ECE – 510

PORTLAND STATE UNIVERSITY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Intelligent Robotics III

Project: Traffic Signal Detection for Autonomous Vehicles

Guidance: Dr. Marek Perkowski

Aniruddha Shahapurkar (anirud@pdx.edu)

Date: 5/25/2020

INDEX

❖ INTRODUCTION:

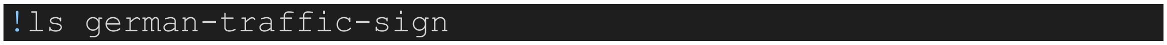
- Traffic sign detection and recognition plays an important role in expert systems, such as traffic assistance driving systems and automatic driving systems. It instantly assists drivers or automatic driving systems in detecting and recognizing traffic signs effectively.
- In our data set, there are 43 different classes for Traffic signals.
- Our dataset is smaller but complex.
- Therefore, preprocessing of the images is crucial for us to succeed in this task.
- We will import data from an external source, and then we will process the data to train a **neural network**.
- Preprocessing and preparing dataset are extremely important in the field of Deep Learning.
- **The dataset is smaller containing fewer images in the training set and validation set and the test set. Also, the dataset contains many classes (total 43), in where the traffic signs can be classified. This means not only our dataset is smaller, but also, we have fewer images belonging to each class since the dataset is being classified into many classes and the dataset is complex. This makes it difficult for our network to learn properly for the high accuracy rates.**

❖ IMPLEMENTATION:

Preprocessing Images:

1. Collecting Data:

- We get the dataset bitbucket library by git-cloning it.
- After cloning it, we have directory called German traffic signs.
- Then we use following command:



```
!ls german-traffic-sign
```

- This command lists the files contained by german-traffic-sign repository. There are four files:
  signname.csv   test.p   train.p   valid.p
  Last three are pickle files.
  The pickle module implements binary protocols for serializing and de-serializing a Python object structure. *"Pickling"* is the process whereby a Python object hierarchy is converted into a byte stream, and *"unpickling"* is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as "serialization", "marshalling," or "flattening"; however, to avoid confusion, the terms used here are "pickling" and "unpickling".

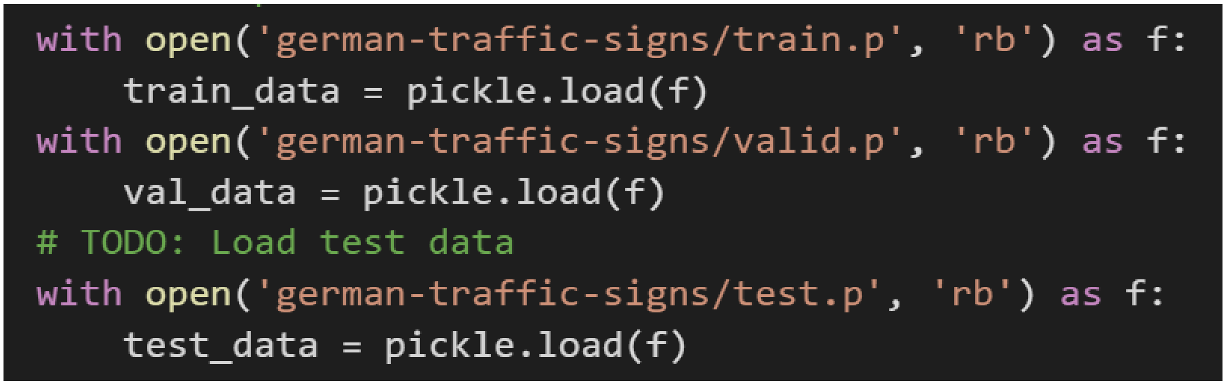
(i) **signname.csv**:
This is a spreadsheet which contains all of our traffic signs along with their labels.

(ii) **test.p**:
We will unpickle this to obtain test data.

(iii) **train.p**:
We will unpickle this to obtain training data.

(iv) **valid.p**:
We will unpickle this to obtain validation data.

- The unpickling is done with the help of following function:

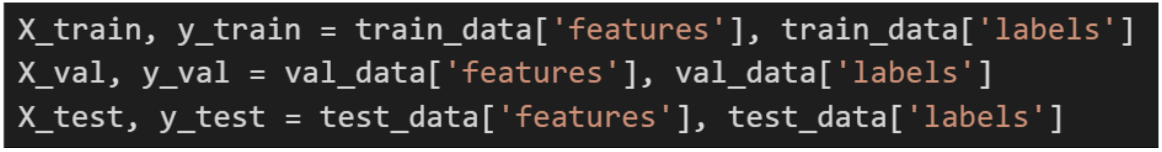


```
with open('german-traffic-signs/train.p', 'rb') as f:
    train_data = pickle.load(f)
with open('german-traffic-signs/valid.p', 'rb') as f:
    val_data = pickle.load(f)
# TODO: Load test data
with open('german-traffic-signs/test.p', 'rb') as f:
    test_data = pickle.load(f)
```

The function performs following steps:

- Open training data, then 'rb' or
  read in binary format, then store it in the variable f
- Unpickle the data stored in f and store it respectively in the train_data, val_data and test data so we can use it.

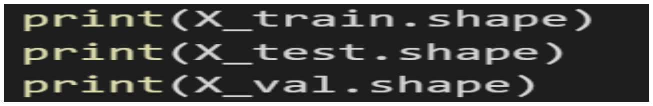
- These unpickeled datasets are dictionaries with key value pairs. We are interested in two values, viz., features and labels.
- Then we split out the features and labels using following code:



```
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']
```

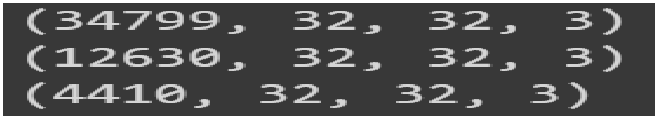
  - features: Values of training images in pixel representation,
  - labels: labels each training image to its belonging class

- Then, we print the shape of our training, validation and test dataset using following code:



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

From this, we get following results:

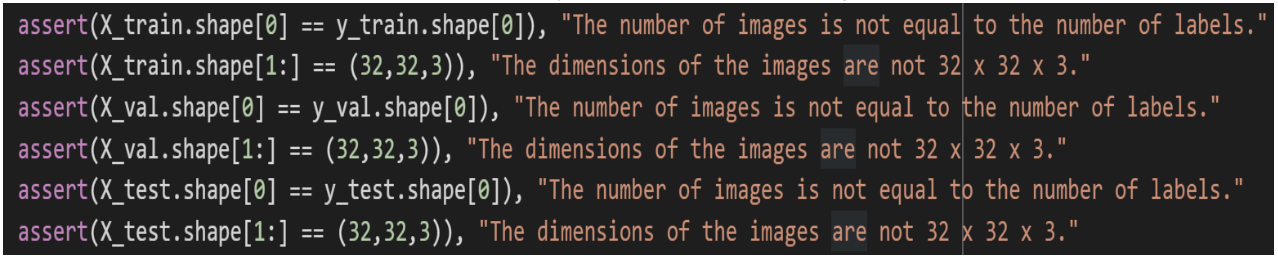


```
(34799, 32, 32, 3)
(12630, 32, 32, 3)
(4410, 32, 32, 3)
```

The print format is as follows:

- Number of images, Dimension X, Dimension Y, Depth.

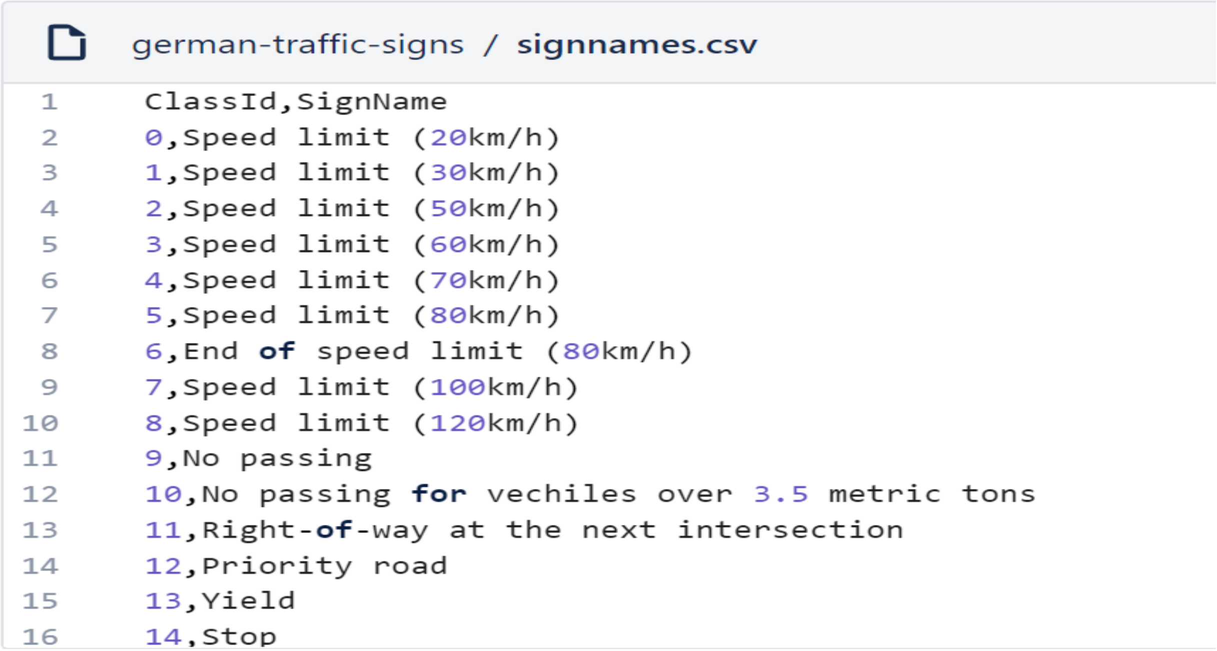
- Then, in order to maintain data consistency, we use the following code:



```
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."
```

  - Our implementation should always pass these conditions.
  - We are basically checking two conditions:
    1. No. of images are equal to no. of labels or not
    2. Dimensions match 32 x 32 x 3 or not

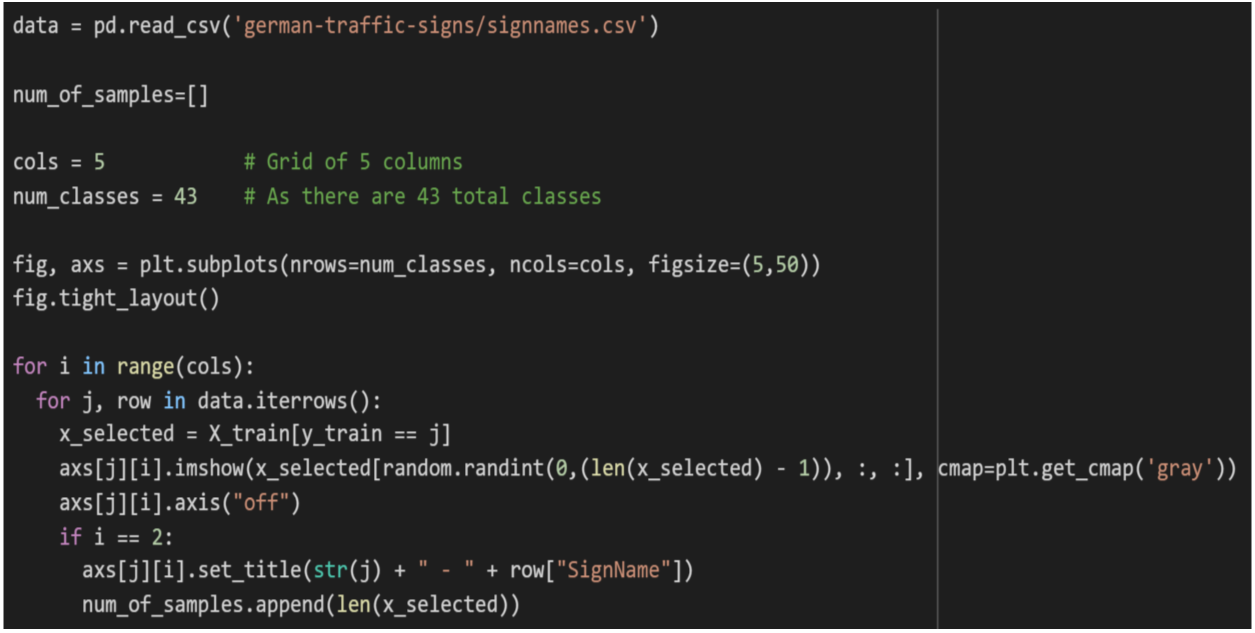
- The signnames.csv file looks as follows:



german-traffic-signs / signnames.csv

```
ClassId,SignName
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
10,No passing for vechiles over 3.5 metric tons
11,Right-of-way at the next intersection
12,Priority road
13,Yield
14,Stop
```

  - It contains all the classes in the traffic sign classes (43 in total) in a CSV format.
  - To manipulate and analyze data inside of a CSV file, we will make use of a python data analysis library known as ***"Pandas"***.
- Let's plot our training images into a grid of appropriate rows and columns using following code:

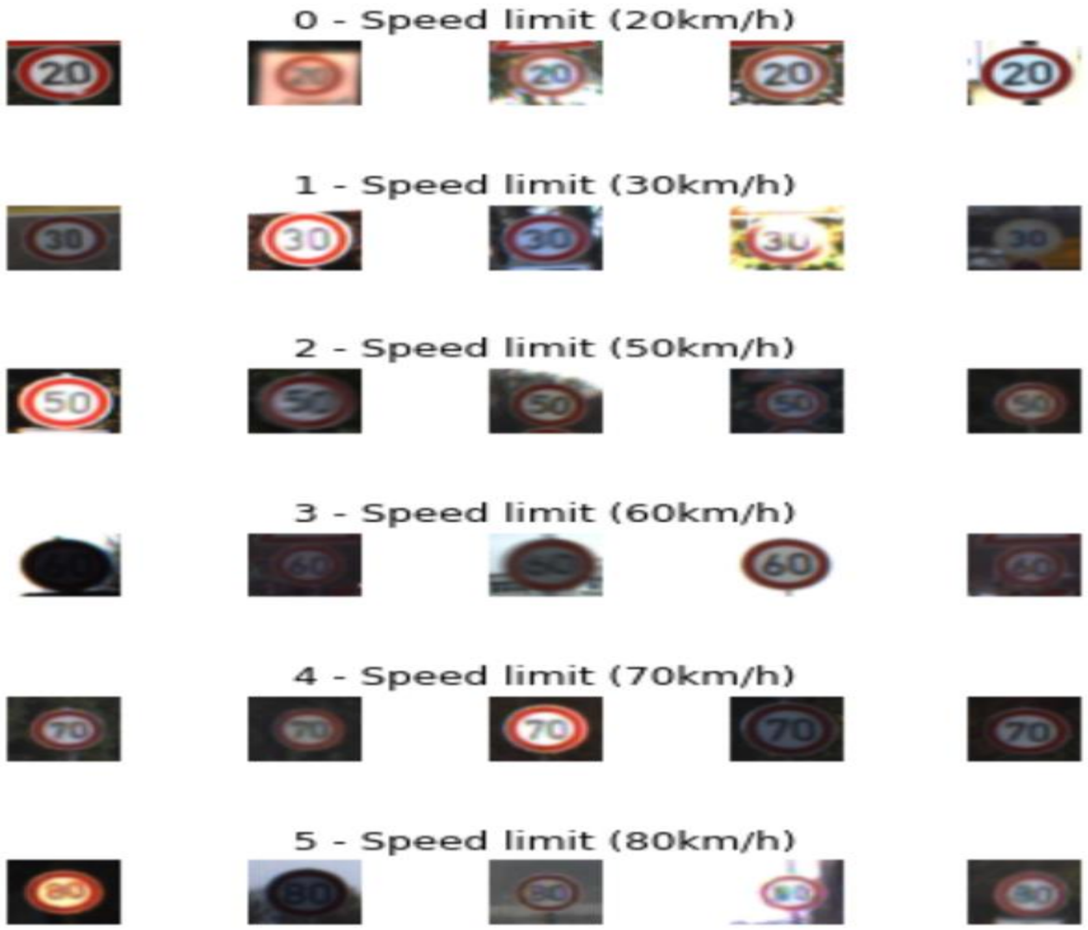


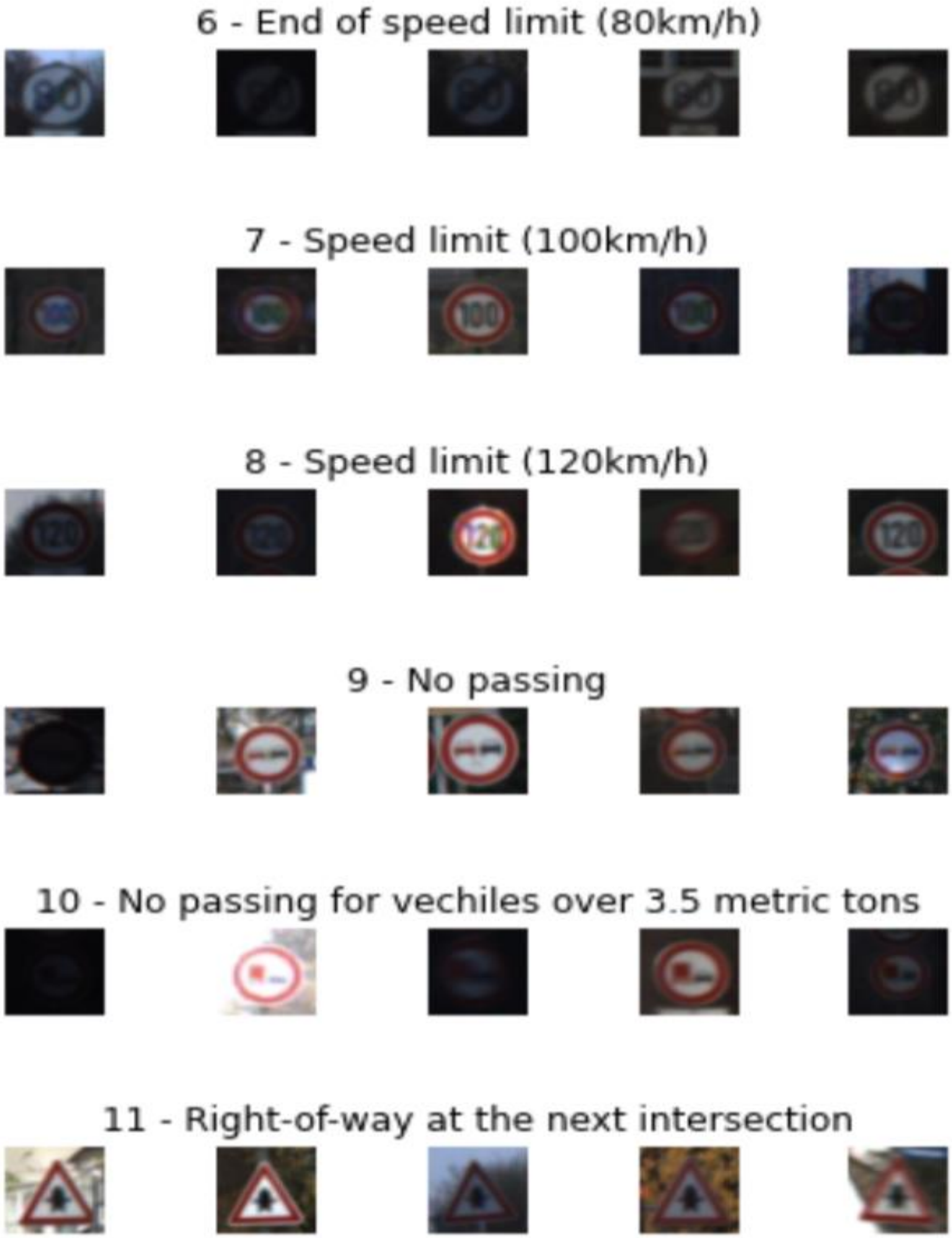
```
data = pd.read_csv('german-traffic-signs/signnames.csv')

num_of_samples=[]

cols = 5            # Grid of 5 columns
num_classes = 43    # As there are 43 total classes

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))
```

- In this code we perform the following operations:
  - We select number of columns equal to 5
  - Set number of rows = Number of classes = 43

  - Set the figure size of the plot to 5 x 50
  - Our outer loop is for columns.
  - For every column, we are iterating through each of its 43 rows.
  - Out of all the training data present in X_train, select the data of the current row.
  - In other words, as we know there are 34799 training images. Out of those, we need to find the images which belong to the class which is currently under iteration.
  - Out of all x_selected images on previos step, we select one image randomly and place it in the current grid, also changing the color map to gray scale for better representation of the selected image.
  - Next step is adding title to each of the plotted class.
  - If we reach column 2 (the middle of the available columns), we set the title to j (1,2,3,4,......43) + Name of the traffic sign class.
  - Then finally we use following line of code: **num_of_samples.append(len(x_selected))** for finding out total number of images belonging to each of the 43 classes.

- After running this code snippet, we get the following results:





6 - End of speed limit (80km/h)

7 - Speed limit (100km/h)

8 - Speed limit (120km/h)

9 - No passing

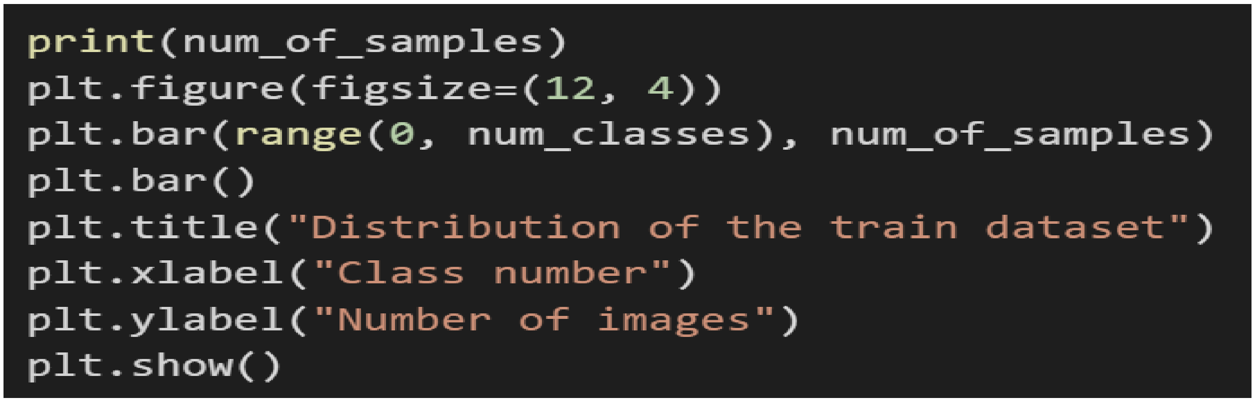
10 - No passing for vechiles over 3.5 metric tons

11 - Right-of-way at the next intersection

- We have only shown first 11 out of 43 classes. But this operation returned successful results.

2 Data Distribution of Training Dataset:

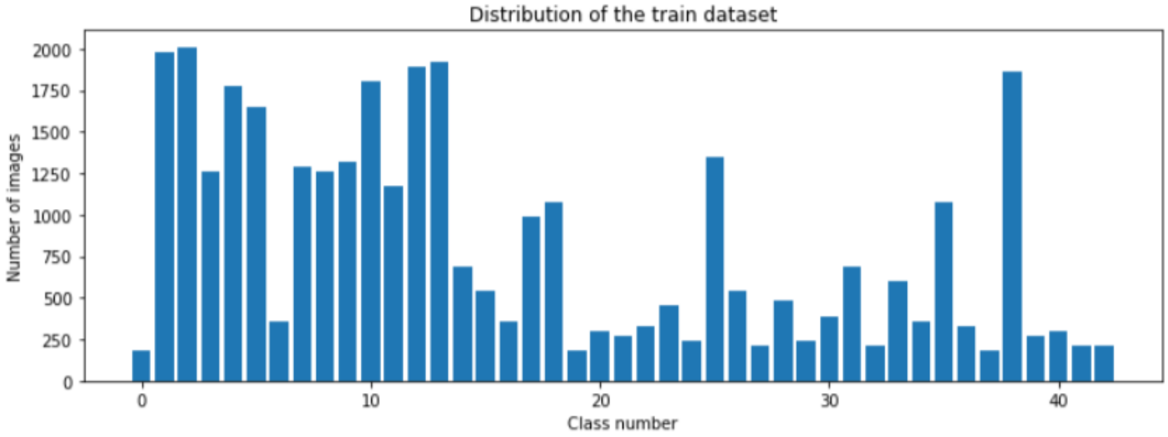
- To summarize and visualize how many training images are present for each class, we use following code:



```
print(num_of_samples)
plt.figure(figsize=(12, 4))
plt.bar(range(0, num_classes), num_of_samples)
plt.bar()
plt.title("Distribution of the train dataset")
plt.xlabel("Class number")
plt.ylabel("Number of images")
plt.show()
```

- In this code, we are just trying to plot the bar graph representation for the samples of each training class.
- After running this code snippet, we observed the following results:

```
[180, 1980, 2010, 1260, 1770, 1650, 360, 1290, 1260, 1320, 1800, 1170, 1890, 1920, 690, 540, 360, 9
```



- As we can clearly see from the above results, in the first part we have printed the training data set, which shows the number of samples present for each class.
- And then we displayed the dataset of our 43 training classes in the form of bar-graph.
- Maximum number of images for class is 2010 and minimum number of images is 180.
- We can clearly see that the distribution is not uniform. Some classes have more data, but some have very low dataset, therefore for some classes we will have less data for training and therefore, it will give us less accurate results.

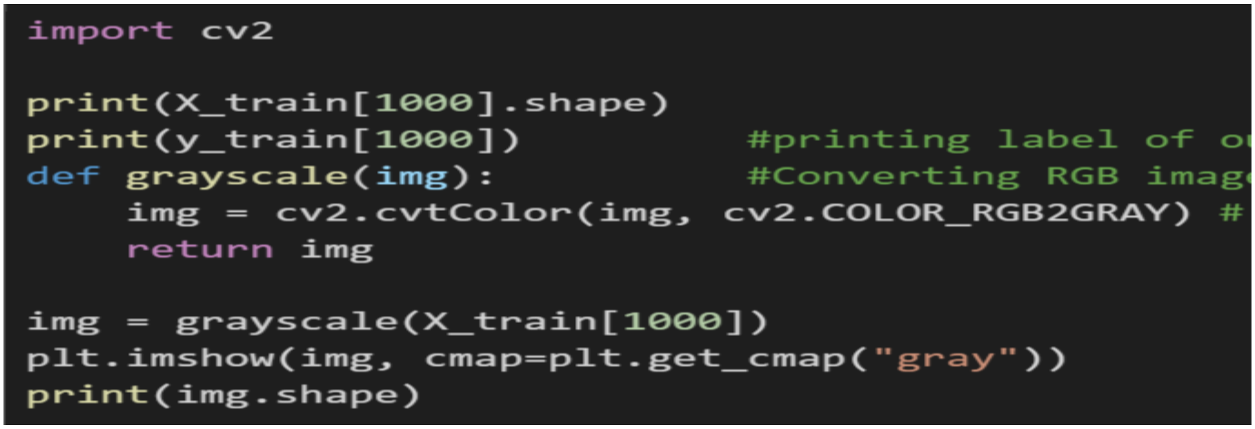
3 Preprocessing Data:

- Before designing and training our model, we need to preprocess our data.
- Our traffic sign images have RGB channels. That means we have a depth of 3. Therefore, there are variety of background and lighting conditions.
- Essentially each one of them contain unique extra features which makes it more difficult to classify them.
- Therefore, we need to preprocess our dataset, so that we can classify them appropriately.
- Let's use open source computer vision library "CV2" for this purpose.

Preprocessing Techniques:

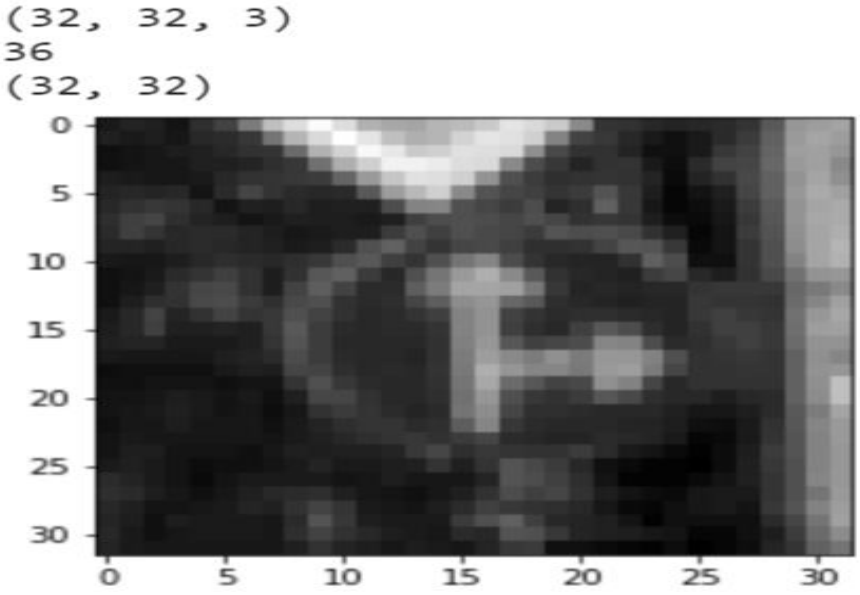
(i) Converting the RGB image to gray scale

- When distinguishing between traffic signs, color is not a very significant feature.
- The lighting in the images varies and many of the traffic signs have similar colors, reinforcing that it is not very relevant piece of information.
- The **features of the traffic signs** that really matter are the **edges the curves the shape and side of it**. That's what the network should focus on.
- Now this brings us to the next reason which is the fact that when we convert an image from our RGB to grayscale we reduce the depth of our image from three to one.
- This means that our network now requires fewer parameters as our input data will only have a depth of one channel in the long run.
- This means that our work will be much more efficient and will require less computing power to classify raw data.
- We achieved this using the following code.



```
import cv2

print(X_train[1000].shape)
print(y_train[1000])          #printing label of o
def grayscale(img):           #Converting RGB imag
    img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY) #
    return img

img = grayscale(X_train[1000])
plt.imshow(img, cmap=plt.get_cmap("gray"))
print(img.shape)
```

- As we can see, using the grayscale() function, we are converting the RGB image to grayscale image.
- The result obtained after running this code is as follows:

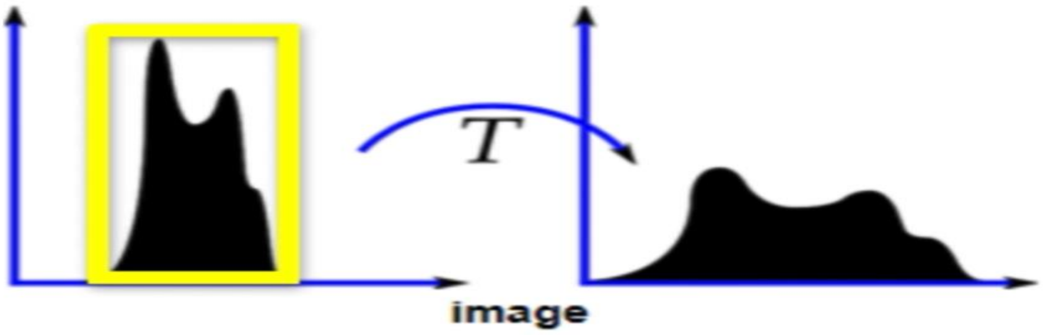


```
(32, 32, 3)
36
(32, 32)
```

- As we can see, on the first line the shape of the original image is presented which is 32 x 32 and is of 3 dimensions
- On the next line, the class number (36) is printed.
- And on the following lines the shape of the grayscale image obtained after passing through the grayscale() function and the grayscale image is displayed.

(ii) Histogram Equalization:

- Histogram equalization aims to standardize the lighting and all our images.
- While **some of our images are very bright and others very dim after histogram equalization these images will have a similar lighting effect**.
- The following image demonstrates how this process works.

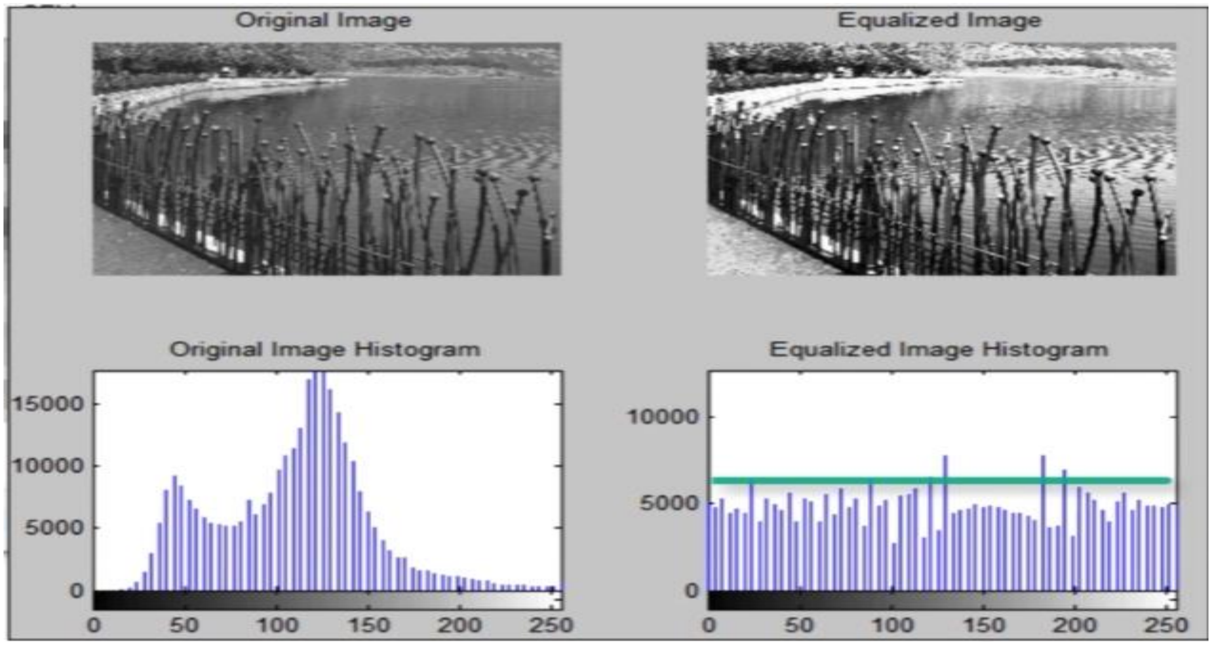


- The first figure is a histogram of a typical image which shows the number of pixels at each brightness value.

- We can see pixel intensities are bunched up around the small range of brightness values.

- Histogram equalization takes our histogram and spreads it at the ends to get a histogram that covers a higher range of brightness values and helps to normalize the lighting and all of our images.

- **This process also results in higher contrast with our image which can help with feature extraction.**

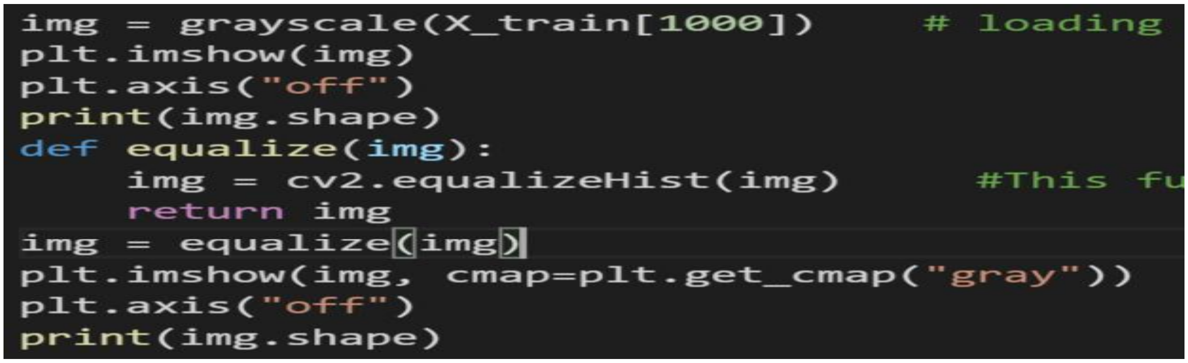


- **It enhances the contrast in the image that any greyscale intensities are now better distributed across the image at the same time deemphasizing any pixel intensities that occur at high frequencies.**

- And it does so theoretically by **flattening the resultant image histogram** as it reassigns grey values in the image.

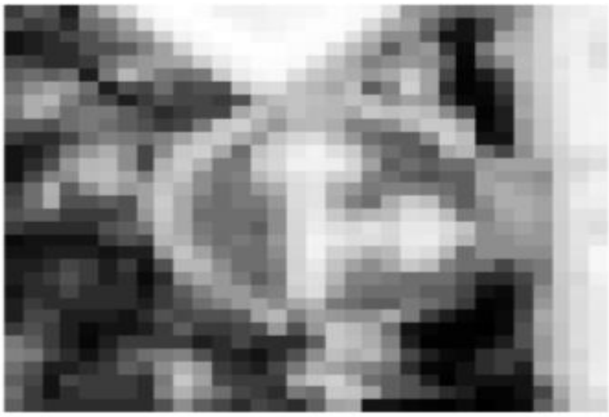
- Thus, instead of intensities being bundled up around a smaller region depending on our program, it often helps to reassign the intensity values of pixels in an image to contain a **more uniform distribution of intensities**.

- So, let's define an equalizer function which takes an image as an argument and we can use “CV2” to perform Instagram equalization in our image.



```
img = grayscale(X_train[1000])     # loading
plt.imshow(img)
plt.axis("off")
print(img.shape)
def equalize(img):
    img = cv2.equalizeHist(img)      #This fu
    return img
img = equalize(img)
plt.imshow(img, cmap=plt.get_cmap("gray"))
plt.axis("off")
print(img.shape)
```

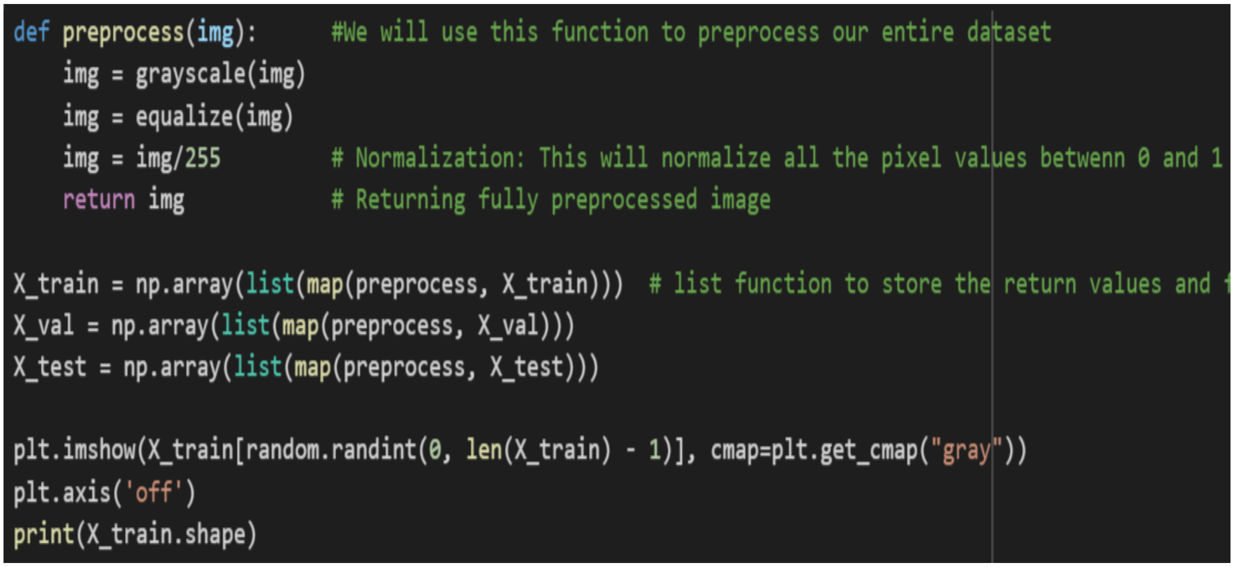
- So as shown in above code snippet, the function takes a single argument which is simply the image that is being modified.
- An important note is that, the equalizer function will only accept grayscale images as they don't have a depth.
- **As our RGB images have three color channels, this function will not be able to perform the histogram equalization properly.**
- This actually makes our initial preprocessing step all the more essential.
- So, let's put this function to use by making a call to it with every new greyscale image as the argument.
- So, we call equalize(img) and we are going to return the stored image inside a variable ‘img’.
- Running this code will effectively modify the histogram of our image improving the contrast. Results of running this snippet are as follows:



- Notice **the increased contrast and the more defined features in the histogram equalized image**.
- This will make it easier for our neural network to learn and extract these features.
- This gives us a clear demonstration of the benefits behind histogram equalization.

(iii) Preprocessing with Normalization:

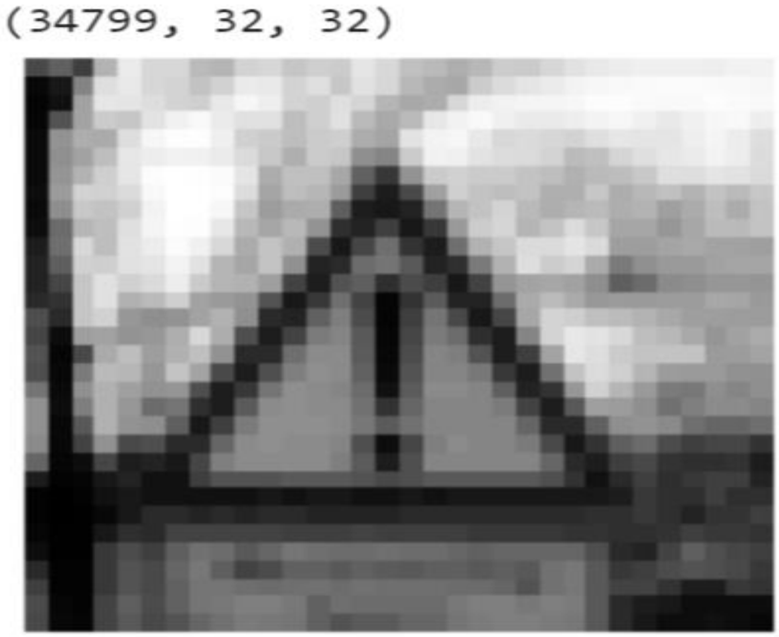
- And now that we visualized the effect of our preprocessing techniques, we need to apply these techniques to all of our images.
- So, we will define a function that does so, which will be used to preprocess our entire dataset. This process is shown below.



```
def preprocess(img):      #We will use this function to preprocess our entire dataset
    img = grayscale(img)
    img = equalize(img)
    img = img/255         # Normalization: This will normalize all the pixel values betwenn 0 and 1
    return img            # Returning fully preprocessed image

X_train = np.array(list(map(preprocess, X_train)))  # list function to store the return values and f
X_val = np.array(list(map(preprocess, X_val)))
X_test = np.array(list(map(preprocess, X_test)))

plt.imshow(X_train[random.randint(0, len(X_train) - 1)], cmap=plt.get_cmap("gray"))
plt.axis('off')
print(X_train.shape)
```

- As shown in figure, this function will take an image as an argument and inside the function, we will call our previous preprocessing functions namely grayscale and equalize.
- Now we also need to define another preprocessing technique which is called as normalization.
- **In normalization, we divide all the pixel intensities of our image by 255.**
- **This causes all the pixel values in our image to be normalized between 0 and 1.**
- **This is done simply by taking our image and dividing it by 255.**
- Having fully define our preprocessing function, now we need to run our dataset through it.
- But while running our datasets through the preprocessing function, we're going to make use of the ***'map'*** function.
- What that will do is iterate through the entire array and for every element of the array that it loops through, it returns a new element based on the specified function that is allocated for each item in the array.
- Ultimately creating a new array with updated values.
- Then, we iterate through our entire X_train datasets to ensure that all of our training images are processed correctly where each image from the array will go into the function as a parameter is preprocessed and then return as an updated value of the array.
- The map function will return the the preprocessed images into the format of a list and therefore we can easily use the list function to store the return values.

- And finally, we need to format this list as an array of images before we can store it inside of our X_train variable, and so we are using “numpy” array on our list.
- And finally, we can now store this inside of X_train array variable updating its values.
- We have now successfully preprocessed our entire training dataset, and now we can process our validation and test data as well in the same manner, storing them inside their respective arrays X_val and X_test.
- Then, we test and plot random image to verify our code is working properly which gave us following results.



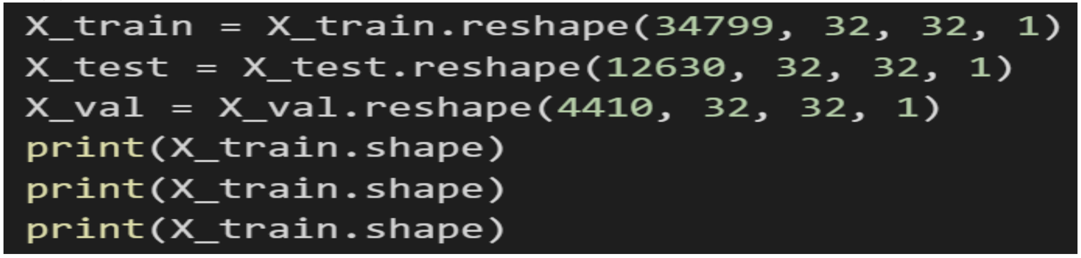
```
(34799, 32, 32)
```

  - So, we can see that on first line we printed size and shape of our preprocessed dataset, which shows we have preprocessed all the 34,799 images in the dataset and their dimensions are 32 x 32 with a depth of 1.

(iv) Reshaping The Dataset:

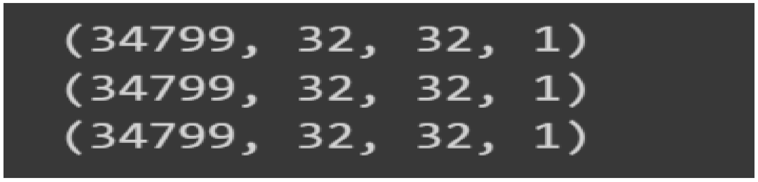
- So, the next step in preparing our data for use any convolutional network is to add depth to our data.
- At the moment, each image is two dimensional and it does not have a depth.
- And we know that the way in which the convolutional networks work is by applying a filter to the channels of the image that's being viewed.
- **In the case of grayscale images, there is one channel present therefore our data must reflect the presence of the depth.**
- And so, **by adding this depth our data will be in the desired shape to be used as an input for the convolutional layer**.

- This can be done by simply using **reshape function** as shown in the below code snippet.



```
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)
print(X_train.shape)
print(X_train.shape)
print(X_train.shape)
```

- Running this code snippet gives us following result.

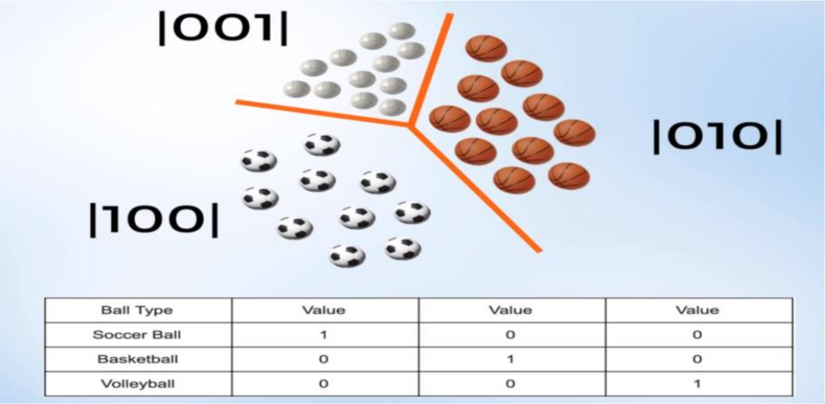


```
(34799, 32, 32, 1)
(34799, 32, 32, 1)
(34799, 32, 32, 1)
```

- This clearly shows that now there is a presence of depth in our dataset which is 1.

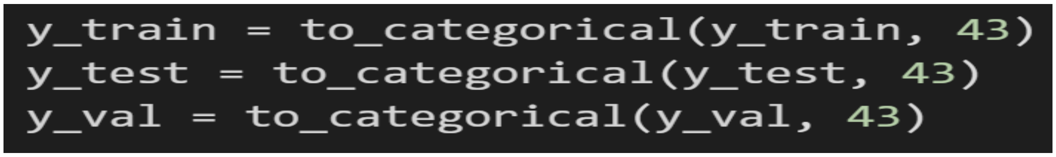
4 One-Hot Encoding Of The Labels:H

- The final step in preparing our data is one hot encoding for our data labels.
- One hot encoding allows us to classify all of our classes without assuming any dependence between the classes.
- One hot encoding works by creating separate columns for each data label in the data set and using an appropriate one or zero value for each column to identify the data class for our data.
- Consider the following table in which we're dealing with three classes.



Ball Type	Value	Value	Value
Soccer Ball	1	0	0
Basketball	0	1	0
Volleyball	0	0	1

- We create three columns. The presence of a class is presented in binary format.
- If something is a soccer ball it will be one hot encoded as 1 0 0.
- If it's a basketball it will be one hot encoded as 0 1 0.
- If it's a volleyball, then 0 0 1.
- As we can see, **this technique allows us to create independent labels for each of our data classes and ensures that our data does not assume any relation between our classes. They are linearly independent**.
- This can be done for any number of classes using the appropriate number of columns.
- We implemented the one hot encoding for our 43 classes using following code.



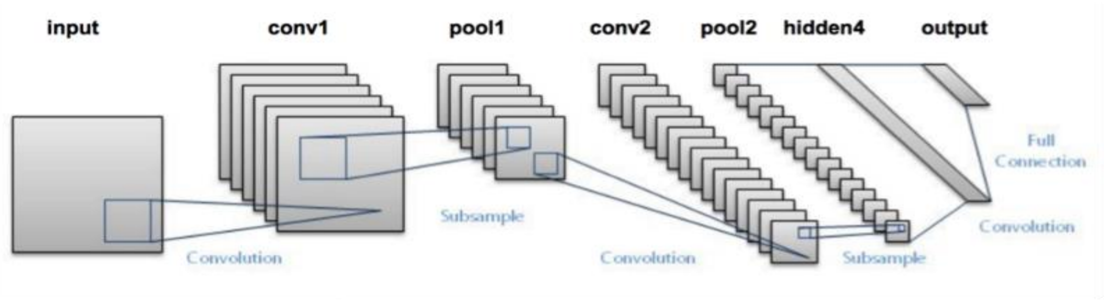
```
y_train = to_categorical(y_train, 43)
y_test = to_categorical(y_test, 43)
y_val = to_categorical(y_val, 43)
```

- **The to_categorical will take in two arguments the first being our labeled data that needs to be one hot encoded and the second is the total amount of classes within our dataset which in our case is 43, and then returns the one hot encoded labels, which we want for proceeding further.**

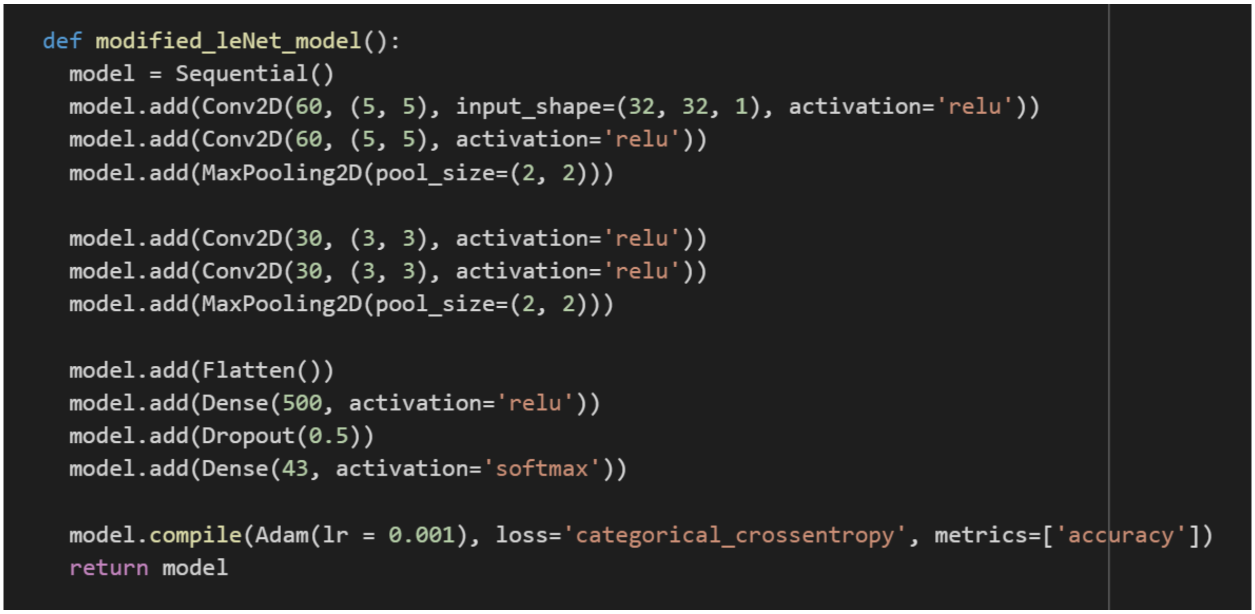
❖ Implementing Convolutional Neural Network Model:

Now, we have successfully preprocessed our data and are now ready to use our data to train and validate our ***"Convolutional Neural Network"***.

❖ leNet Model:



- The **leNet model consists of convolutional layers, pooling layers and fully connected layers** as shown above.
- We implemented it in our model as shown below.



```
def modified_leNet_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
```

Note: This is the snapshot for final implementation

The significance of each layer is as follows:

1. **Convolutional Layer:**
   The image is first processed in the convolutional layer where the convolved images that clearly detect specific features from the image are then operated on by the **relu** activati -on function to produce the final output.

2. **Pooling:**

- The images are then scaled into an abstracted form of the original feature map preserving the general pattern. Pooling operation scales down the feature map and reduces its dimensionality to prevent overfitting, yet still retain all of the important infor mation such that the features are invariant to small translations distortions and so on.

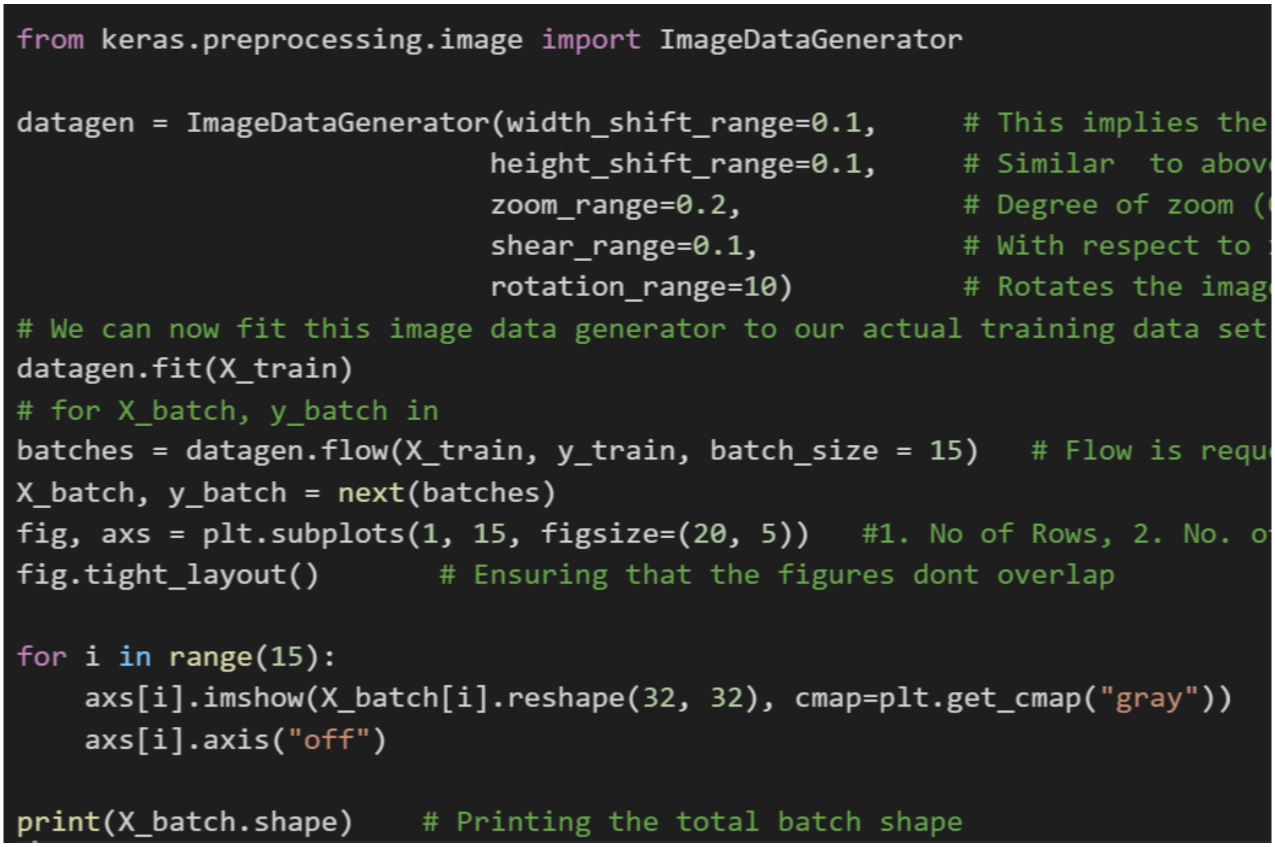
3. **Relu:**

- Used to add nonlinearity. The reason being any kind of neural network must contain no nlinearity since most of the realworld data that the neural network is required to learn is non-linear. It converts all the negative pixel values in the feature map to zeros.
- The reason behind selection of Relu function is the other activation functions like sigmoid or tan H are predisposed to a common artificial neural network problem called the vanishing gradient problem. At a very basic level, this problem refers to a decreased feedforward gradient within the deep neural network.
- Deep neural networks rely on gradient descent and back propagation techniques to adjust their weight parameters. This is essentially how they learn.
- However, if the gradient in the network gets too small or effectively vanishes then the network is unable to make use of gradient descent in an efficient way, and learning slo ws down significantly.
- If the gradient is small, the network can only adjust weight parameters by a very small a mount with each update which means the network learns very slowly.
- This problem affects the sigmoid function because the sigmoid function has a relatively low gradient, as it levels at extreme values of infinity and negative infinity.
- The low gradient becomes an issue as within a neural network the gradient value is fed forward through the network and multiplied with additional gradient values.
- If the sigmoid activation function is used then all these gradient values are limited between the range of 0 and point to some small value (e.g. 0.5). As we are now multiply ing many decimals together creates a smaller and smaller number.
- Therefore, as the network feeds forward the gradient in the network seems to vanish to a very low number.
- This issue can be fixed by using the rectified linear unit.The relu activation function. The relu function returns a value of 0 if the input is negative.
- However, if the input is positive then the output that returned is just the same value. The key thing to notice though is that the gradient value of the relu function is either 0 or it is 1.
- In the case of a gradient of 1, We know that the vanishing gradient problem cannot occur as multiplying by 1 does not cause the gradient values to decrease.

Note: We added flattening layer in order to convert our data obtained from pooling layer into 1-dimensional format as we will be feeding it to the fully connected layer.

❖ Image Augmentation using Fit Generator:

- As we already know that our dataset is smaller containing fewer images in the training set and validation set and the test set. Also, the dataset contains many classes (total 43) in where the traffic signs can be classified. This means not only our dataset is smaller, but also, we have fewer images belonging to each class since the dataset is being classified into many classes and the dataset is complex. This makes it difficult for our network to learn properly for the high accuracy rates.
- Thus, we need to increase our training dataset. We can do this by using Image Augmentation using Fit Generator. Its implementation is shown below.

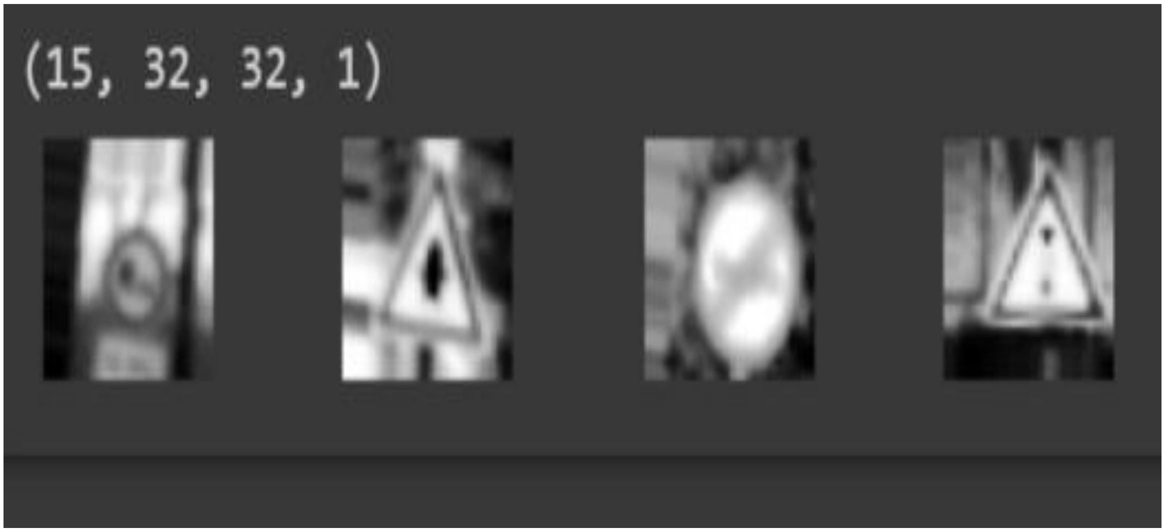


```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(width_shift_range=0.1,     # This implies the
                             height_shift_range=0.1,    # Similar  to abov
                             zoom_range=0.2,            # Degree of zoom (
                             shear_range=0.1,           # With respect to 
                             rotation_range=10)         # Rotates the imag
# We can now fit this image data generator to our actual training data set
datagen.fit(X_train)
# for X_batch, y_batch in
batches = datagen.flow(X_train, y_train, batch_size = 15)   # Flow is requ
X_batch, y_batch = next(batches)
fig, axs = plt.subplots(1, 15, figsize=(20, 5))   #1. No of Rows, 2. No. o
fig.tight_layout()       # Ensuring that the figures dont overlap

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

print(X_batch.shape)    # Printing the total batch shape
```

- The above code snippet can be explained as follows:
  - ➢ **ImageDataGenerator**: This class will allow us to define the types of transformations we wantto make to our data sets, and it will also allow us to set range limits for adju sting how extreme these transformations can get.
- We can now fit this image data generator to our actual training X_train data set using datagen.fit() function which can be explained as follows:
  - ➢ datagen.fit() : ***The interesting thing to note about the image data generator is that it's an iterator which only returns batches of transformed images when requested. This means that the data generator does not perform operations on all the images***

    - ***in our data sets as that would be very memory intensive, due to all the new images being created.***
    - ***Instead it only creates these new augmented images in real time what are requested***
    - ***This reduces overall memory requirements. However, it leads to additional time cost during the training process as that is when the data will be created.***
- After this we created batches of augmented image dataset using next iterator which can be explained as follows:
    - next() : We need to take our batch iterator containing data about our newly created images and store this data inside of an X_batch and respective y_batch array.
    - This is done using the built in Python command '***next'***. And the next function simply calls an iterator item and retrieves the next item for it.
    - As we defined our iterator to create a batch size of 15, we get 15 new images each time the next() function is called on the iterator.

- After running this code snippet, we obtained the following results:

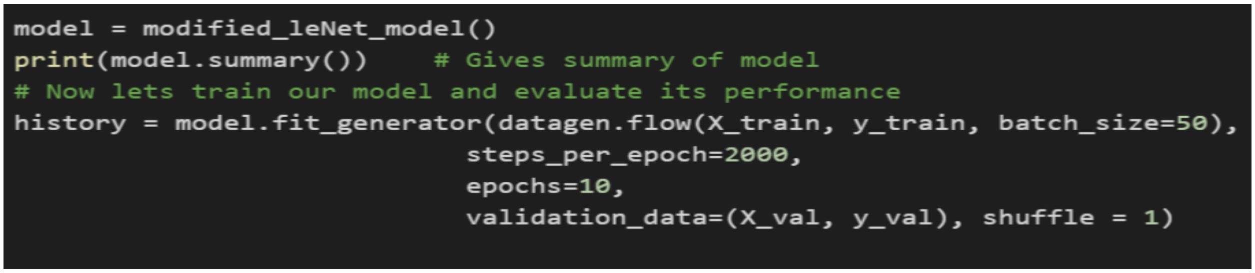


```
(15, 32, 32, 1)
```

- The first line shows the number of augmented images which is 15, then the dimensions and depths of the augmented images which are 32 x 32 and 1, respectively.
- And running this code displays our brand new 15 images that our image generator created.
- For better visualization I have shown only four images out of 15.
- They have been augmented in different ways to help add variety to our datasets.
- As we can see these images do not look like our typical traffic sign dataset images.
- Some have been zoomed in or out while others have been rotated sheared or shifted.
- For the most part many of these images have been transformed in more than one way.
- This means that our data augmentation process was indeed effective and will be a useful training tool for our model.

❖ Training Model and Evaluating its performance:

- Now let’s train our model and evaluate its performance.
- We did this using following code snippet.



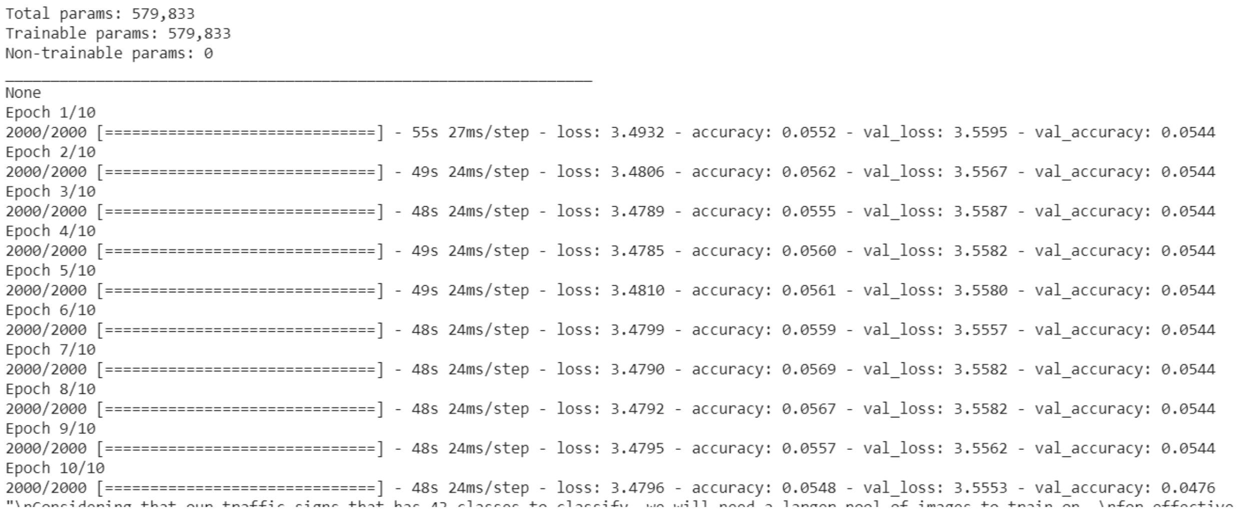
```
model = modified_leNet_model()
print(model.summary())    # Gives summary of model
# Now lets train our model and evaluate its performance
history = model.fit_generator(datagen.flow(X_train, y_train, batch_size=50),
                            steps_per_epoch=2000,
                            epochs=10,
                            validation_data=(X_val, y_val), shuffle = 1)
```

- Notice the use of fit generator for training our model in the above code snippet. Its functionality can be explained as follows:
  - ➢ We are using the model.fit_generator() statements, which is capable of running our Image generator in parallel with the model training process.
  - ➢ The statement will train our model using the augmented data produced by our gene rator.
  - ➢ Also notice the use of datagen.flow() statement.
  - ➢ Considering that our traffic signs that has 43 classes to classify, we needed a larger pool of images to train on, for effective training. So, we used a dataset of 10,000 images.
  - ➢ And for a batch size of 50, we needed a step size of 2000.
  - ➢ So, **2000 batches times 50 images per batch to get a total of 100,000 images per epoch. (2000 batches * 50 images/batch = 100,000 images/epoch)**.

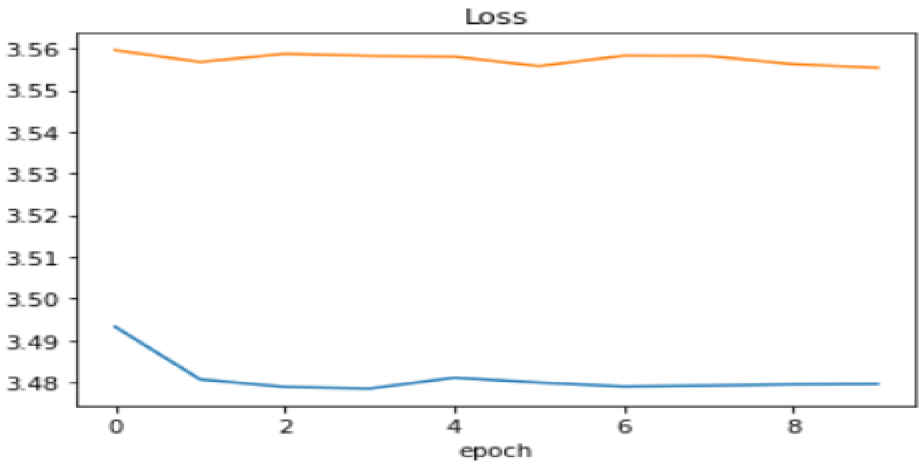
Let’s compare model performance for different parameter configurations.

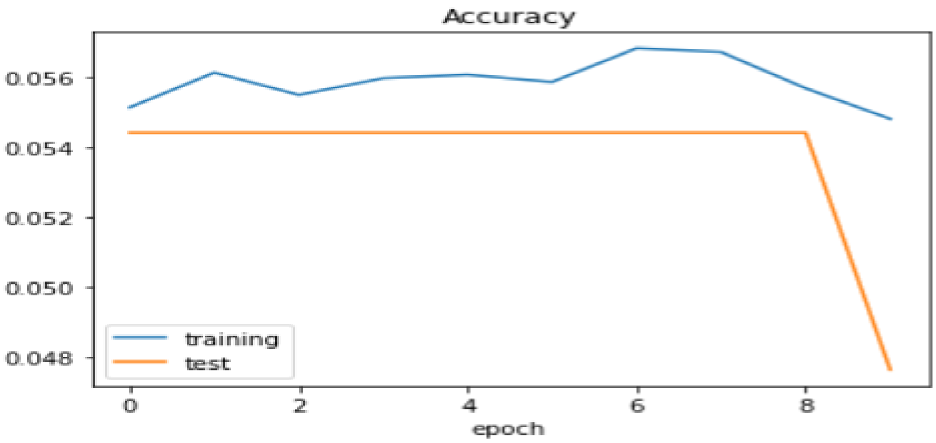
Parameters:

Learning rate: 0.01, **Convolutional Layers:** 2, **Activation Function:** sigmoid, **Dropout Layers:** 0



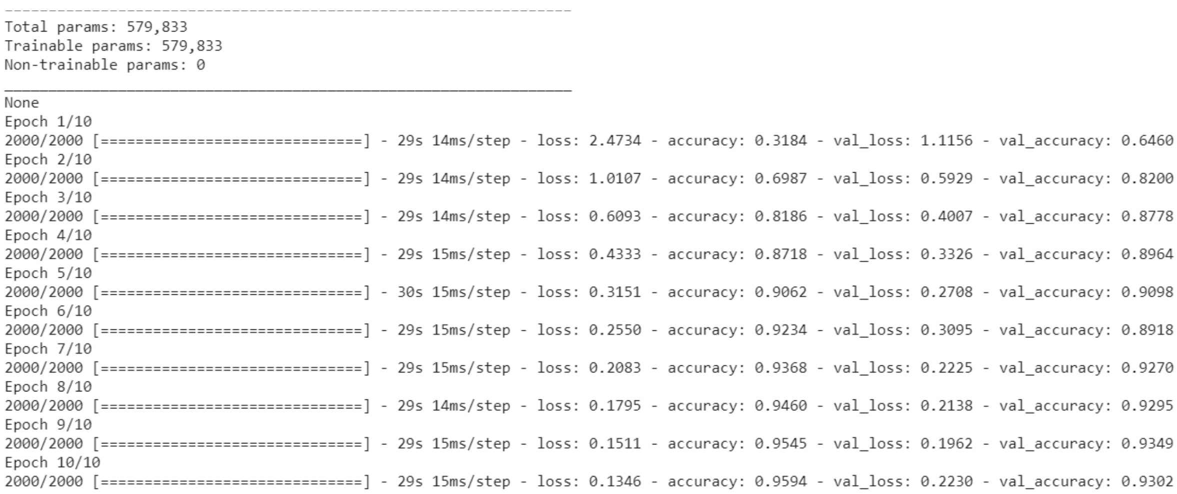
```
Total params: 579,833
Trainable params: 579,833
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/10
2000/2000 [==============================] - 55s 27ms/step - loss: 3.4932 - accuracy: 0.0552 - val_loss: 3.5595 - val_accuracy: 0.0544
Epoch 2/10
2000/2000 [==============================] - 49s 24ms/step - loss: 3.4806 - accuracy: 0.0562 - val_loss: 3.5567 - val_accuracy: 0.0544
Epoch 3/10
2000/2000 [==============================] - 48s 24ms/step - loss: 3.4789 - accuracy: 0.0555 - val_loss: 3.5587 - val_accuracy: 0.0544
Epoch 4/10
2000/2000 [==============================] - 49s 24ms/step - loss: 3.4785 - accuracy: 0.0560 - val_loss: 3.5582 - val_accuracy: 0.0544
Epoch 5/10
2000/2000 [==============================] - 49s 24ms/step - loss: 3.4810 - accuracy: 0.0561 - val_loss: 3.5580 - val_accuracy: 0.0544
Epoch 6/10
2000/2000 [==============================] - 48s 24ms/step - loss: 3.4799 - accuracy: 0.0559 - val_loss: 3.5557 - val_accuracy: 0.0544
Epoch 7/10
2000/2000 [==============================] - 48s 24ms/step - loss: 3.4790 - accuracy: 0.0569 - val_loss: 3.5582 - val_accuracy: 0.0544
Epoch 8/10
2000/2000 [==============================] - 48s 24ms/step - loss: 3.4792 - accuracy: 0.0567 - val_loss: 3.5582 - val_accuracy: 0.0544
Epoch 9/10
2000/2000 [==============================] - 48s 24ms/step - loss: 3.4795 - accuracy: 0.0557 - val_loss: 3.5562 - val_accuracy: 0.0544
Epoch 10/10
2000/2000 [==============================] - 48s 24ms/step - loss: 3.4796 - accuracy: 0.0548 - val_loss: 3.5553 - val_accuracy: 0.0476
"\nConsidering that our traffic signs that has 43 classes to classify, we will need a larger pool of images to train on. \nfor effective
```



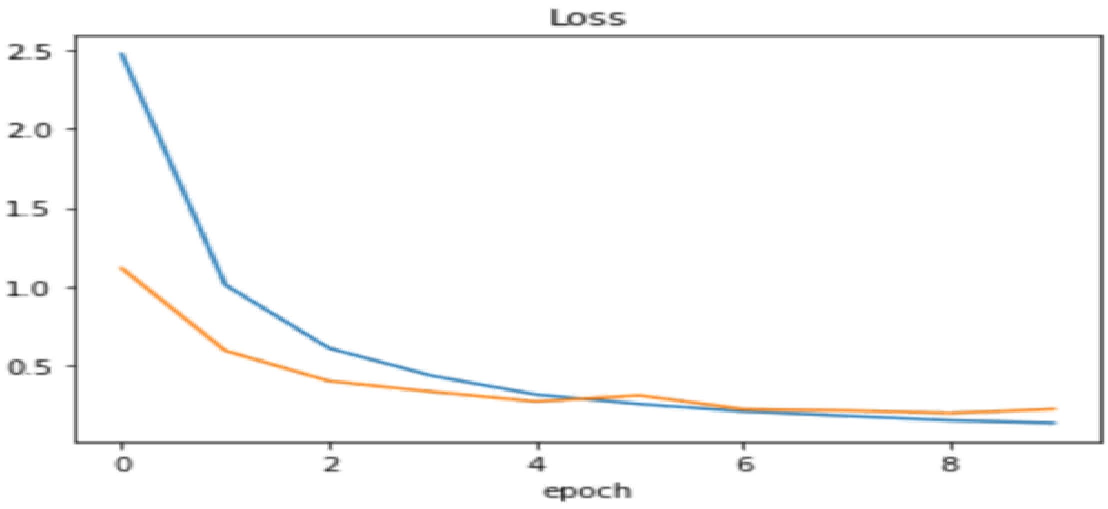


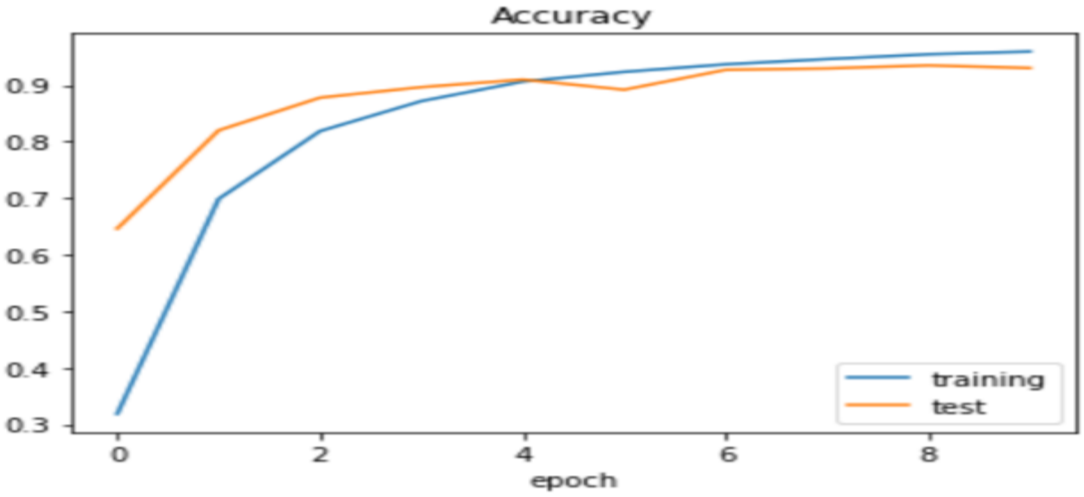
Parameters:

Learning rate: 0.001, **Convolutional Layers:** 4, **Activation Function:** sigmoid, **Dropout Layers:** 0



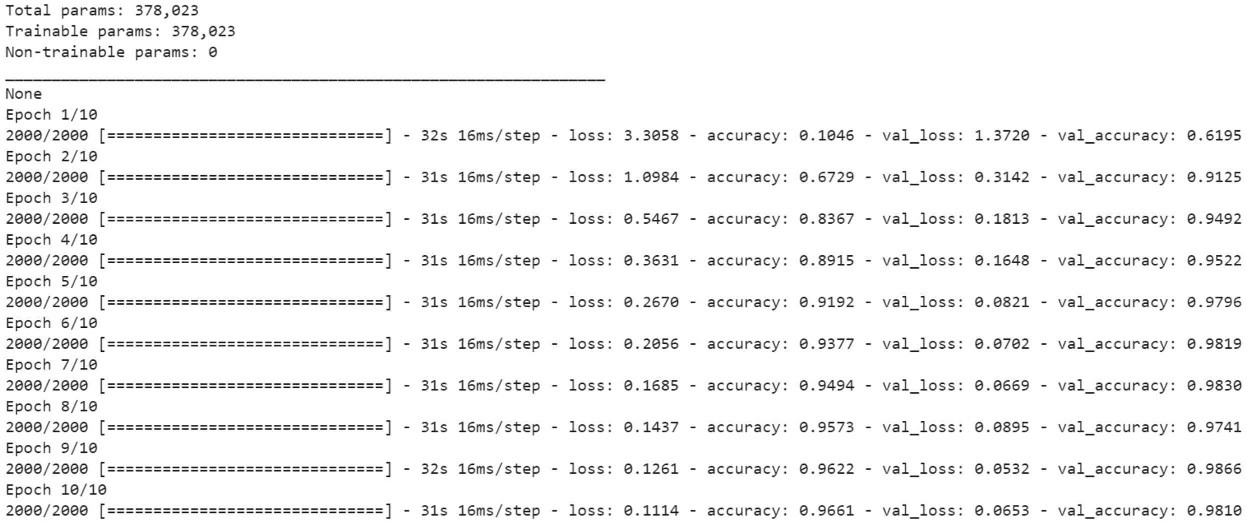
```
_________________________________________________________________
Total params: 579,833
Trainable params: 579,833
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/10
2000/2000 [==============================] - 29s 14ms/step - loss: 2.4734 - accuracy: 0.3184 - val_loss: 1.1156 - val_accuracy: 0.6460
Epoch 2/10
2000/2000 [==============================] - 29s 14ms/step - loss: 1.0107 - accuracy: 0.6987 - val_loss: 0.5929 - val_accuracy: 0.8200
Epoch 3/10
2000/2000 [==============================] - 29s 14ms/step - loss: 0.6093 - accuracy: 0.8186 - val_loss: 0.4007 - val_accuracy: 0.8778
Epoch 4/10
2000/2000 [==============================] - 29s 15ms/step - loss: 0.4333 - accuracy: 0.8718 - val_loss: 0.3326 - val_accuracy: 0.8964
Epoch 5/10
2000/2000 [==============================] - 30s 15ms/step - loss: 0.3151 - accuracy: 0.9062 - val_loss: 0.2708 - val_accuracy: 0.9098
Epoch 6/10
2000/2000 [==============================] - 29s 15ms/step - loss: 0.2550 - accuracy: 0.9234 - val_loss: 0.3095 - val_accuracy: 0.8918
Epoch 7/10
2000/2000 [==============================] - 29s 15ms/step - loss: 0.2083 - accuracy: 0.9368 - val_loss: 0.2225 - val_accuracy: 0.9270
Epoch 8/10
2000/2000 [==============================] - 29s 14ms/step - loss: 0.1795 - accuracy: 0.9460 - val_loss: 0.2138 - val_accuracy: 0.9295
Epoch 9/10
2000/2000 [==============================] - 29s 15ms/step - loss: 0.1511 - accuracy: 0.9545 - val_loss: 0.1962 - val_accuracy: 0.9349
Epoch 10/10
2000/2000 [==============================] - 29s 15ms/step - loss: 0.1346 - accuracy: 0.9594 - val_loss: 0.2230 - val_accuracy: 0.9302
```



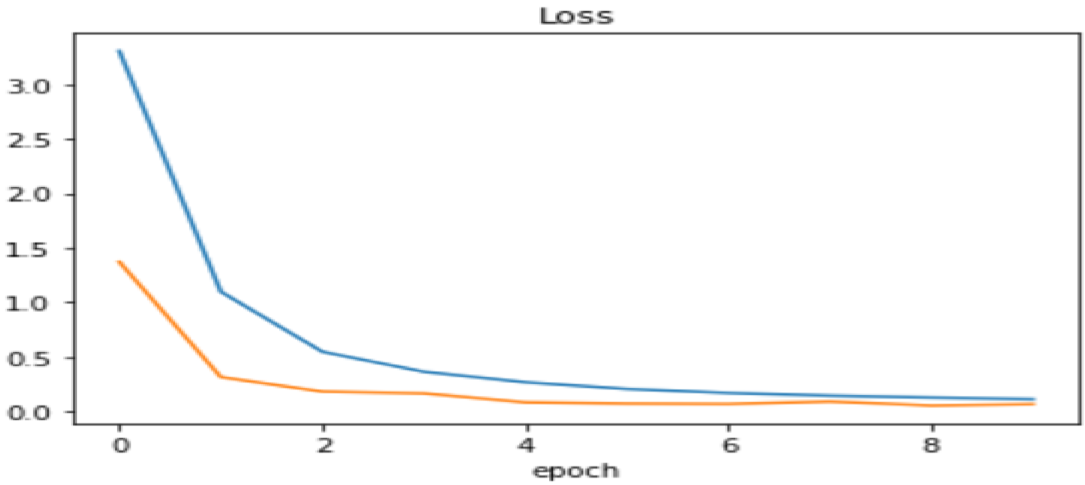


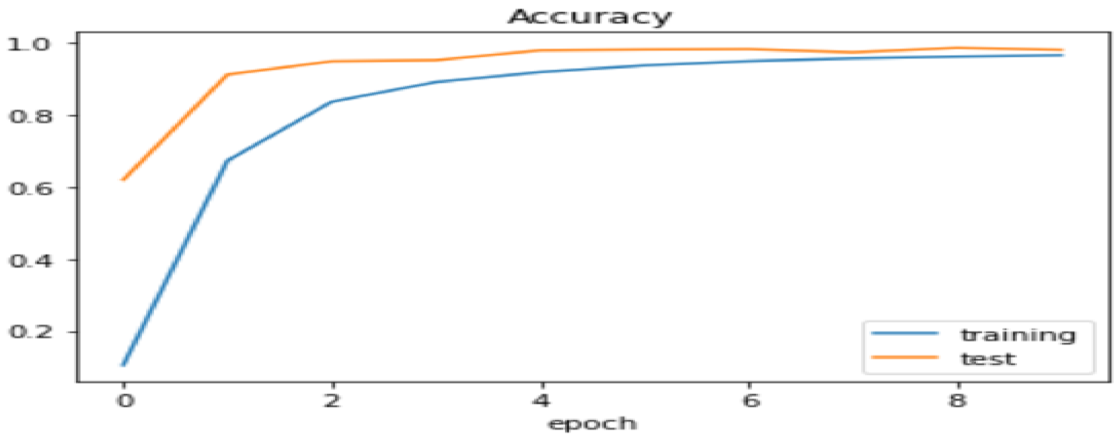
Parameters:

Learning rate: 0.001, **Convolutional Layers:** 4, **Activation Function:** Sigmoid, **Dropout Layers:** 1



```
Total params: 378,023
Trainable params: 378,023
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/10
2000/2000 [==============================] - 32s 16ms/step - loss: 3.3058 - accuracy: 0.1046 - val_loss: 1.3720 - val_accuracy: 0.6195
Epoch 2/10
2000/2000 [==============================] - 31s 16ms/step - loss: 1.0984 - accuracy: 0.6729 - val_loss: 0.3142 - val_accuracy: 0.9125
Epoch 3/10
2000/2000 [==============================] - 31s 16ms/step - loss: 0.5467 - accuracy: 0.8367 - val_loss: 0.1813 - val_accuracy: 0.9492
Epoch 4/10
2000/2000 [==============================] - 31s 16ms/step - loss: 0.3631 - accuracy: 0.8915 - val_loss: 0.1648 - val_accuracy: 0.9522
Epoch 5/10
2000/2000 [==============================] - 31s 16ms/step - loss: 0.2670 - accuracy: 0.9192 - val_loss: 0.0821 - val_accuracy: 0.9796
Epoch 6/10
2000/2000 [==============================] - 31s 16ms/step - loss: 0.2056 - accuracy: 0.9377 - val_loss: 0.0702 - val_accuracy: 0.9819
Epoch 7/10
2000/2000 [==============================] - 31s 16ms/step - loss: 0.1685 - accuracy: 0.9494 - val_loss: 0.0669 - val_accuracy: 0.9830
Epoch 8/10
2000/2000 [==============================] - 31s 16ms/step - loss: 0.1437 - accuracy: 0.9573 - val_loss: 0.0895 - val_accuracy: 0.9741
Epoch 9/10
2000/2000 [==============================] - 32s 16ms/step - loss: 0.1261 - accuracy: 0.9622 - val_loss: 0.0532 - val_accuracy: 0.9866
Epoch 10/10
2000/2000 [==============================] - 31s 16ms/step - loss: 0.1114 - accuracy: 0.9661 - val_loss: 0.0653 - val_accuracy: 0.9810
```



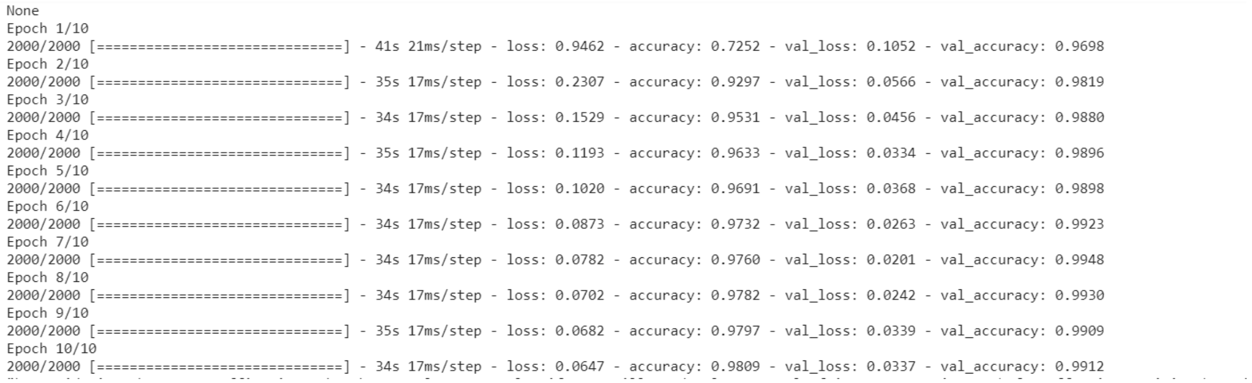


Parameters:

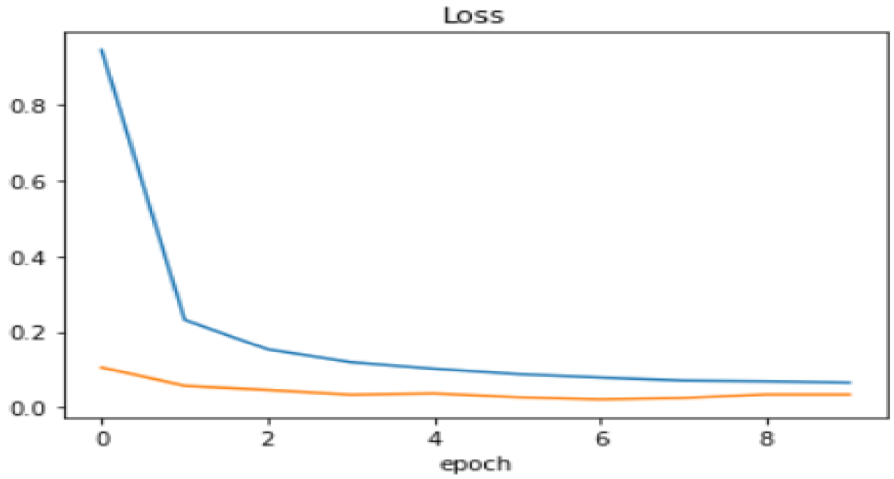
Learning rate: 0.001, **Convolutional Layers:** 4, **Activation Function:** Sigmoid, **Dropout Layers:** 0

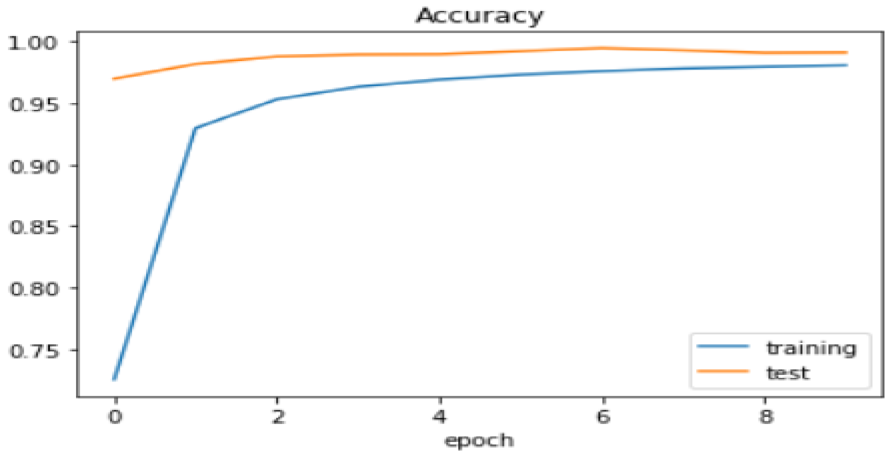
Parameters:

Learning rate: 0.001, **Convolutional Layers:** 4, **Activation Function:** sigmoid, **Dropout Layers:** 1



```
None
Epoch 1/10
2000/2000 [==============================] - 41s 21ms/step - loss: 0.9462 - accuracy: 0.7252 - val_loss: 0.1052 - val_accuracy: 0.9698
Epoch 2/10
2000/2000 [==============================] - 35s 17ms/step - loss: 0.2307 - accuracy: 0.9297 - val_loss: 0.0566 - val_accuracy: 0.9819
Epoch 3/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.1529 - accuracy: 0.9531 - val_loss: 0.0456 - val_accuracy: 0.9880
Epoch 4/10
2000/2000 [==============================] - 35s 17ms/step - loss: 0.1193 - accuracy: 0.9633 - val_loss: 0.0334 - val_accuracy: 0.9896
Epoch 5/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.1020 - accuracy: 0.9691 - val_loss: 0.0368 - val_accuracy: 0.9898
Epoch 6/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.0873 - accuracy: 0.9732 - val_loss: 0.0263 - val_accuracy: 0.9923
Epoch 7/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.0782 - accuracy: 0.9760 - val_loss: 0.0201 - val_accuracy: 0.9948
Epoch 8/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.0702 - accuracy: 0.9782 - val_loss: 0.0242 - val_accuracy: 0.9930
Epoch 9/10
2000/2000 [==============================] - 35s 17ms/step - loss: 0.0682 - accuracy: 0.9797 - val_loss: 0.0339 - val_accuracy: 0.9909
Epoch 10/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.0647 - accuracy: 0.9809 - val_loss: 0.0337 - val_accuracy: 0.9912
```





❖ Improving Model Performance:

- **Comparison Table:**

Sr. No.	Learning Rate	Convolutional layers	Activation Function	Dropout Layers	Training Loss	Training Accuracy(%)	Validation Loss	Validation Accuracy(%)
1	0.01	2	Sigmoid	0	3.4796	5.48	3.553	4.76
2	0.001	4	Sigmoid	0	0.1346	95.94	0.2230	93.02
3	0.001	4	sigmoid	1	0.1114	96.61	0.0653	98.10
4	0.001	4	relu	1	0.0647	98.09	0.0337	99.12

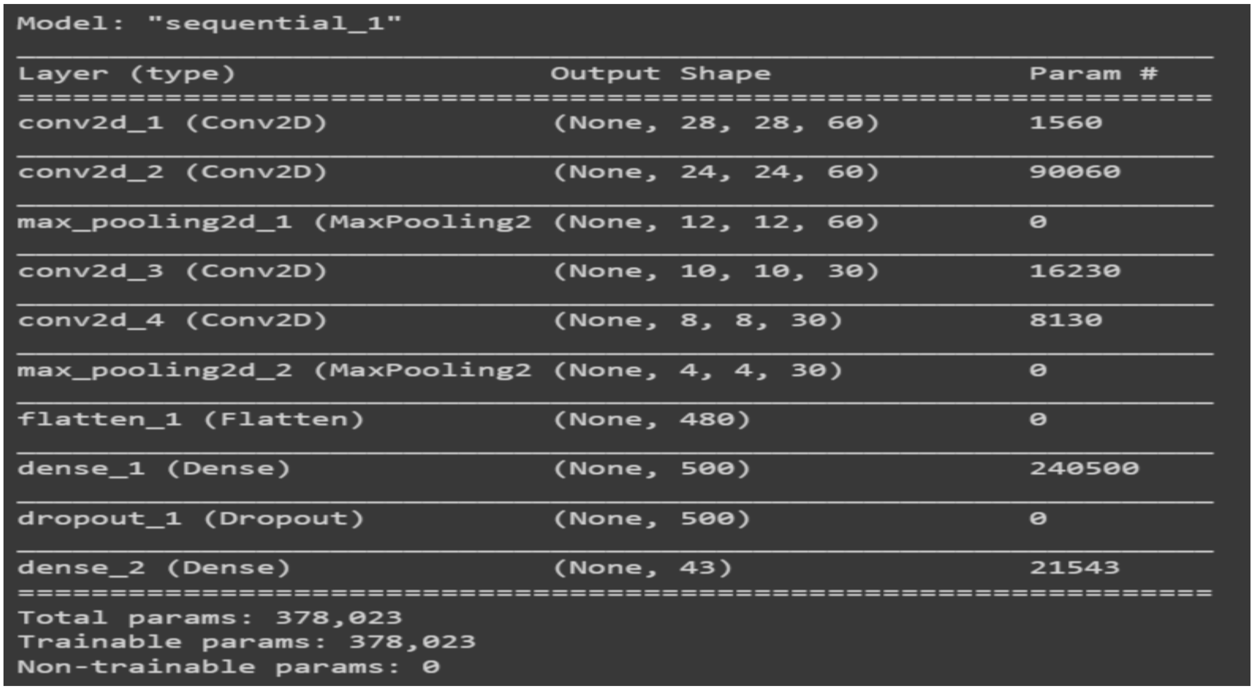
Table. Comparison Table for different configurations of model

1. **Changing the learning rate (Adam):**
   Although the Adam optimizer computes individual adaptive learning rates, it is important to specify a fitting initial learning rate for optimal performance.
   A learning rate that is too high can often lead to a lower accuracy.
   ***However, a lower learning rate can help a neural network learn more effectively when a more complex dataset is involved*.**
   (In our model, ***we changed our learning rate from 0.01 to 0.001 and got improvements in accuracy***.)

2. **Increasing the number of filters inside your convolutional layer:**
   It can ***help our network extract more features from the images and can result in improved accuracy.***
   This ***requires more parameters and therefore more computing power, but it is a necessary modification for improving our network's performance****.*
3. **Adding additional Convolutional layers:**
   And as you can see our convolutional layers come with their own parameters which means that more computing power is required when these layers are added.
   However, a very interesting point to note is that while these additional layers introduce their own parameters the total number of parameters in our model will be decreased. (Our previous model had a total of about 5,78,000 parameters and our new modified model only has a total of 3,70,000 parameters.)
   The reason for this is the fact that ***with each convolutional layer, the dimensions of our image decrease***. This means that by that time our image data reaches over fully connected layers; it has a much smaller dimension. This results in few work parameters within the fully connected layer which results in less overall parameters within our model. Therefore, with this new modification our model will require less computing power and will potentially improve our accuracy.
   (In our model this improved our training accuracy as well as validation accuracy).

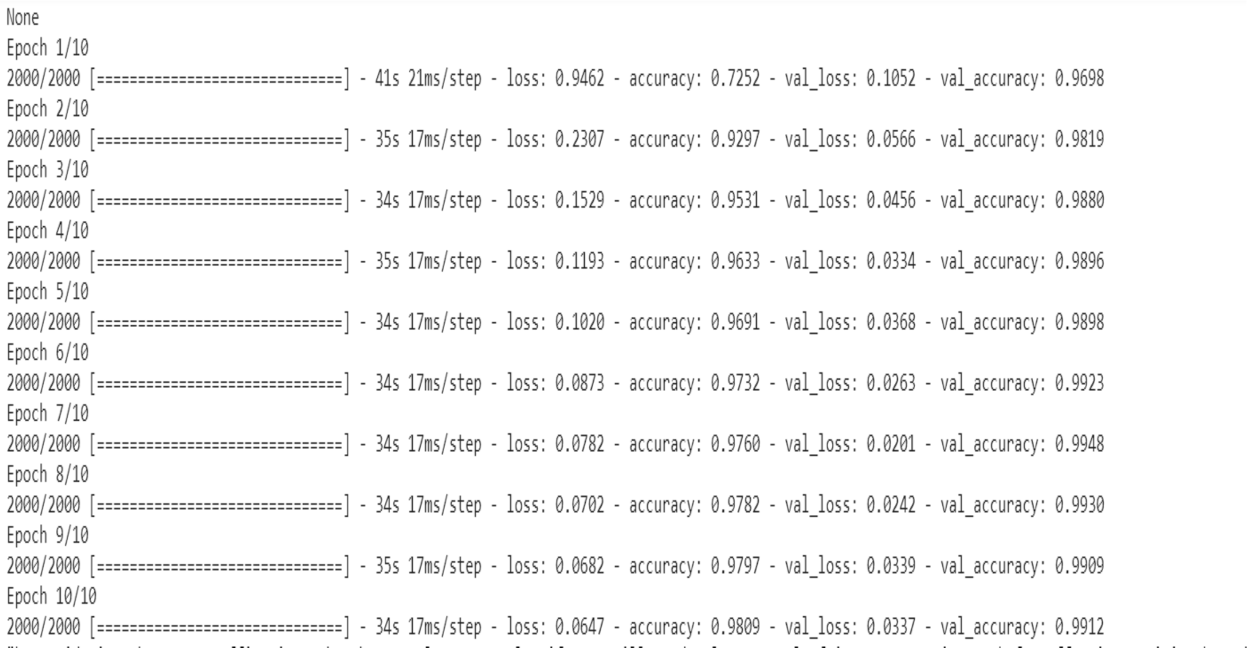
4. Our previous modifications improved our overall accuracy but there ***is still a problem of overfitting***. To tackle this issue, ***we can add Dropout layers***.
   Dropping the layers are an effective way of preventing overfitting. We already had a drop out layer inside of our model. However, as overfitting is present, we can attempt to prevent it by adding another dropout layer. Using more than one drop out layer is common and can be a very effective technique.
   Note: Due to too many dropout layers in the model, it might result in larger gap between training loss accuracy and validation loss accuracy. In that case, we can eliminate few dropout layers to shorten the gap.
   It's important to realize often times that each small modification has a small impact on improving our accuracy. However, when all of these small modifications are used alongside each other, the effects can be quite significant.

❖ **Final Model Parameters:**



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 60)        1560
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 60)        90060
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 60)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 10, 10, 30)        16230
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 30)          8130
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 30)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 480)               0
_________________________________________________________________
dense_1 (Dense)              (None, 500)               240500
_________________________________________________________________
dropout_1 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_2 (Dense)              (None, 43)                21543
=================================================================
Total params: 378,023
Trainable params: 378,023
Non-trainable params: 0
```

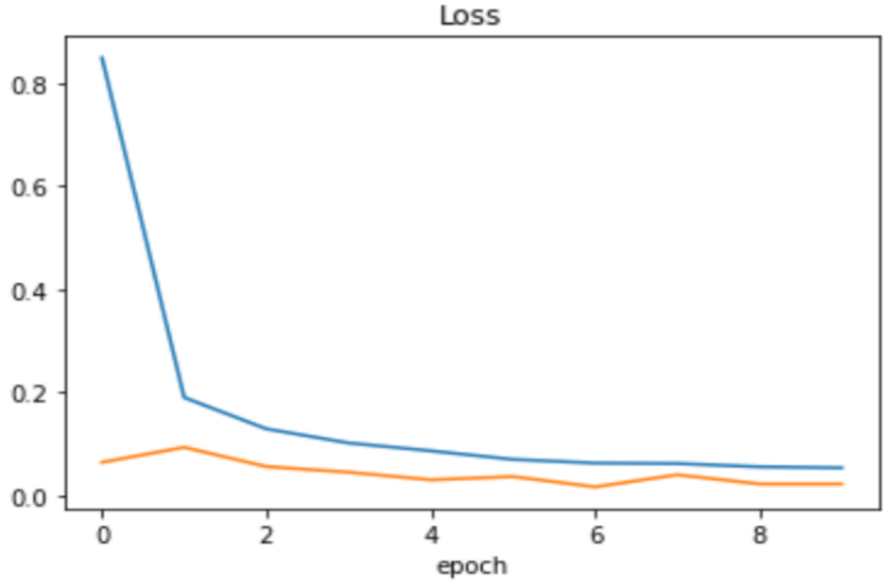
- This snapshot shows that we got the expected model summary.
- Also, we get the summary of each of the 10 epochs which is as shown below.

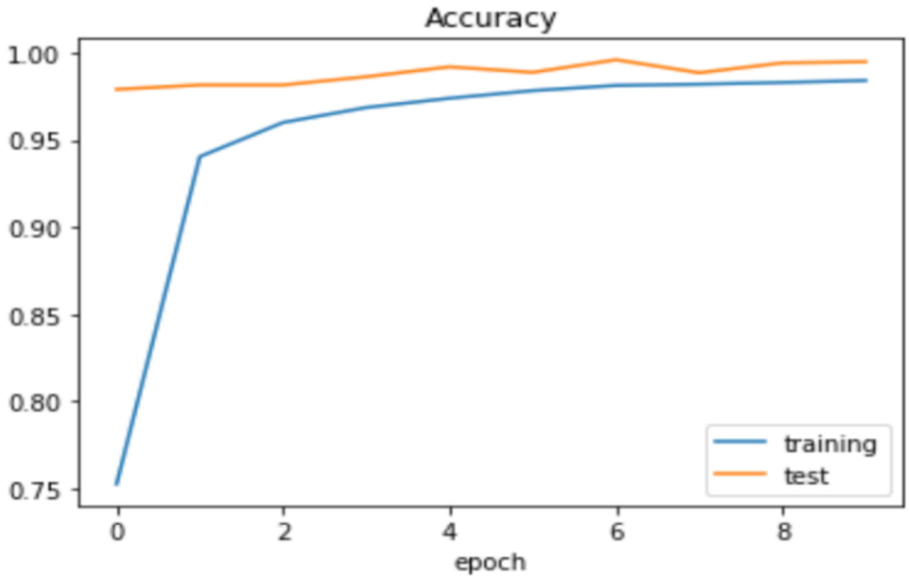


```
None
Epoch 1/10
2000/2000 [==============================] - 41s 21ms/step - loss: 0.9462 - accuracy: 0.7252 - val_loss: 0.1052 - val_accuracy: 0.9698
Epoch 2/10
2000/2000 [==============================] - 35s 17ms/step - loss: 0.2307 - accuracy: 0.9297 - val_loss: 0.0566 - val_accuracy: 0.9819
Epoch 3/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.1529 - accuracy: 0.9531 - val_loss: 0.0456 - val_accuracy: 0.9880
Epoch 4/10
2000/2000 [==============================] - 35s 17ms/step - loss: 0.1193 - accuracy: 0.9633 - val_loss: 0.0334 - val_accuracy: 0.9896
Epoch 5/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.1020 - accuracy: 0.9691 - val_loss: 0.0368 - val_accuracy: 0.9898
Epoch 6/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.0873 - accuracy: 0.9732 - val_loss: 0.0263 - val_accuracy: 0.9923
Epoch 7/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.0782 - accuracy: 0.9760 - val_loss: 0.0201 - val_accuracy: 0.9948
Epoch 8/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.0702 - accuracy: 0.9782 - val_loss: 0.0242 - val_accuracy: 0.9930
Epoch 9/10
2000/2000 [==============================] - 35s 17ms/step - loss: 0.0682 - accuracy: 0.9797 - val_loss: 0.0339 - val_accuracy: 0.9909
Epoch 10/10
2000/2000 [==============================] - 34s 17ms/step - loss: 0.0647 - accuracy: 0.9809 - val_loss: 0.0337 - val_accuracy: 0.9912
```

❖ From this summary we can see that with each epoch, our accuracy for training and validation set improved and the validation loss reduced which shows that we are able to train our model to a great extent.

- **Plotting Loss and Accuracy plot to observe the Efficiency of model for Training Data:**

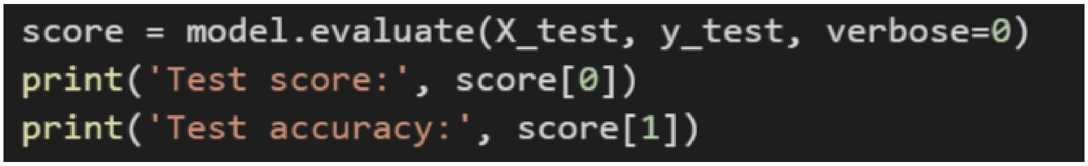




- From these plots, we can see that we have a much smaller gap between our training loss and training accuracy as well as our validation loss and validation accuracy, respectively.
- And this demonstrates consistency in our training and implies a better trained model and we now finish our model training with a validation accuracy of over 99 percent and a training accuracy of 98 percent which is higher than what our model could provide without the fit generator.

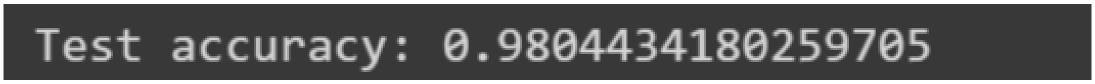
Evaluating model performance for Test data:

- In order to evaluate accuracy of the model for test data, we used the following code.



```
score = model.evaluate(X_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

- After running this code snippet, we observed the following results.

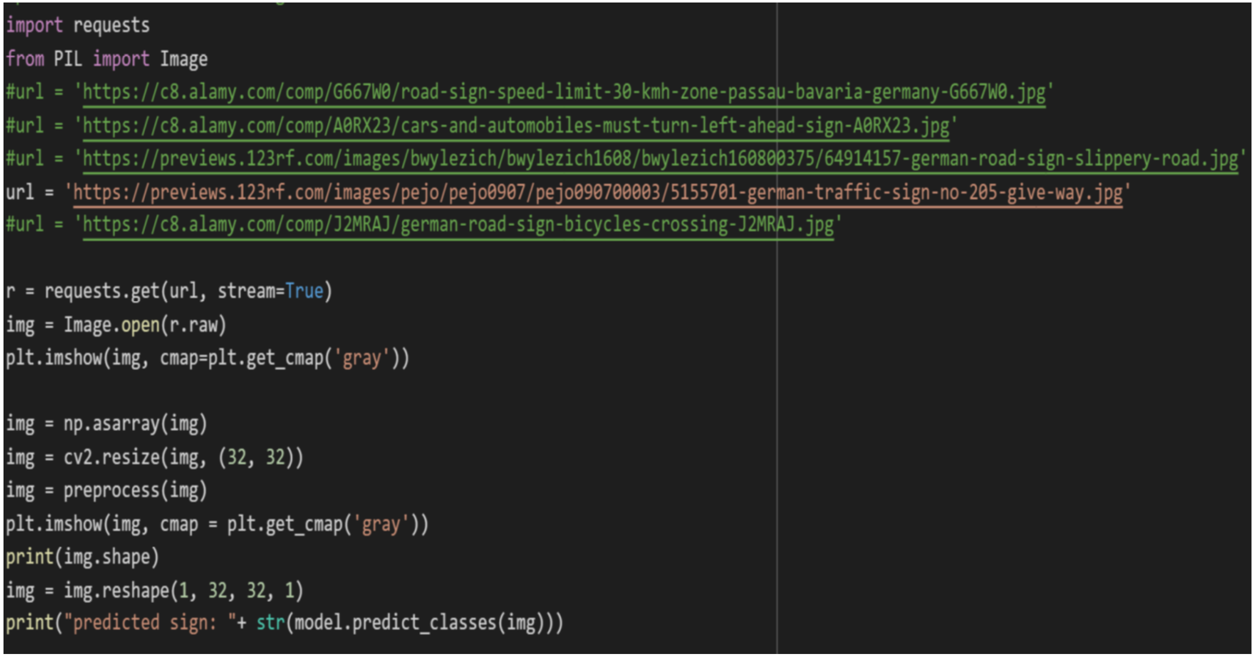


```
Test accuracy: 0.9804434180259705
```

- As we can see, we obtained a training accuracy of 98% which can be considered as good. This shows that our data augmentation technique was effective.

❖ Classifying Traffic Signs from the Internet:

- Let us now put the network to the real test by evaluating the real images from the web.
- We will try a variety of images and see if our network can classify them correctly.
- We did this using the following code snippet.

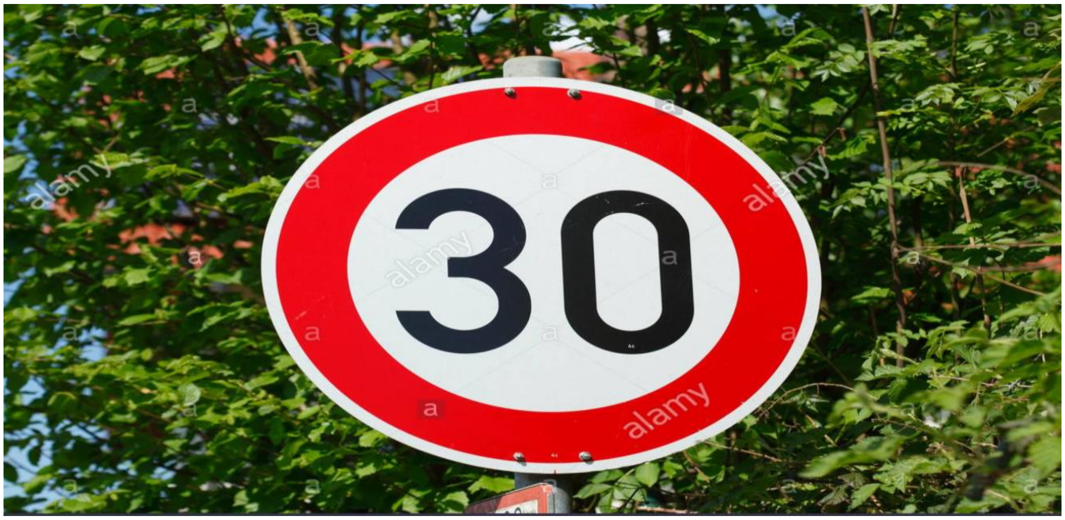


```
import requests
from PIL import Image
#url = 'https://c8.alamy.com/comp/G667W0/road-sign-speed-limit-30-kmh-zone-passau-bavaria-germany-G667W0.jpg'
#url = 'https://c8.alamy.com/comp/A0RX23/cars-and-automobiles-must-turn-left-ahead-sign-A0RX23.jpg'
#url = 'https://previews.123rf.com/images/bwylezich/bwylezich1608/bwylezich160800375/64914157-german-road-sign-slippery-road.jpg'
url = 'https://previews.123rf.com/images/pejo/pejo0907/pejo090700003/5155701-german-traffic-sign-no-205-give-way.jpg'
#url = 'https://c8.alamy.com/comp/J2MRAJ/german-road-sign-bicycles-crossing-J2MRAJ.jpg'

r = requests.get(url, stream=True)
img = Image.open(r.raw)
plt.imshow(img, cmap=plt.get_cmap('gray'))

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(model.predict_classes(img)))
```

- As we can see, we tested out different images from internet.
- Let’s compare the actual signs with the predicted signs to see how our model performs.

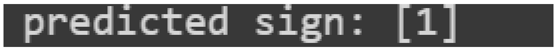
- Note that these signs are randomly picked from the internet and are not from our training dataset.

1. **Speed Limit 30 Sign:**



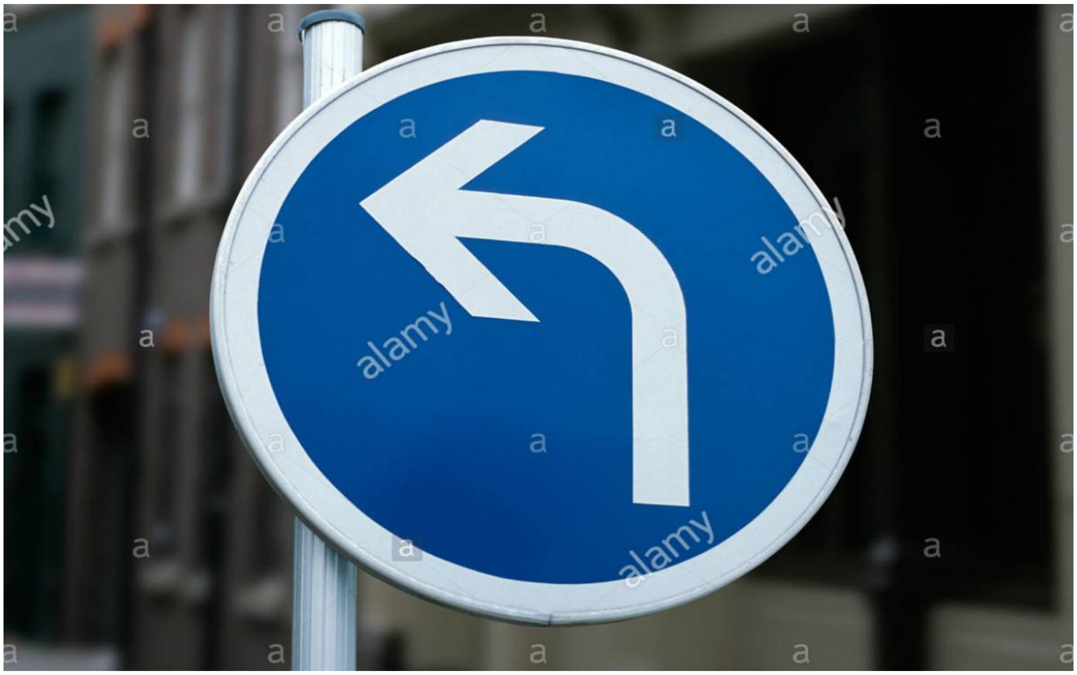
This sign belongs to the 1st class.

Predicted Sign:



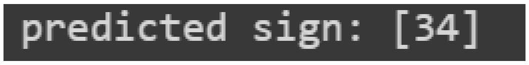
```
predicted sign: [1]
```

2. **Turn Left Sign:**



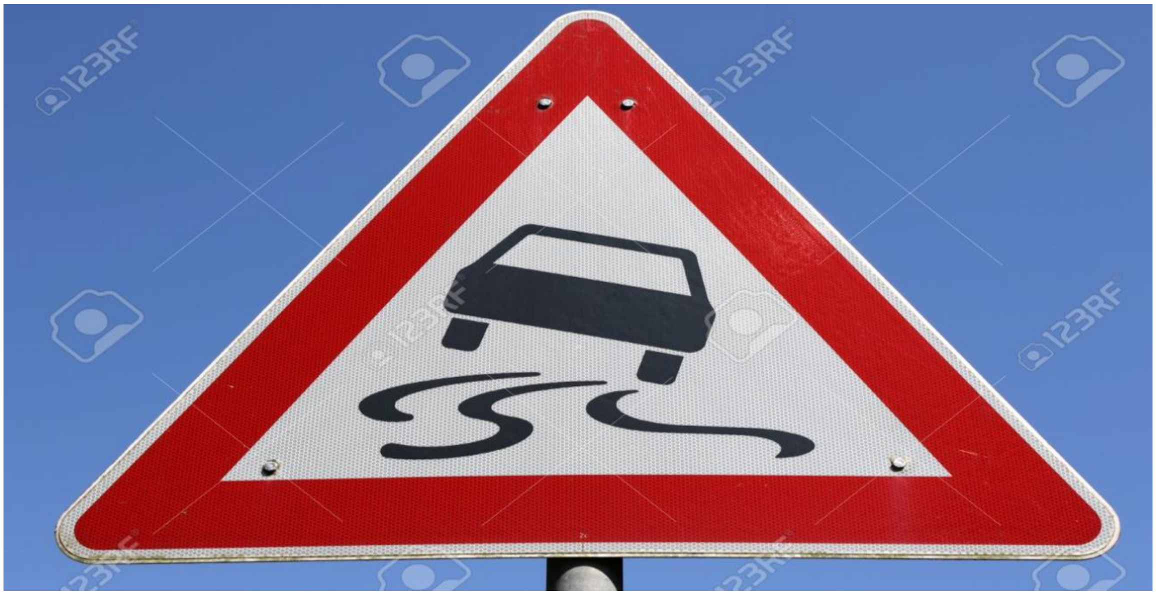
This sign belongs to the 34th class.

Predicted Sign:



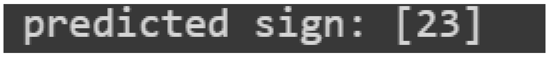
```
predicted sign: [34]
```

3. Slippery Road Sign:



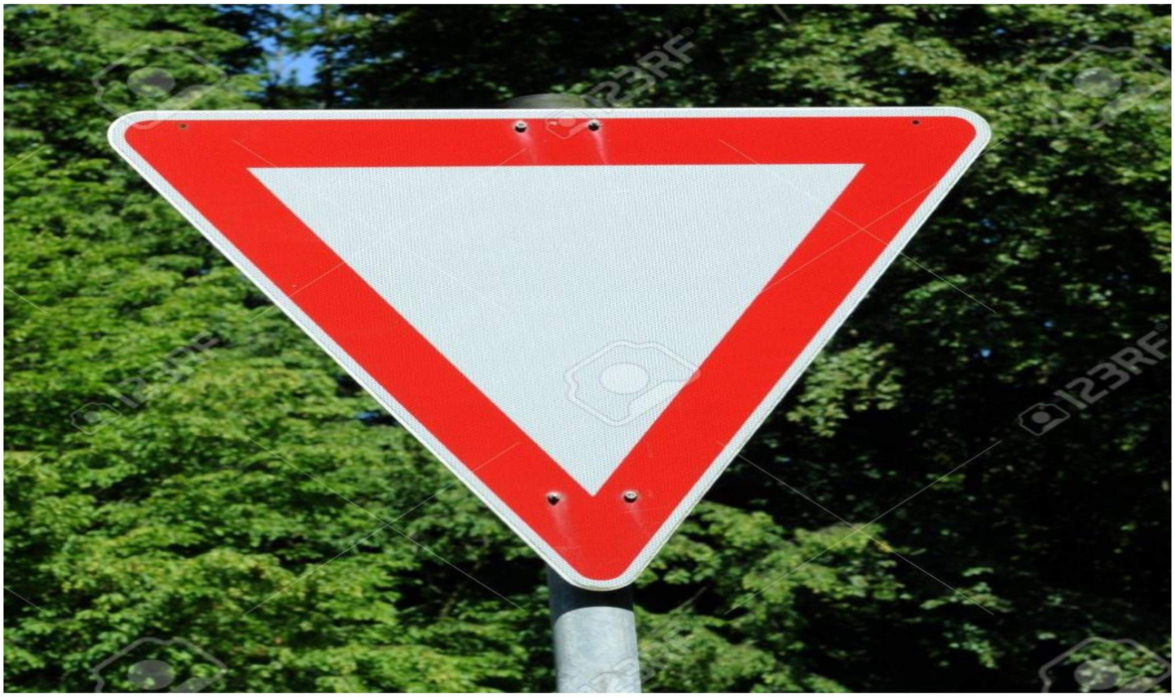
This sign belongs to the **23rd** class.

Predicted Sign:



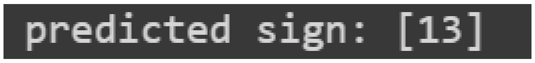
```
predicted sign: [23]
```

4. Give Way Sign:



This sign belongs to the **13th** class.

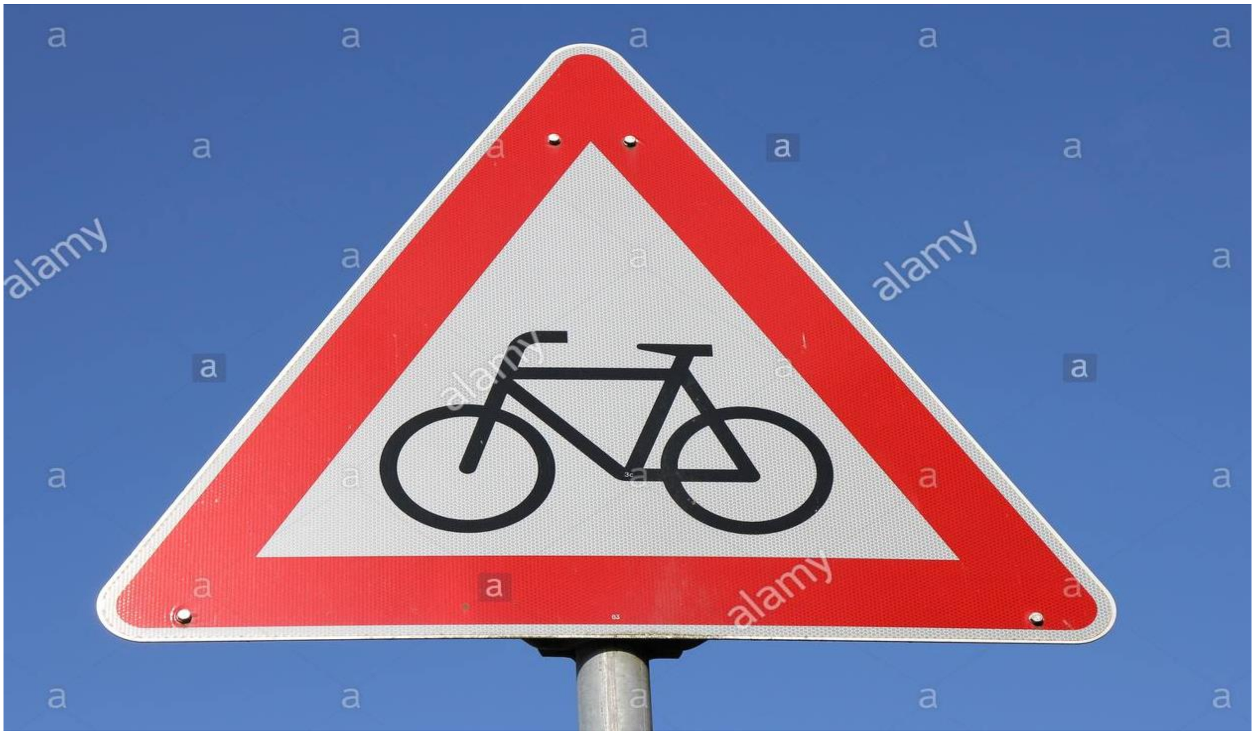
Predicted Sign:



```
predicted sign: [13]
```

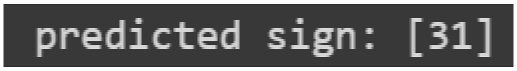
All the above signs are predicted accurately by our model.

5. Bicycles crossing:



This image belongs to class 29.

Predicted Sign:



```
predicted sign: [31]
```

This sign is incorrectly predicted by our model.

❖ Conclusion:

In this project, we successfully built a modified leNet model for classifying traffic signs. We observed that we can obtain better results from the model by configuring its parameters like learning rate, number of convolutional layers, dropout layers and the activation function.

❖ References:

[1] German traffic signs data: https://bitbucket.org/jadslim/german-traffic-signs/src/master/

[2] leNet model: https://engmrk.com/lenet-5-a-classic-cnn-architecture/

[3] Convolutional Neural Networks:
https://towardsdatascience.com/a-comprehensive-guide-to- convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[4] Convolutional Neural Networks: https://cs231n.github.io/convolutional-networks/

[5] Activation Functions: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6