Traffic Signs Classification

I. Data Set Summary & Exploration

I used the numpy library to calculate summary statistics of the traffic signs data set:

1. Size of training set is: 34799 samples
2. Size of validation set is: 4410 samples
3. Size of test set is: 12630 samples
4. Shape of traffic sign image is: 32x32x3
5. Number of unique classes: 43

Here is an exploratory visualization of the data set. Following are the bar charts for each of the training, validation and test sets, showing how the distribution of samples across 43 different classes look like:

X-axis represents the class index, and y-axis represents the number of samples in each class.

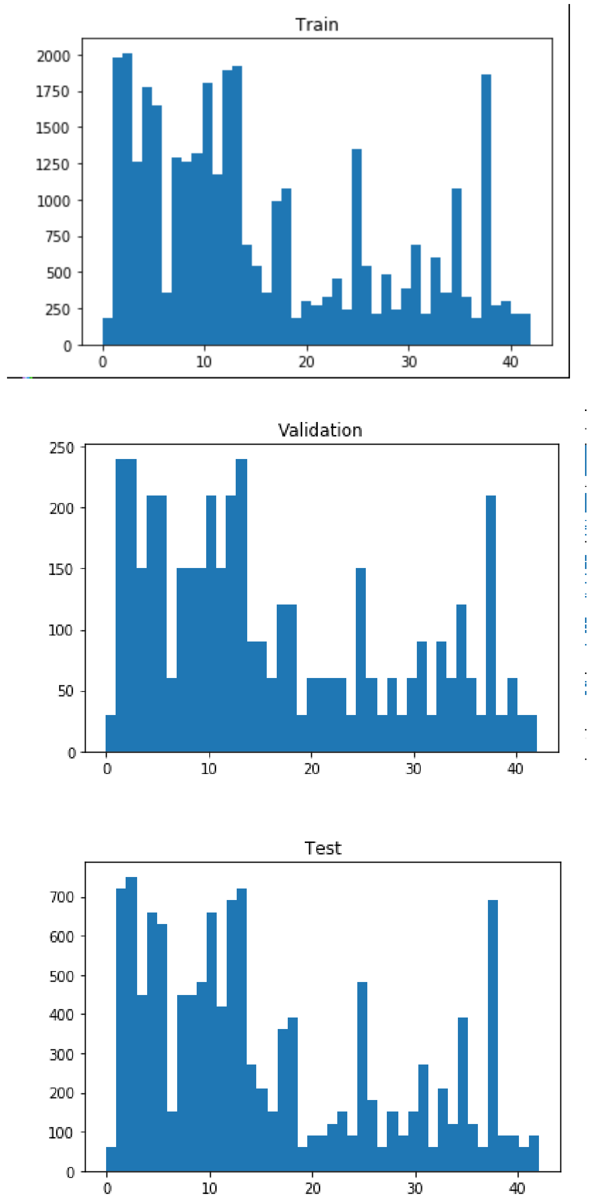


Figure 1: Distribution of training, validation and test data sets

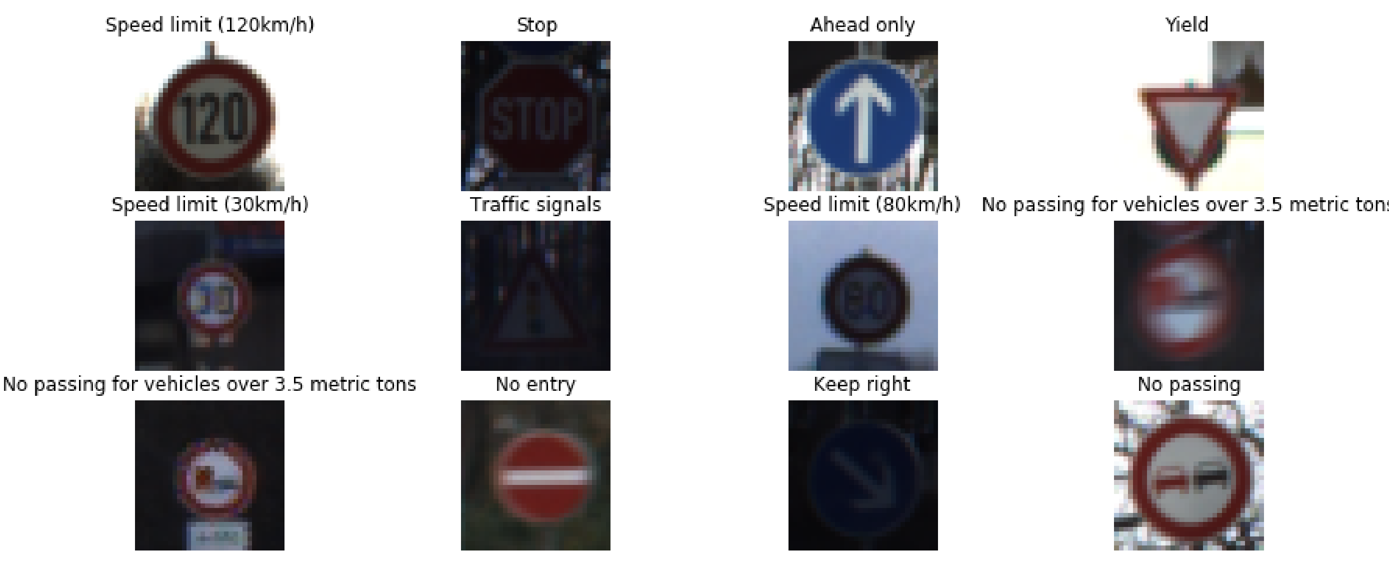


Figure 2: A few examples of the data set and their true labels

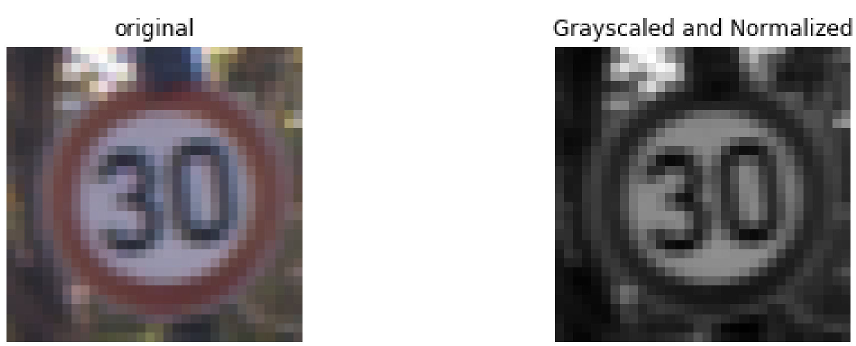
II. Design and model architecture

1. Data pre-processing

a. <u>Grayscaling</u>: As a first step, the data sets are converted to greyscale, as suggested in Semanent and LeCun paper (http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf). This works well under color indifference conditions. This also reduces number of channels from 3 to 1 and helps in reducing training time.

b. <u>Normalizing</u>: To have equal importance to all features in an image during the 'learning' process, all image samples are made to have zero mean and equal standard deviations. This is performed by simply, x_new = (x_data – 128)/128.

Here is an example of grayscaled and normalized image.



c. <u>Data augmentation</u>:
Reason to append more data to the existing training set: As you can observe in Figure 1, distribution of training data is not uniform. Some of the classes occur more frequently that the rest (example, class 2), whereas, a few other classes are represented far less (example, class 0). If we generate additional examples for minority classes in training set, then homogeneity of

samples and observability for the network will be enhanced, therefore the accuracy will be improved.

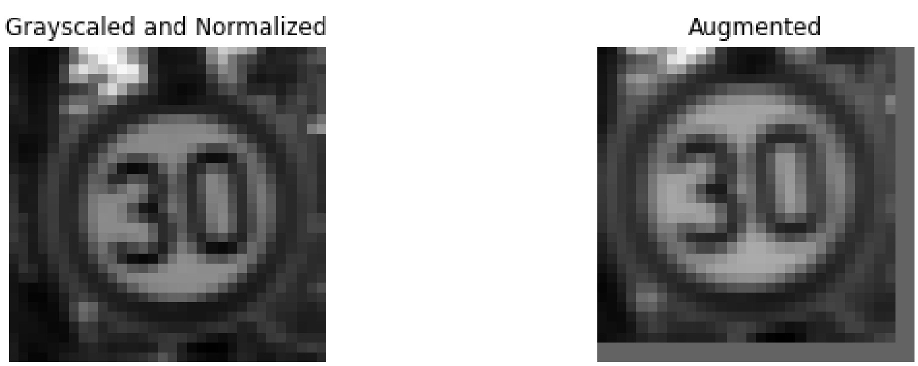
Steps involved in data augmentation: This was the most time-consuming task of this project, both in terms of tuning augmentation parameters and generating augmented images.

i. First, ‘minority’ classes were identified by counting the number of samples for each class that falls below a threshold. Threshold here is the mean of samples in each class, which is 809.
ii. Next, for every ‘minority’ class, images were chosen one at a time in circular sequence, and were subjected to the following sequence of transformation:

```
translation(scaling(AffineTransform(brightness(picture_to_augment))))
```

where ‘brightness’ method applies random variation in image brightness, ‘AffineTransform’ performs random 2D geometric transformation of the input image, ‘scaling’ performs random zoon-in and zoom-out and finally ‘translation’ performs random shifting of image in x and y directions. These sequences of operations change the original image into a different image as though the ‘traffic sign’ was captured at a different spatio-temporal instance, therefore not causing any redundancy in ‘augmented’ dataset.
Tuning the parameters within each method that performs ‘random’ transformation was a difficult task, and was conducted iteratively in a way so as to get the best result possible. The python script submitted for the project has another method (originally written as a part of augmentation process), called ‘rotation’ which tilts image in random acute angles. However, it was found to be ineffective in enhancing classification accuracy, or rather not of significant importance given the time consumption for data augmentation, and therefore is not used.

Here is an example of augmented image.



Training set size of after data augmentation:

```
X, y shapes: (46714, 32, 32, 1) (46714,1)
```

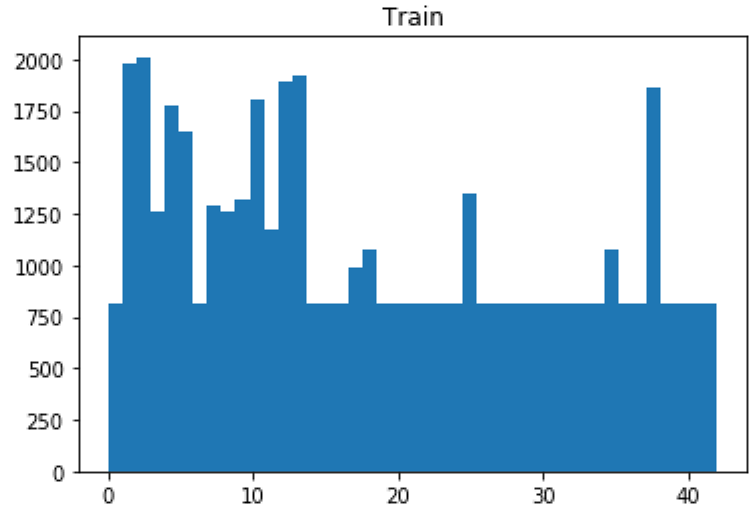


Figure 3: Distribution of training set after data augmentation, which looks more uniform, mean number of samples is 809

2. Model architecture

My final model consisted of the following layers: This architecture is similar to the one described in Sermanet and LeCun paper (http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf). It is a 3 stage Conv-Net with 3 fully connected layers (total 6 layers).

Following table describes sequential architecture.

Table 1 Final Model

Layer	Function	Description
	Input	32x32x1 Normalized grayscale image
Layer 1	Convolution 5x5	1x1 stride, valid padding, outputs 28x28x6
	RELU	Rectified linear unit, outputs 28x28x6
	Max pooling	**No** max-pooling in Layer 1 (to reserve feature map information), hence output is still 28x28x6
Layer 2	Convolution 5x5	1x1 stride, valid padding, outputs 24x24x16
	RELU	Rectified linear unit, outputs 24x24x16
	Max pooling	**No** max-pooling in Layer 2 (to reserve feature map information), hence output is still 24x24x16
Layer 3	Convolution 5x5	1x1 stride, valid padding, outputs 20x20x32
	RELU	Rectified linear unit, outputs 20x20x32
	Max pooling	2x2 stride, outputs 10x10x32
	Flattening	Layer 3 output is flattened to yield 10x10x32 = 3200 linear points. This will go to classifier. The output of Layer 2 (24x24x16 = 9216) is also fed directly to classifier as higher-resolution features (Sermanent and LeCun paper). Therefore, total number of points after flattening, 3200 + 9216 = 12416
Layer 4	Full connection	Input 12416, output 800
	RELU	Rectified linear unit, output 800
Layer 5	Full connection	Input 800, output 400
	RELU	Rectified linear unit, output 400
Layer 6	Full connection	Input 400, output 43
	SoftMax	Final output, 43 class probabilities

III. Training

To train the model, I used Adam optimizer to minimize cross-entropy loss, with following settings:

Learning rate = 0.001
Epochs = 15
Batch size = 128
Random weight initializer mu = 0.0
Random weight initializer sigma = 0.1

I have also used dropout feature for full connection layer 4 and layer 5, with keep probability 0.6.
In addition, I have used L2 regularization to reduce over fitting. The beta parameter for L2 regularization is 0.0001

IV. Approach for validation accuracy better than 0.93

The approach followed was iterative, wherein first the basic LeNet-5 architecture was implemented, followed by tuning of hyperparameters in LeNet-5 architecture, such as epoch, batch size etc. Initial test and validation accuracies were around 0.90.
Next, as described in Sermanet and LeCun paper, grayscaling and data normalization was introduced, which increased validation accuracy to around 0.92.
Later, different dimensions of LeNet layers were tested, such as adding additional layers, changing strides, removing max-pool layers (to retain high resolution features), etc. This made the test accuracy to be around 0.97, however validation accuracy was still around 0.93. The difference in validation and test accuracy meant that the model must be overfitting.
Therefore, to reduce overfitting, dropout regularization and L2 regularization were introduced. This made the validation accuracy to raise up to 0.96.
Next, the data augmentation was introduced to improve diversity of test samples. (There was a lot of tuning and trial-and-error involved in this step). Data augmentation further improved validation accuracy.

The final model's (Table 1) epoch-by-epoch test and validation accuracy are as follows:

```
EPOCH 1 ...
   Training Accuracy = 0.952
  Validation Accuracy = 0.919
EPOCH 2 ...
   Training Accuracy = 0.987
  Validation Accuracy = 0.964
EPOCH 3 ...
   Training Accuracy = 0.994
  Validation Accuracy = 0.963
EPOCH 4 ...
   Training Accuracy = 0.997
  Validation Accuracy = 0.974
EPOCH 5 ...
   Training Accuracy = 0.998
  Validation Accuracy = 0.977
EPOCH 6 ...
   Training Accuracy = 0.998
  Validation Accuracy = 0.975
EPOCH 7 ...
   Training Accuracy = 0.998
  Validation Accuracy = 0.979
EPOCH 8 ...
   Training Accuracy = 0.998
  Validation Accuracy = 0.977
EPOCH 9 ...
   Training Accuracy = 0.999
  Validation Accuracy = 0.974
EPOCH 10 ...
   Training Accuracy = 0.999
  Validation Accuracy = 0.975
EPOCH 11 ...
   Training Accuracy = 0.997
```

```
  Validation Accuracy = 0.971
EPOCH 12 ...
   Training Accuracy = 0.999
  Validation Accuracy = 0.982
EPOCH 13 ...
   Training Accuracy = 0.999
  Validation Accuracy = 0.976
EPOCH 14 ...
   Training Accuracy = 0.999
  Validation Accuracy = 0.977
EPOCH 15 ...
   Training Accuracy = 0.999
  Validation Accuracy = 0.986
```

That is **98.6%** validation accuracy!
And test accuracy is found to be **96.1%** (cell 26 of Ipython script)

V. Test on new images

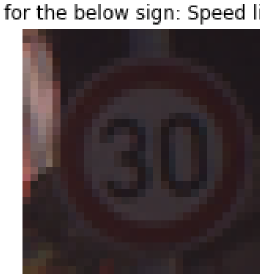
I chose 12 images of different size and qualities from German traffic signs.

Image size vary up to 250x250 pixels, all reshaped to 32x32 within the code to be suitable for training.

Following are those 12 images:

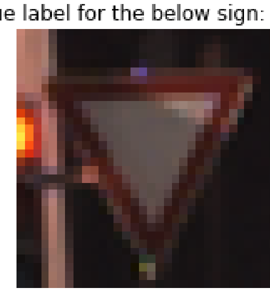
True label for the below sign: Speed limit (30km/h)

True label for the below sign: Speed limit (30km/h)



True label for the below sign: Yield

True label for the below sign: Yield



Above two image-pairs represent the same sign, but with two different image intensities. First pair for ‘Speed limit 30 km/h’ and second pair for ‘Yield’. It would be interesting to see how the network performs.

Images in last row, ‘Keep left’ and ‘Roundabout mandatory’ seem pose difficulty for the network to classify. They are recognizable seen even for human eyes. They seem to be shot in dark.

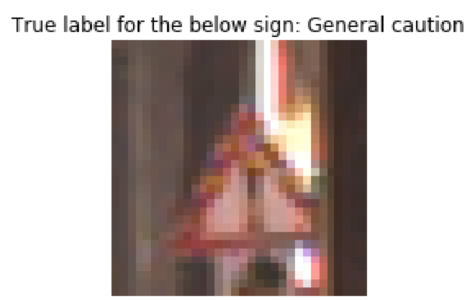


Image above in the right-hand side, ‘General Caution,’ is yet another problematic picture, barely visible to human eye.

VI. Performance on new images

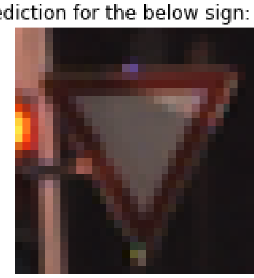
The model could predict all images correctly, leading to accuracy of 100%. However, prediction probabilities vary (discussed later). Following are the prediction for each image:

Prediction for the below sign: Speed limit (30km/h)

Prediction for the below sign: Speed limit (30km/h)

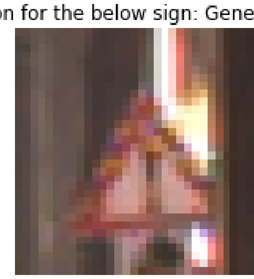
Prediction for the below sign: No passing

Prediction for the below sign: Yield



Prediction for the below sign: Yield

Prediction for the below sign: General caution



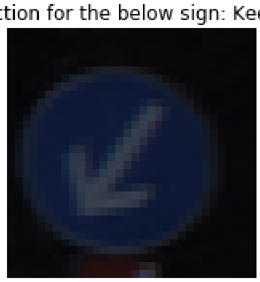
Prediction for the below sign: Road work

Prediction for the below sign: Pedestrians

Prediction for the below sign: Children crossing

Prediction for the below sign: Wild animals crossing

Prediction for the below sign: Keep left

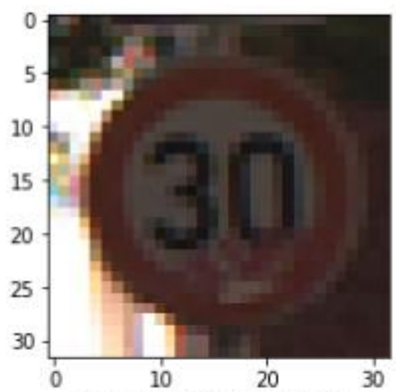


Prediction for the below sign: Roundabout mandatory

VII. SoftMax probabilities/ Top-5 predictions

The code for making predictions on my final model is in the 30th cell of the Ipython notebook. For 10 out of 12 images, the network clearly identifies true label.

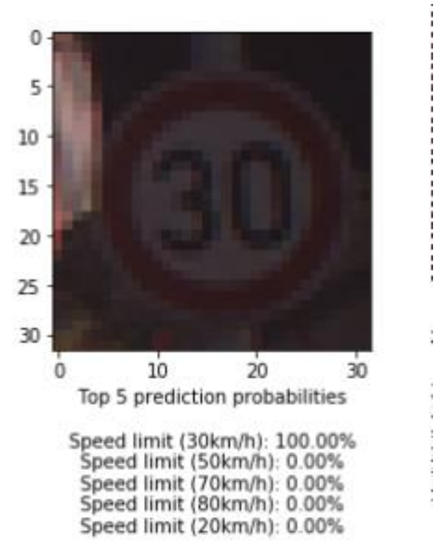
1. First image: True label – Speed limit 30 km/hr.



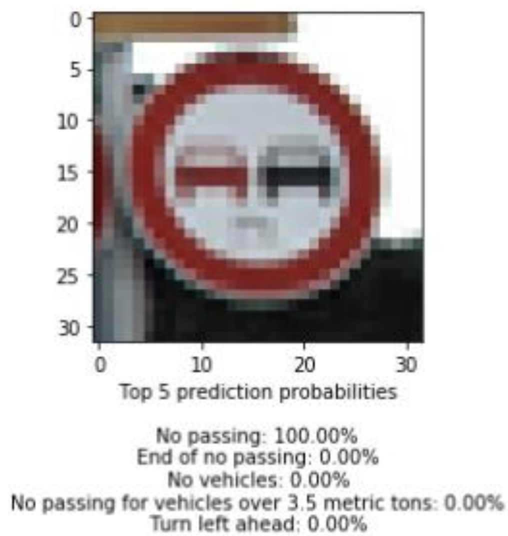
Top 5 prediction probabilities

Speed limit (30km/h): 100.00%
Speed limit (50km/h): 0.00%
Speed limit (70km/h): 0.00%
Speed limit (80km/h): 0.00%
Speed limit (20km/h): 0.00%

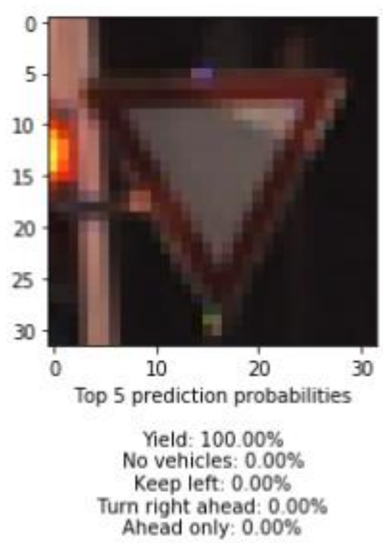
2. Second image: True label – Speed limit 30 km/hr.



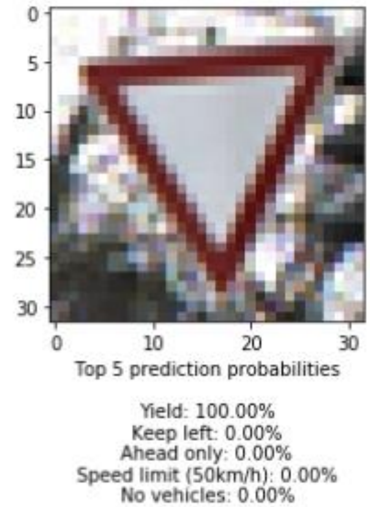
3. Third image: True label – No passing



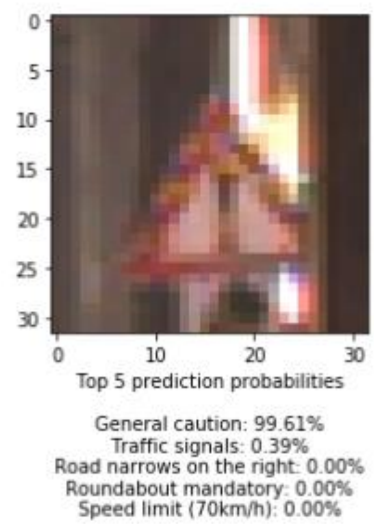
4. Fourth image: True label – Yield



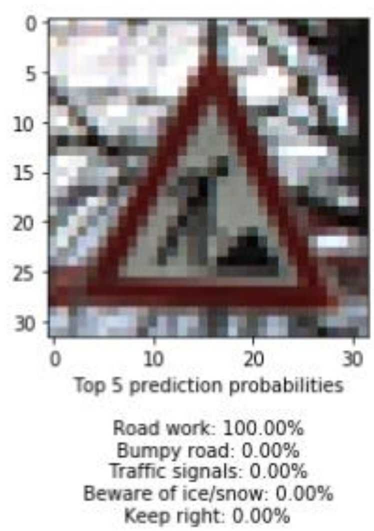
5. Fifth image: True label – Yield



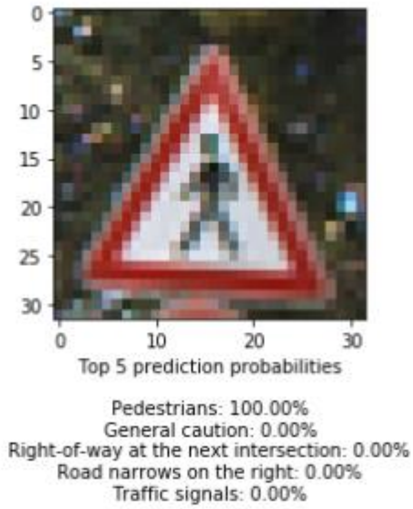
6. Sixth image: True label – General caution
   Here, the prediction wasn't as good the rest, given the quality of picture



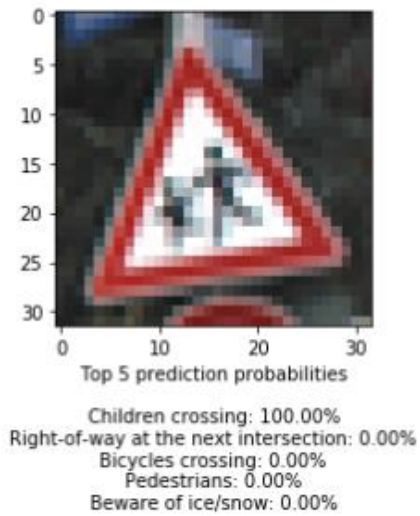
7. Seventh image: True label – Road work



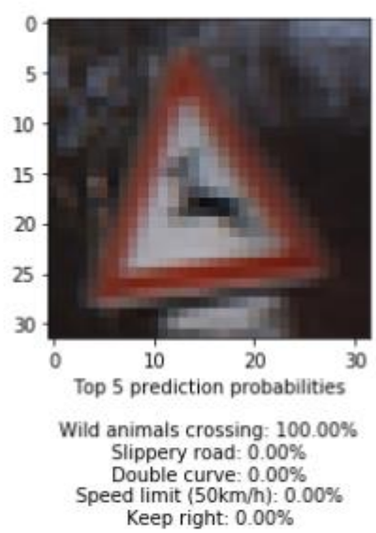
8. Eighth image: True label – Pedestrians



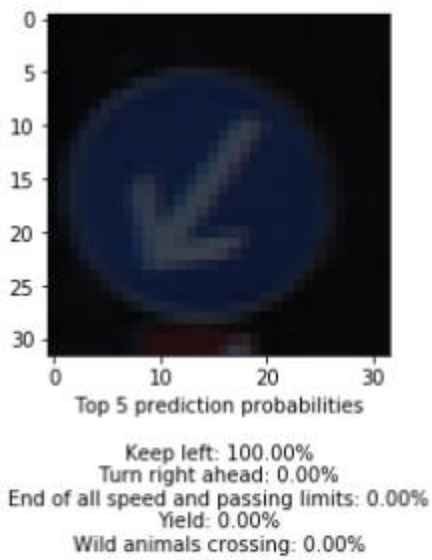
9. Ninth image: True label – Children crossing



10. Tenth image: True label - Wild animals crossing



11. Eleventh image: True label – Keep left



12. Twelfth image: True label – Round about mandatory
    Here as well, the guess isn't strong given the poor quality of image