Traffic Sign Recognition For Autonomous Navigation

December 1st, 2019

Team Members: Dhruv Sabharwal, Yash Dixit and Harpreet Virk

Table Of Contents:

Abstract

Motive

This project deals with building a crucial component of driverless cars viz. automatic traffic sign recognition. The model uses the Belgium TS dataset for this purpose.
(http://btsd.ethz.ch/shareddata/)

Introduction

By applying machine learning using convolutional neural networks, we created a model that reliably classifies traffic signs by learning to identify even the most complex features. The dataset is split into training, testing and validation data and has the following characteristics:

1. The image sizes originally are inconsistent but then reshaped to 32 (width) x 32 (height) x 3(RGB color channels).
2. The training dataset contains 62 unique labels with 4575 images.
3. The testing dataset contains 53 unique labels with 2520 images.
4. The testing dataset is split into testing and validation data (1:1 ratio i.e. 1260 images each)

The model can identify traffic signs with around 90 percent accuracy on the testing data set. This report provides a comprehensive outline on the traffic sign detection and classification. The details of the convolutional neural network (CNN) methods that have been used to construct and train the model as well as their specifications are summarised in the following pages with use of visualisation in terms of graphs and a confusion matrix. The shortcomings of the model, such as predicting two almost similar looking labels as one or the other, have also been presented and discussed at length. The conclusion to the models predictions and classification and the effect of various chosen parametres on the final accuracy are provided towards the end of report and methods that helped increase the accuracy have also been highlighted.

Procedure

1. Reading And Shuffling Data

The implementation focuses on first reading the data and putting it into two lists, namely labels and images. The labels list is a list of numbers, where each number represents a unique image label. The images list is a list of arrays, where each array represents a single image. The images and their labels are combined together using the *zip* function in python so as to map each image to its respective image label. The training data is then shuffled. Absence of shuffling could mean our model is biased towards classes which have representation initially in the training data, especially when it is unsure in its predictions.

You can see a sample of the images (one from each label), with labels displayed below. The number in brackets signifies how many images belonging to that label are present in the data:

2. Reshaping And Grayscale

We then move towards converting our images into similar size. This is implemented as all our images had inconsistent sizes initially and working on different sized image data set would be detrimental to the CNN network that we created.

We convert our 3 channel (RGB) images (32 x 32 x 3) into a single channel grayscale image (32 x 32 x1). We implement gray scaling of the images to see whether it can help us in better isolating and reducing the noise in each of our images immensely.

You can see a sample of the images from the grayscaled dataset, with labels displayed below:

3. One-Hot Encoding

We then introduce one hot encoding to prevent the model from producing wrong correlations between label classes. You can see a sample of the one hot encoding here:

```
<class 'numpy.ndarray'>
22
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 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.]
(2520, 62)


<class 'numpy.ndarray'>
19
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 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.]
```

Here an arbitrary training and testing label was provided and the corresponding one hot encoding is displayed.

4. Convolutional Neural Network

We created a sequential model which has a stack of several layers - Our stack has four blocks of layers. The first three blocks contain a convolutional layer, max-pooling layer and a dropout layer.

4.1 Convolution layer

The three convolutional layers have 32, 64 and 128 filters respectively. The filter size signifies the number of features (and their combinations) that the CNN can potentially learn. We increased the filters at each layer such that more relationships between the features could be established by our network. The kernel size was kept fixed at 3x3 in all the layers. Kernel size of 3x3 means that information from all the pixels in the 3x3 vicinity of the current pixel will be used to get the post-convolution value of the current pixel. We did this hoping to extract some useful features since most of the useful features in an image are local. We used odd sized kernels such that the symmetry around the output pixel is maintained.

4.2 Max-Pool and Dropout

The max pooling layer returns the pixel with maximum value from a set of pixels (2x2 in our case) within a kernel. It reduces the spatial size of the feature maps produced by the filters in order to reduce the amount of parameters and computation in the network. We used a dropout of 0.25. This means that we drop 25% of the nodes during training to prevent overfitting of the model on the training dataset.

By the end of the third block our model identifies several useful features which it then uses to classify images into the 62 classes.

The final block in our CNN contains a *flatten* layer followed by some *dense* and *dropout* layers. The flatten layer converts the pooled feature map to a single column which is then passed to the dense layer. The dense layer adds the vector we just obtained to the neural network. We have two hidden layers with 512 nodes each and one output layer with 62 nodes.

4.3 Compiling the model

We then compiled our convolutional neural network using the *compile* function with adagrad optimizer and cross entropy loss.

We created a *checkpoint* to store the best weights of our model during training. For this we linked our collaboratory to google drive and saved the weights whenever there was an increase in the validation accuracy.

4.4 Fitting the Convolutional Neural Network

We used *ImageDataGenerator* for augmentation (described in the next section). This function returns an iterator which we stored in *datagen*. After this we fit our grayscale training images to this iterator.

The final step then was to fit our CNN model to our training data and test its performance on the validation data which we achieved by using *fit_generator*. We provided the training set, the number of epochs (50 in our case), callback list and validation data as input to this function.

When the fit_generator function is executed, the model runs for 50 epochs and calculates training accuracy and validation accuracy for each epoch. The best weight values are stored in the callback list which are later retrieved when we run our model on the testing data.

Problems

Data Augmentation

In the initial training dataset we can see that:

1. Label 22 has 375 images
2. Label 26 has 6 images

There is about a ~6200% difference in the number of images for these labels. This is suggestive of a significant imbalance across classes in the training set. As discussed before, this would result in over-representation of the classes that have a higher number of images in the data set leading to a bias towards these classes. Hence to solve this issue, we implement data augmentation.

The following are the data augmentation procedures that have been implemented, to try to minimise the difference:

1) Random Rotation: between -12^0 and 12^0 angles

2) Featurewise_center: set input mean to 0 over the dataset, feature-wise, to standardize pixel values across the entire dataset.

3) Width_shift_range & Height_shift_range: set to 0.2 (fraction of total width/height). This is used as the traffic signs aren't always in the central frame of the Images. They may be off-center in many ways in the real inputs.

Changing Optimiser from "RMSprop" to "Adagrad" to improve validation accuracy

The initial model ran into a dead end with it's gradient descent algorithm with validation accuracy reaching maximum values of around 78%. The initial optimiser being used was "RMSprop" (Root Mean Squared Propagation). RMSprop takes away the need to adjust learning rate as it chooses a different learning rate for each parameter. However, after running the model for multiple epochs it was noticed that the optimizer reached its saturation point way before it could reach its best validation accuracy score. The main reason for this was identified to be the sparse data that was available for the validation data set (as mentioned above only 53 labels and 2520 images were available).

Hence the next best optimizer option was “Adagrad” which was equipped with dealing with the sparse data availability. “Adagrad” greatly improved the robustness of our model, as it used a different learning rate for every epoch. This also ensured that the optimizer did not reach its saturation point before giving us a high validation score. Hence “Adagrad” was implemented.

Optimizer: RMSprop

```
[11] model_gen.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

After running for 50 epochs:

```
1260/1260 [==============================] - 0s 127us/step
Loss:  0.964694242818015
Accuracy:  0.7412698412698413
```

Optimizer: Adagrad

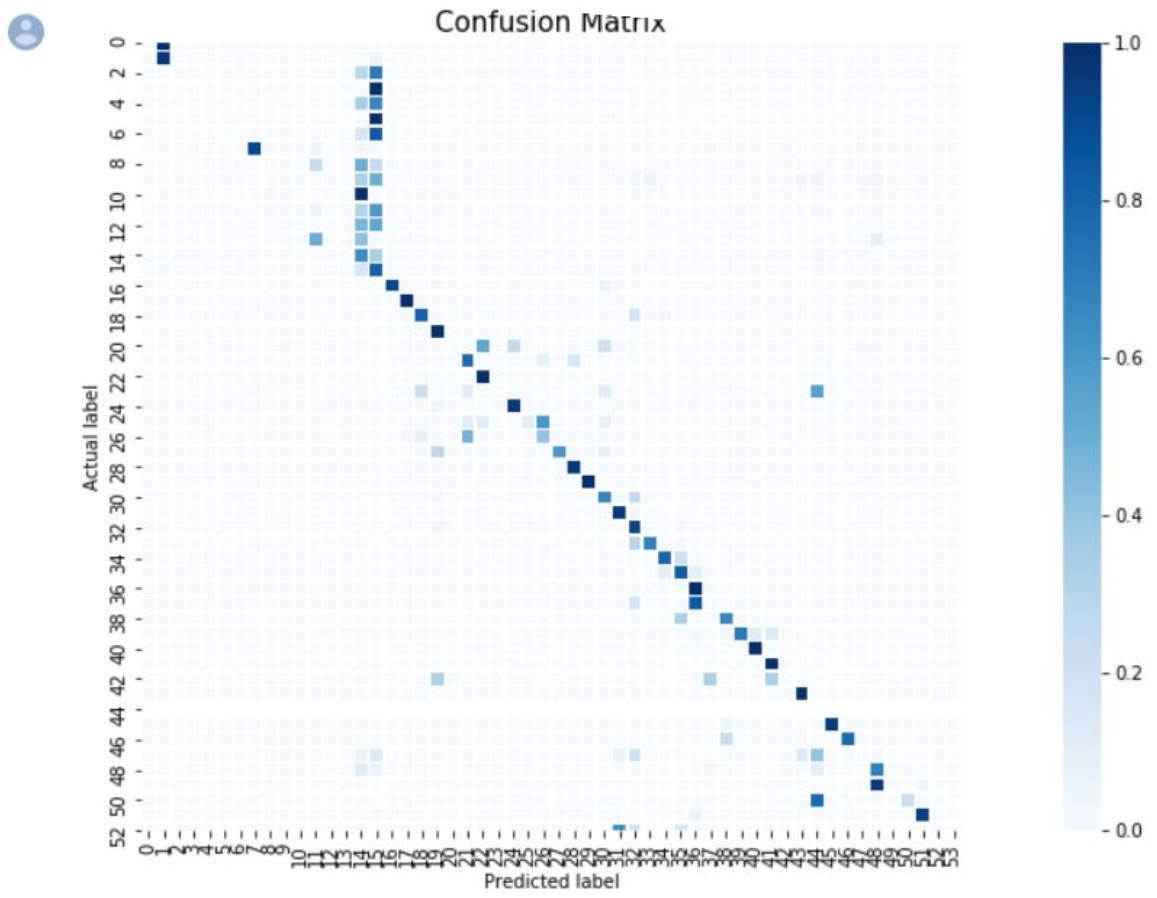
```
[23] model_gen.compile(optimizer='adagrad', loss='categorical_crossentropy', metrics=['accuracy'])
```

After running for 50 epochs:

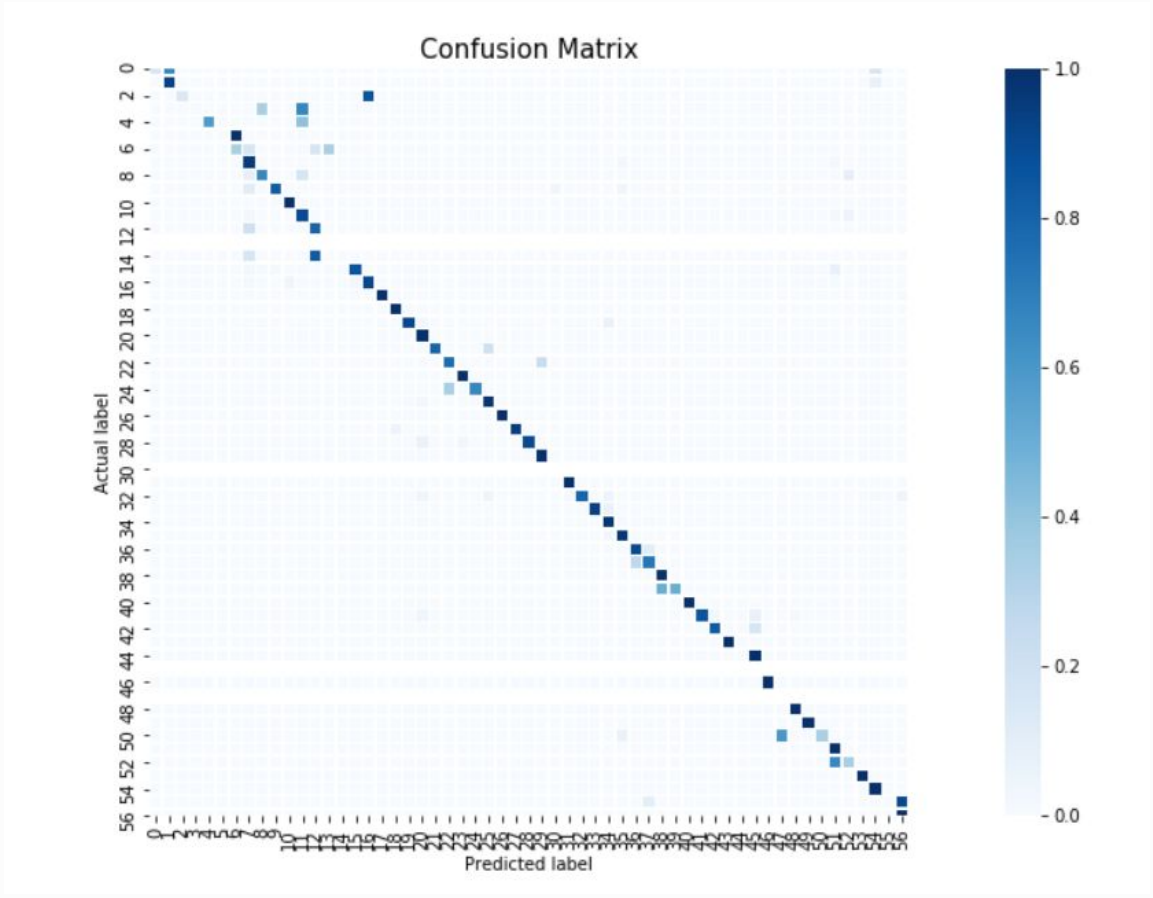
```
1260/1260 [==============================] - 0s 122us/step
0.2877432550939303
0.9158730158730158
```

Confusion In Predicting Similar looking Signs

The training dataset contained similar looking signs for the first few labels (refer to image in Procedure section 1). This similarity was mainly in the form of the red triangle border that was present in all images. The model struggles in the initial predictions amongst these similar images. It fails to distinguish the small difference in details that are present. The same can be seen from the confusion matrix that has been provided below.



To rectify this problem we first identified what labels were causing most of the confusion for our model. From the above confusion matrix it can be inferred that labels 2-15 were being incorrectly predicted. To fix the problem in the region an extra convolutional layer was added. The following was done to extract additional features from the input image so as to make a distinction between the confused images. The change in the optimiser also increased accuracy and the model could make better predictions about the image labels. Data Augmentation also helped in increasing the accuracy in this region. The following is the confusion matrix after the changes were incorporated into the model:



We can see here that the changes in the optimiser and the addition of an extra convolutional layer has significantly improved the model and the confusion matrix now confirms this by predicting the right labels in the first half of the model with a higher level of accuracy.
The presence of white line in the confusion matrix on labels indicate that we have some tags where the particular tag was never guessed. This can be because of lack of images in the testing dataset. When normalizing the matrix the tag is never guessed which yield to a np.nan value. By default all nan values lead to the confusion matrix have white background.

Absence of images in certain labels in testing dataset

There existed labels in the testing dataset namely label 9, 11, 15, 26, 33, 36, 48 and 52 which contained no images at all. This imbalance in data negatively affected the testing accuracy of the model. To counter the lack of these images in the dataset we moved 10% of the images from the training set into the testing set (only for the missing labels). The model was trained again on this modified dataset and improvement in the model testing accuracy was observed.

Results

After training the model for 50 epochs our best model accuracy turned out to be 91% on the validation data. This is indicative of the fact that from the 62 unique labels our model can accurately predicted the label class for 50+ labels. Majority of the incorrectly predicted labels still exist in the first half of the label set , i.e labels 2-15 . The reason for this is still their resemblance in terms of color, design and the same sign pattern.

There is an anomaly at actual label 2 being predicted at label 16. (see the final confusion matrix above) The image of the two labels are provided below for a comparison:

The reason for this can be attributed to the fact that both the signs have a red triangle border encapsulating two curved lines. The model incorrectly predicts these two almost similar looking labels. In the future, a better convolutional network that extracts these features and differentiates between theses images can be implemented and higher accuracies can be reached if larger datasets become available.

The following is the accuracy score at the end of the 50 epochs

```
scores = model_gen.evaluate(grayscale_images_testing, labels_test)
print("Loss: ", scores[0])    #Loss
print("Accuracy: ", scores[1])    #Accuracy
```

```
#Calculating Testing accuracy and Testing loss
scores = model_gen.evaluate(grayscale_images_testing, labels_test)
print(scores[0])    #Testing Loss
print(scores[1])    #Testing Accuracy
```

```
1260/1260 [==============================] - 0s 122us/step
0.2877432550939303
0.9158730158730158
```

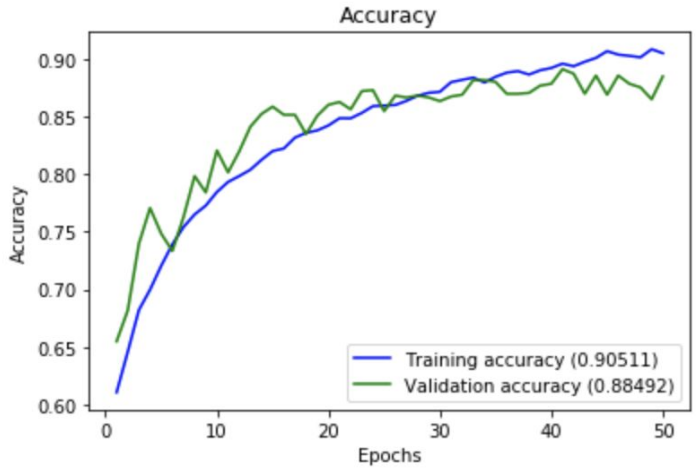
We can also notice that the model successfully predicts the test labels here:

```
#Predicting Traffic Signs
test_predictions = model_gen.predict_classes(grayscale_images_testing, batch_size=batch_size, verbose =1)
print(test_predictions.shape)
print(type(test_predictions))
pred_value=300
print("Predicted Class Label: ",test_predictions[pred_value])
print("Actual Class Label: ",testing_labels[pred_value])
```

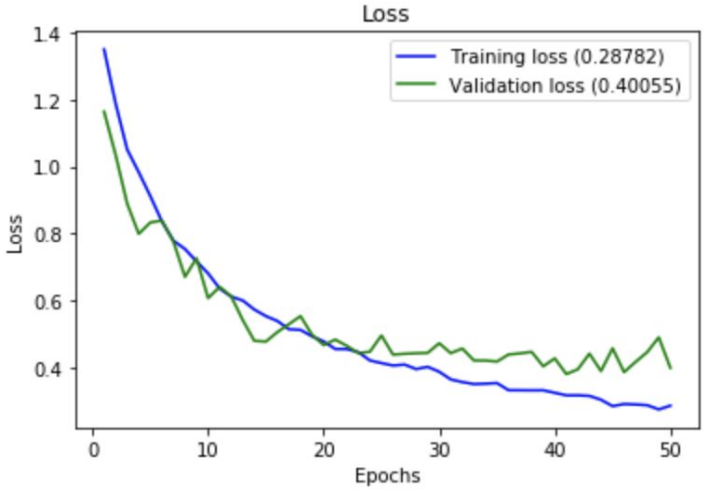
```
1260/1260 [==============================] - 0s 145us/step
(1260,)
<class 'numpy.ndarray'>
Predicted Class Label:  45
Actual Class Label:  45
```

Findings:

Training accuracy vs Testing accuracy graph over 50 iterations of the dataset:



Training loss vs Testing loss graph over 50 iterations of the dataset:



We can observe that we get around 90% accuracy on the training dataset after around the 40th epoch. As this depends on the training data size, and varies with the number of training and testing images, a larger dataset is required to improve our model accuracy and test it further.

Conclusion

The project provides insight into the problem of autonomous traffic sign recognition in the real world. Although we had very few images in our training and testing datasets, our classification model was able to get around 91% accuracy on the validation dataset.

We also observed that our model sometimes predicts almost similar looking labels incorrectly. In order to improve the accuracy further, a larger training and testing dataset containing images from all the different class labels with varying imaging conditions would help greatly. In addition to that, a more robust convolutional neural network that extracts these features and differentiates between theses images can be designed and implemented after more research.

Once this robustness of the training dataset is insured, the model can be used in the operation of driverless cars. We also calculated that our testing time per image is around **13ms**, which can be easily applied to the live feed of a moving vehicle to provide traffic sign predictions on the go.

Dependencies & environment

Coding environment: Colab Jupyter Python3 Notebook

The following coding dependencies have been used:

- Jupyter, NumPy, SciPy, scikit-learn, Keras, TensorFlow, Matplotlib, Pandas, Python3.5

References

1. Confusion Matrix:
https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html

2. Keras Model History:
https://www.kaggle.com/danbrice/keras-plot-history-full-report-and-grid-search