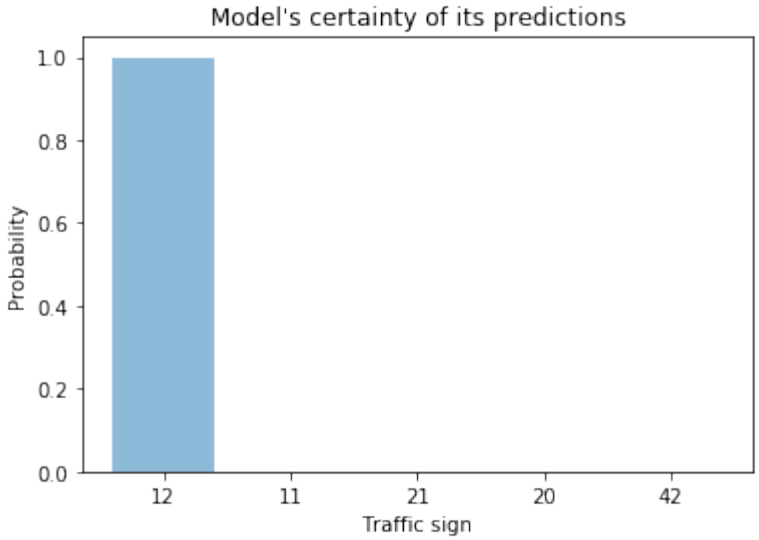



```
Traffic Sign Key
21 :  Double curve
25 :  Road work
11 :  Right-of-way at the next intersection
22 :  Bumpy road
29 :  Bicycles crossing
```

```
YCrossing.png
```

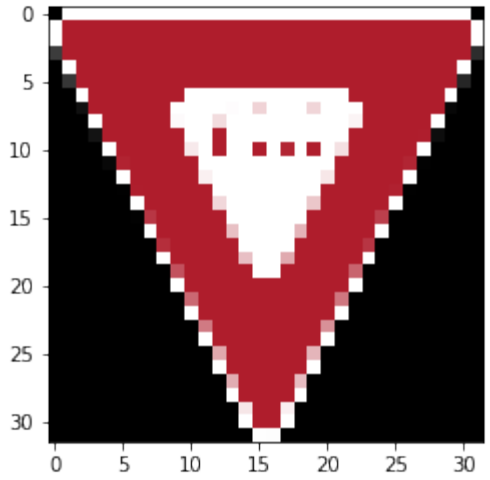



```
Top five:  TopKV2(values=array([[  9.99373615e-01,   4.02835110e-04,   2.23572220e-
04,
          7.45435802e-09,   5.34850519e-09]], dtype=float32), indices=array([[12, 11,
21, 20, 42]]))
```

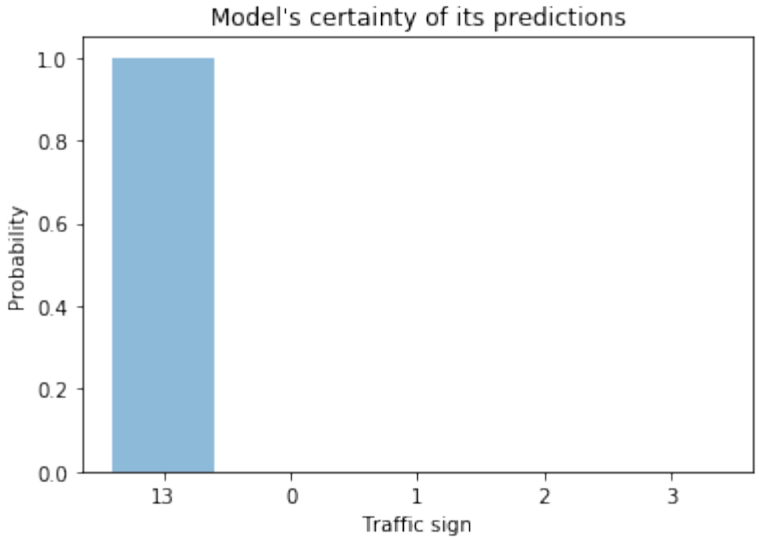



```
Traffic Sign Key
12 :  Priority road
11 :  Right-of-way at the next intersection
21 :  Double curve
20 :  Dangerous curve to the right
42 :  End of no passing by vehicles over 3.5 metric tons
```

yeild.png

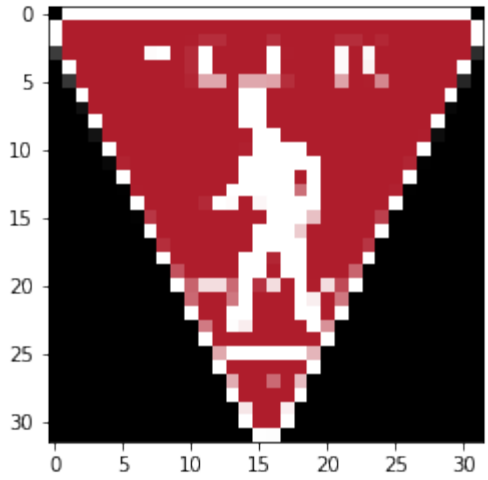



```
Top five:  TopKV2(values=array([[ 1.,  0.,  0.,  0.,  0.]], dtype=float32),
indices=array([[13,  0,  1,  2,  3]]))
```

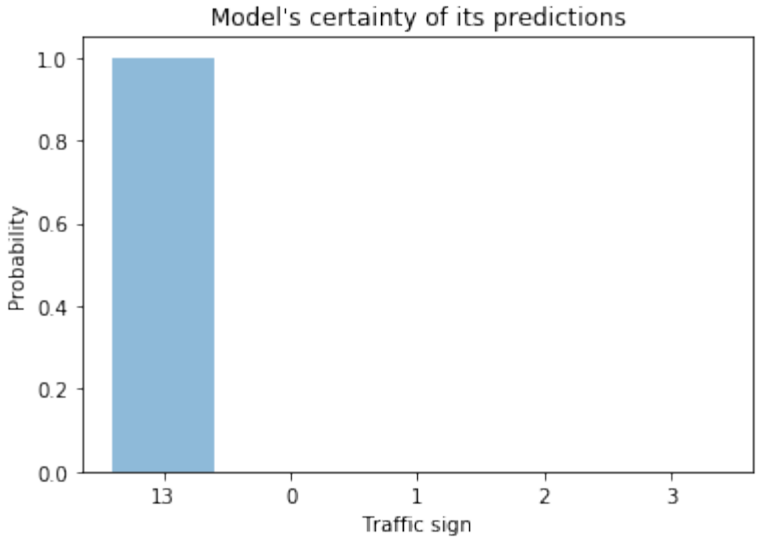



```
Traffic Sign Key
13 :  Yield
0 :  Speed limit (20km/h)
1 :  Speed limit (30km/h)
2 :  Speed limit (50km/h)
3 :  Speed limit (60km/h)
```

yield_pedestrian.png




```
Top five:  TopKV2(values=array([[ 1.,  0.,  0.,  0.,  0.]], dtype=float32),
indices=array([[13,  0,  1,  2,  3]]))
```




```
Traffic Sign Key
13 :  Yield
0 :  Speed limit (20km/h)
1 :  Speed limit (30km/h)
2 :  Speed limit (50km/h)
3 :  Speed limit (60km/h)
```

Special thanks to Jassica young and Ricardo Calix Mr Ricardo Calix who's book "Getting Started with Deep learning" was a Great help. This assignment was not possible without this book.

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```