Traffic Signs Recognition using Convolutional Neural Network (CNN)

This document describes the Python script used to generate and train a CNN model that is capable of recognising German road traffic signs with close to 97% accuracy.

First, we import the relevant modules. The most important one to note here is the Keras library, a high-level, deep learning API developed by Google for easy implementation of neural networks.

```
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Dense
from keras.layers import Flatten, Dropout
from keras.utils.np_utils import to_categorical
from keras.layers.convolutional import Conv2D, MaxPooling2D
import random
import pickle
import pandas as pd
import cv2

from keras.callbacks import LearningRateScheduler, ModelCheckpoint
%matplotlib inline
```

```
[4] np.random.seed(0)
```

Next, we clone the data available on bitbucket into google drive so that we can load the datasets.

```
[1] !git clone https://bitbucket.org/jadslim/german-traffic-signs

Cloning into 'german-traffic-signs'...
Unpacking objects: 100% (6/6), done.
```

```
[2] !ls german-traffic-signs

signnames.csv  test.p  train.p  valid.p
```

We can now load the training, validation, and testing datasets. Because the datasets were saved in pickle format, we need to read the file in binary format ('rb') and use the Pickle module to load the file.

Once we have done that, we can save the features and labels into different variables and check that the shapes of the variables are correct.

We can also read the signnames.csv using Pandas library – read_csv() method and save the dataframe into a variable called **data.**

```
[5] #load the data
    with open('german-traffic-signs/train.p', 'rb') as f:
        train_data = pickle.load(f)
    with open('german-traffic-signs/valid.p', 'rb') as f:
        val_data = pickle.load(f)
    #load test data
    with open('german-traffic-signs/test.p', 'rb') as f:
        test_data = pickle.load(f)

    print(type(train_data))

    <class 'dict'>
```

```
[6] # Split out features and labels
    X_train, y_train = train_data['features'], train_data['labels']
    X_val, y_val = val_data['features'], val_data['labels']
    X_test, y_test = test_data['features'], test_data['labels']
```

```
[7] #4 dimensional (rgb data)
    print(X_train.shape)
    print(X_val.shape)
    print(X_test.shape)

    #check if shape of data is as expected
    assert(X_train.shape[0] == y_train.shape[0]), "The number of images is not equal to the number of labels."
    assert(X_train.shape[1:] == (32,32,3)), "The dimensions of the images are not 32 x 32 x 3."
    assert(X_val.shape[0] == y_val.shape[0]), "The number of images is not equal to the number of labels."
    assert(X_val.shape[1:] == (32,32,3)), "The dimensions of the images are not 32 x 32 x 3."
    assert(X_test.shape[0] == y_test.shape[0]), "The number of images is not equal to the number of labels."
    assert(X_test.shape[1:] == (32,32,3)), "The dimensions of the images are not 32 x 32 x 3."
    data = pd.read_csv('german-traffic-signs/signnames.csv')

    (34799, 32, 32, 3)
    (4410, 32, 32, 3)
    (12630, 32, 32, 3)
```

Now, let's print out the dataframe. We can see that the dataframe consists of 2 fields, first being the index of the traffic sign and second being the name of the sign. There are a total of **43 traffic sign categories**.

```
print(data)
```

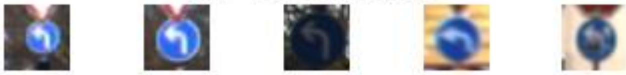
```
    ClassId                                           SignName
0         0                               Speed limit (20km/h)
1         1                               Speed limit (30km/h)
2         2                               Speed limit (50km/h)
3         3                               Speed limit (60km/h)
4         4                               Speed limit (70km/h)
5         5                               Speed limit (80km/h)
6         6                        End of speed limit (80km/h)
7         7                              Speed limit (100km/h)
8         8                              Speed limit (120km/h)
9         9                                         No passing
10       10       No passing for vechiles over 3.5 metric tons
11       11              Right-of-way at the next intersection
12       12                                      Priority road
13       13                                              Yield
14       14                                               Stop
15       15                                        No vechiles
16       16           Vechiles over 3.5 metric tons prohibited
17       17                                           No entry
18       18                                    General caution
19       19                        Dangerous curve to the left
20       20                       Dangerous curve to the right
21       21                                       Double curve
22       22                                         Bumpy road
23       23                                      Slippery road
24       24                          Road narrows on the right
25       25                                          Road work
26       26                                    Traffic signals
27       27                                        Pedestrians
28       28                                  Children crossing
29       29                                  Bicycles crossing
30       30                                 Beware of ice/snow
31       31                              Wild animals crossing
32       32                End of all speed and passing limits
33       33                                   Turn right ahead
34       34                                    Turn left ahead
35       35                                         Ahead only
36       36                               Go straight or right
37       37                                Go straight or left
38       38                                         Keep right
39       39                                          Keep left
40       40                               Roundabout mandatory
41       41                                  End of no passing
42       42  End of no passing by vechiles over 3.5 metric ...
```

For each of the category, let's randomly display 5 images so that we can better understand what our data contains.

```
[9] num_of_samples=[]

    cols = 5
    num_classes = 43

    fig, axs = plt.subplots(nrows=num_classes, ncols=cols, figsize=(5,50))
    fig.tight_layout()

    for i in range(cols):
        for j, row in data.iterrows():
            x_selected = X_train[y_train == j]
            axs[j][i].imshow(x_selected[random.randint(0,(len(x_selected) - 1)), :, :], cmap=plt.get_cmap('gray'))
            axs[j][i].axis("off")
            if i == 2:
              axs[j][i].set_title(str(j) + " - " + row["SignName"])
              num_of_samples.append(len(x_selected))
    print(num_of_samples)
    plt.figure(figsize=(12, 4))
    plt.bar(range(0, num_classes), num_of_samples)
    plt.title("Distribution of the train dataset")
    plt.xlabel("Class number")
    plt.ylabel("Number of images")
    plt.show()
```


0 - Speed limit (20km/h)
1 - Speed limit (30km/h)
2 - Speed limit (50km/h)
3 - Speed limit (60km/h)
4 - Speed limit (70km/h)
5 - Speed limit (80km/h)
6 - End of speed limit (80km/h)
7 - Speed limit (100km/h)
8 - Speed limit (120km/h)
9 - No passing

34 - Turn left ahead



35 - Ahead only

36 - Go straight or right

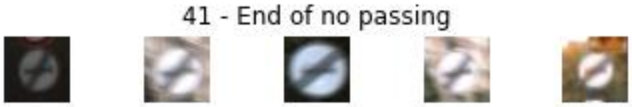
37 - Go straight or left

38 - Keep right

39 - Keep left

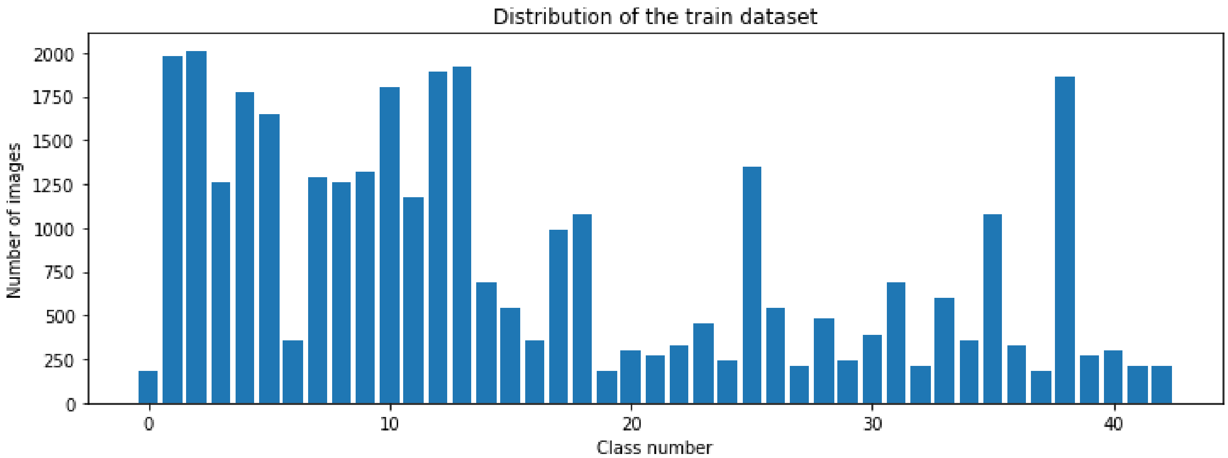
40 - Roundabout mandatory

41 - End of no passing



42 - End of no passing by vechiles over 3.5 metric tons

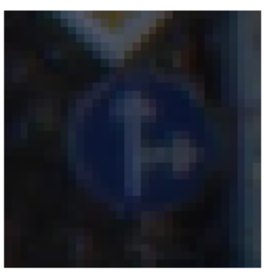
The bar chart below shows the number of images we have for each traffic sign category in our training dataset. Notice that the count is not uniform but let's just work with what we have.



Before we set up and train our model, we need to **preprocess the images** to ensure the best results. We want to **grayscale** them to simplify algorithms and reduce computational requirements.

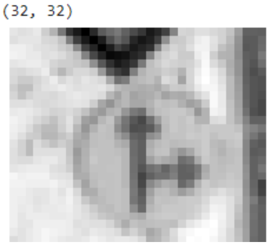
```
[10] import cv2

     plt.imshow(X_train[1000])
     plt.axis("off")
     print(X_train[1000].shape)
     print(y_train[1000])
```

```
(32, 32, 3)
36
```



```
def grayscale(img):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return img

img = grayscale(X_train[1000])
plt.imshow(img, cmap='gray_r')
plt.axis("off")
print(img.shape)
```

```
(32, 32)
```

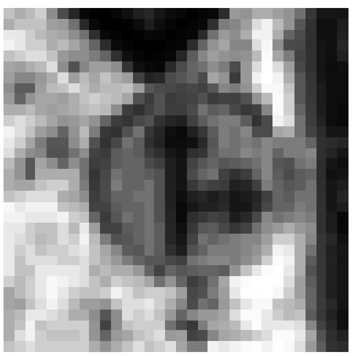


We should also use the **cv2.equalizeHist() method** to spread out the pixel intensity values. This allows the image's areas with lower contrast to gain a higher contrast. See example below.

Next, we define a function called **preprocess() that combines the grayscale() and equalize() methods, as well as normalise the pixel intensity value to a value between 0 and 1**. This way, the numbers will be small and the computation becomes easier and faster.

```
#distributes the grayscale across each image uniformly
def equalize(img):
    img = cv2.equalizeHist(img)
    return img

img = equalize(img)
plt.imshow(img, cmap='gray_r')
plt.axis("off")
print(img.shape)
```

```
(32, 32)
```



```
[13] #create function to apply image preprocessing to all images
def preprocess(img):
    img = grayscale(img)
    img = equalize(img)
    img = img/255
    return img

X_train = np.array(list(map(preprocess, X_train)))
X_val = np.array(list(map(preprocess, X_val)))
X_test = np.array(list(map(preprocess, X_test)))
```

We then **apply the preprocess method on our feature datasets**.

In order to input our feature data into the CNN model, we need to first **reshape our data** (which is current 32 pixel by 32 pixel) into this specific format (**32 pixel by 32 pixel by 1 channel**).

We also have to use the **to_categorical() method to one-hot encode** our label datasets. One-hot encoding converts categorical information into a format that may be fed into machine learning algorithms to improve prediction accuracy.

```
print(X_train.shape)
print(X_test.shape)
print(X_val.shape)
```

```
(34799, 32, 32)
(12630, 32, 32)
(4410, 32, 32)
```

```
[16] #to input the data into CNN, need the depth dimension
     X_train = X_train.reshape(34799, 32, 32, 1)
     X_test = X_test.reshape(12630, 32, 32, 1)
     X_val = X_val.reshape(4410, 32, 32, 1)
```

```
[17] print(X_train.shape)
```

```
(34799, 32, 32, 1)
```

```
[18] #one-hot encoding so that it can be fed into algorithm (increase prediction accuracy)
     y_train = to_categorical(y_train, 43)
     y_test = to_categorical(y_test, 43)
     y_val = to_categorical(y_val, 43)
```

Previously, the first image in our dataset has the label of 41 (referencing to End of No Passing road sign).

```
print(y_train[0])
```

```
41
```

After one-hot encoding, the label becomes a series of 43 binary values. Since the first image's label is essentially 41, notice that the 41^{st} value is 1 while the rest are 0.

```
print(y_train[0])
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
```

A common trick in improving image recognition models is to **generate artificial images** to train the model. To do that, we can use the **Keras ImageDataGenerator** to translate the image vertically and horizontally, zoom into or out of the image, shear the image, and rotate the image.

We can set the batch size as 15 such that every time the generator is called, it will generate 15 new images.

```
[19] #generate additional artificial data of different variations (e.g. rotation, zoomed in/out, offset) to improve model performance
     from keras.preprocessing.image import ImageDataGenerator

     datagen = ImageDataGenerator(width_shift_range=0.1,            #max 10% shift
                                  height_shift_range=0.1,           #max 10% shift
                                  zoom_range=0.2,                   #max 20% zoom in/out
                                  shear_range=0.1,                  #max 10% shear
                                  rotation_range=10)                #max 10degrees rotation

     datagen.fit(X_train)
```

```
#to get datagen to start generating new images
#batches is an iterator object, can be called by next() method (will generate 15 images everytime method is called)
batches = datagen.flow(X_train, y_train, batch_size = 15)
X_batch, y_batch = next(batches)

fig, axs = plt.subplots(1, 15, figsize=(20, 5))
fig.tight_layout()

for i in range(15):
    axs[i].imshow(X_batch[i].reshape(32, 32))
    axs[i].axis("off")

print(X_batch.shape)
```

```
(15, 32, 32, 1)
```

Next, let's create the **CNN model**. There is no specific formula for creating a model and is usually done through a series of trial and error. In Convolutional Neural Networks, **filters** (e.g. 60, 30 below) detect spatial patterns such as edges in an image by detecting the changes in intensity values of the image.

Kernel size (e.g. (5, 5) or (3, 3) below) is the size of the filter.

```
#create model
def modified_model():
  model = Sequential()
  model.add(Conv2D(60, (5, 5), input_shape=(32, 32, 1), activation='relu'))
  model.add(Conv2D(60, (5, 5), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))

  model.add(Conv2D(30, (3, 3), activation='relu'))
  model.add(Conv2D(30, (3, 3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))

  model.add(Flatten())
  model.add(Dense(500, activation='relu'))
  model.add(Dropout(0.5))
  model.add(Dense(43, activation='softmax'))

  model.compile(Adam(lr = 0.001), loss='categorical_crossentropy', metrics=['accuracy'])
  return model
model = modified_model()
print(model.summary())
```

For a multi-classification problem, we can set our last **activation function as 'softmax'**. The softmax activation function transforms the raw outputs of the neural network into a vector of probabilities, allowing us to return the classification with the highest probability.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 60)        1560

 conv2d_1 (Conv2D)           (None, 24, 24, 60)        90060

 max_pooling2d (MaxPooling2D  (None, 12, 12, 60)       0
 )

 conv2d_2 (Conv2D)           (None, 10, 10, 30)        16230

 conv2d_3 (Conv2D)           (None, 8, 8, 30)          8130

 max_pooling2d_1 (MaxPooling  (None, 4, 4, 30)         0
 2D)

 flatten (Flatten)           (None, 480)               0

 dense (Dense)               (None, 500)               240500

 dropout (Dropout)           (None, 500)               0

 dense_1 (Dense)             (None, 43)                21543

=================================================================
Total params: 378,023
Trainable params: 378,023
Non-trainable params: 0
```

Now that we have set up our model, let's **fit the model** with our training images. Since we have an image generator, we can use the **model.fit_generator()** method to create new images and to fit the model.

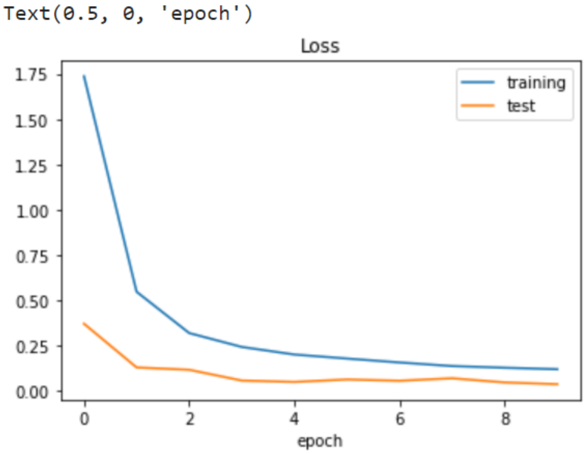
Batch size refers to the number of new images we generate each time we call the generator and **steps_per_epoch** refers to the number of times we call the generator in 1 epoch. An **epoch** just basically means when all training datasets have been passed through the model once.

```
history = model.fit_generator(datagen.flow(X_train, y_train, batch_size=50),
                              steps_per_epoch=X_train.shape[0]/50,
                              epochs=10,
                              validation_data=(X_val, y_val), shuffle = 1)
```

```
Epoch 1/10
<ipython-input-22-7dbfefff6b2a>:1: UserWarning: `Model.fit_generator` is deprecated and will be removed in a future version. Please u
  history = model.fit_generator(datagen.flow(X_train, y_train, batch_size=50),
695/695 [==============================] - 21s 22ms/step - loss: 1.7361 - accuracy: 0.5076 - val_loss: 0.3688 - val_accuracy: 0.8866
Epoch 2/10
695/695 [==============================] - 15s 21ms/step - loss: 0.5454 - accuracy: 0.8339 - val_loss: 0.1271 - val_accuracy: 0.9605
Epoch 3/10
695/695 [==============================] - 15s 21ms/step - loss: 0.3171 - accuracy: 0.9015 - val_loss: 0.1141 - val_accuracy: 0.9658
Epoch 4/10
695/695 [==============================] - 15s 22ms/step - loss: 0.2404 - accuracy: 0.9248 - val_loss: 0.0547 - val_accuracy: 0.9830
Epoch 5/10
695/695 [==============================] - 15s 22ms/step - loss: 0.1985 - accuracy: 0.9369 - val_loss: 0.0469 - val_accuracy: 0.9859
Epoch 6/10
695/695 [==============================] - 16s 24ms/step - loss: 0.1762 - accuracy: 0.9466 - val_loss: 0.0606 - val_accuracy: 0.9796
Epoch 7/10
695/695 [==============================] - 15s 22ms/step - loss: 0.1546 - accuracy: 0.9523 - val_loss: 0.0536 - val_accuracy: 0.9837
Epoch 8/10
695/695 [==============================] - 17s 24ms/step - loss: 0.1348 - accuracy: 0.9586 - val_loss: 0.0671 - val_accuracy: 0.9766
Epoch 9/10
695/695 [==============================] - 15s 22ms/step - loss: 0.1255 - accuracy: 0.9616 - val_loss: 0.0443 - val_accuracy: 0.9846
Epoch 10/10
695/695 [==============================] - 15s 22ms/step - loss: 0.1172 - accuracy: 0.9638 - val_loss: 0.0346 - val_accuracy: 0.9893
```

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['training','test'])
plt.title('Loss')
plt.xlabel('epoch')
```

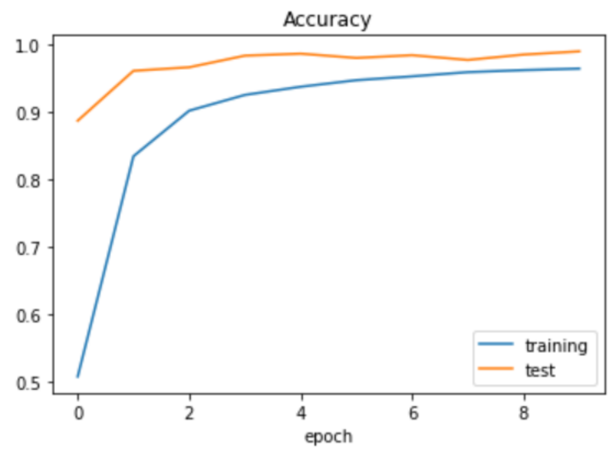
```
Text(0.5, 0, 'epoch')
```



Notice that the validation accuracy is higher than the training accuracy. Technically, we can continue to run the model for a few more epochs to see if they converge.

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['training','test'])
plt.title('Accuracy')
plt.xlabel('epoch')
```

```
Text(0.5, 0, 'epoch')
```



Next, we measure how our model work on the test data (this is the 3rd set of data that the model had not seen before). Great, **96.7% accuracy**!

```
[25] #Evaluate model on test data
     score = model.evaluate(X_test, y_test, verbose=0)
     print('Test score:', score[0])
     print('Test accuracy:', score[1])
```

```
Test score: 0.1292283833026886
Test accuracy: 0.9671417474746704
```

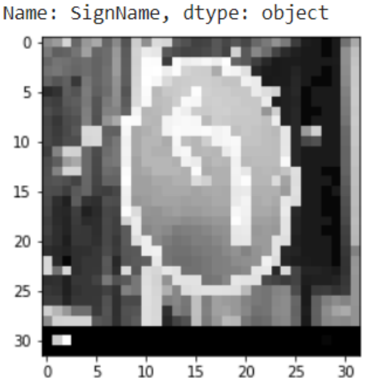
Now that we have a trained model, let's **test it out** on random images culled from the internet. Yay, the model works well!

```
#predict never seen before images on internet
import requests
from PIL import Image
url = 'https://c8.alamy.com/comp/A0RX23/cars-a
r = requests.get(url, stream=True)
img = Image.open(r.raw)
plt.imshow(img, cmap=plt.get_cmap('gray'))
```

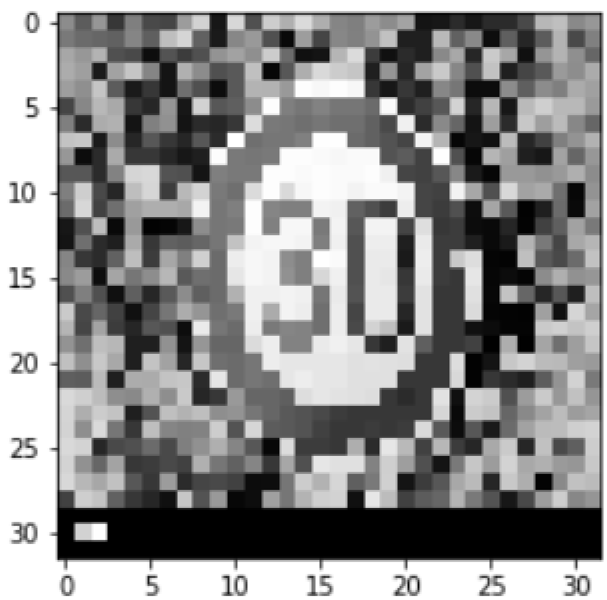
```
<matplotlib.image.AxesImage at 0x7f8613b4b130>
```

```
img = np.asarray(img)
img = cv2.resize(img, (32, 32))
img = preprocess(img)
plt.imshow(img, cmap = plt.get_cmap('gray'))
print(img.shape)
img = img.reshape(1, 32, 32, 1)

print("predicted sign: "+ str(np.argmax(model.predict(img), axis = 1)))
print("predicted sign: " + data['SignName'].iloc[np.argmax(model.predict(img), axis = 1 )])
```

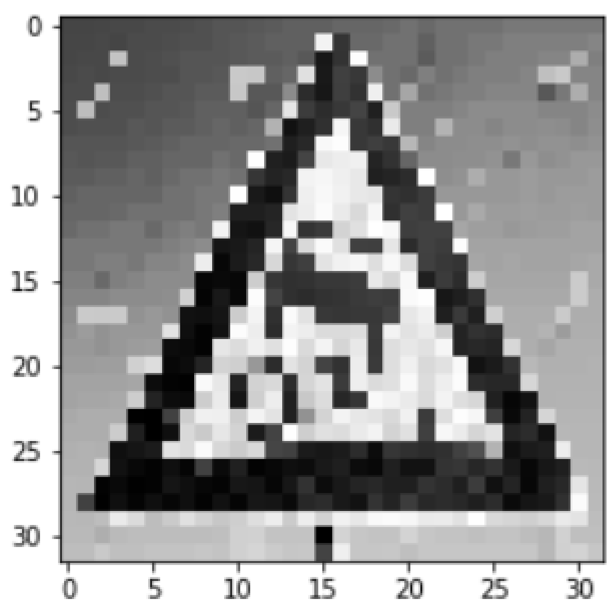
```
(32, 32)
1/1 [==============================] - 0s 150ms/step
predicted sign: [34]
1/1 [==============================] - 0s 18ms/step
34    predicted sign: Turn left ahead
Name: SignName, dtype: object
```

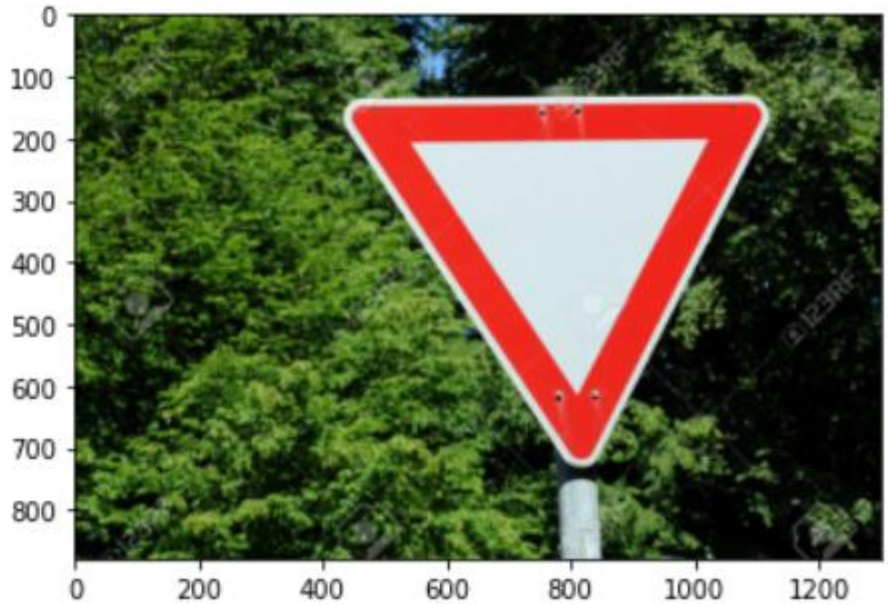



```
1    predicted sign: Speed limit (30km/h)
Name: SignName, dtype: object
```

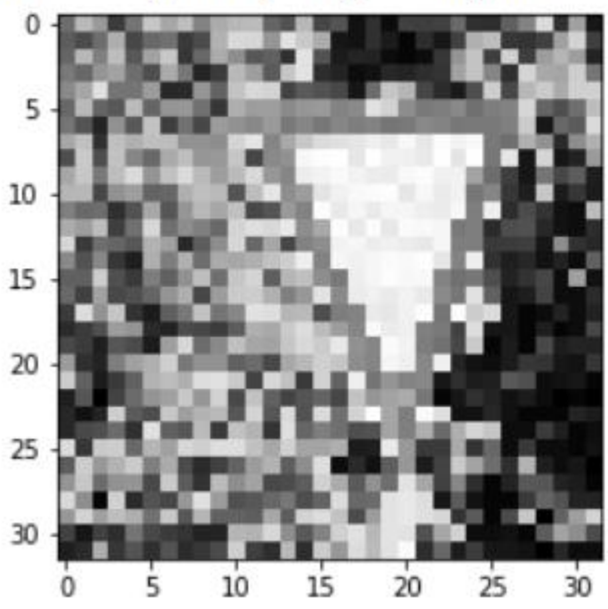



```
23    predicted sign: Slippery road
Name: SignName, dtype: object
```

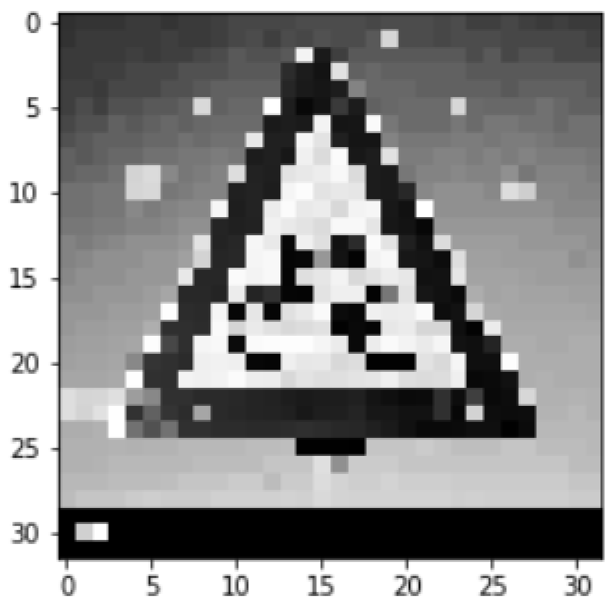




```
13    predicted sign: Yield
Name: SignName, dtype: object
```




```
29    predicted sign: Bicycles crossing
Name: SignName, dtype: object
```




```
14    predicted sign: Stop
Name: SignName, dtype: object
```