TRAFFIC SIGN RECOGNITION USING DEEP LEARNING ALGORITHM

SUBMITTED BY *Sankari Balasubramaniyan*

OBJECTIVE

The main objective of the project is to study the issue of effective detection and classification of various traffic signs. To comprehend the functionalities of the project, 2 key developments have been performed: a). Accurate localisation of traffic signs using YOLOv3 tiny model, b). further classification of detected traffic sign image among various classes.

SYSTEM REQUIREMENT

- Python 3.6.9
- Tensorflow 2.0.0, keras 2.2.4, h5py 2.10.0
- Opencv4
- Jetpack v43 (in Nvidia GPU)
- Cuda

METHODOLOGY

A two-stage methodology have been implemented to create an end-to-end solution for traffic sign recognition. The first stage involves localisation of one or multiple traffic signs on the image, represented by bounding box and categorised among one of four different classes. At the second stage, the cut fragments of the located traffic signs are fed to a custom convolutional neural network designed for the purpose of this project to classify the given traffic sign among 43 different classes.

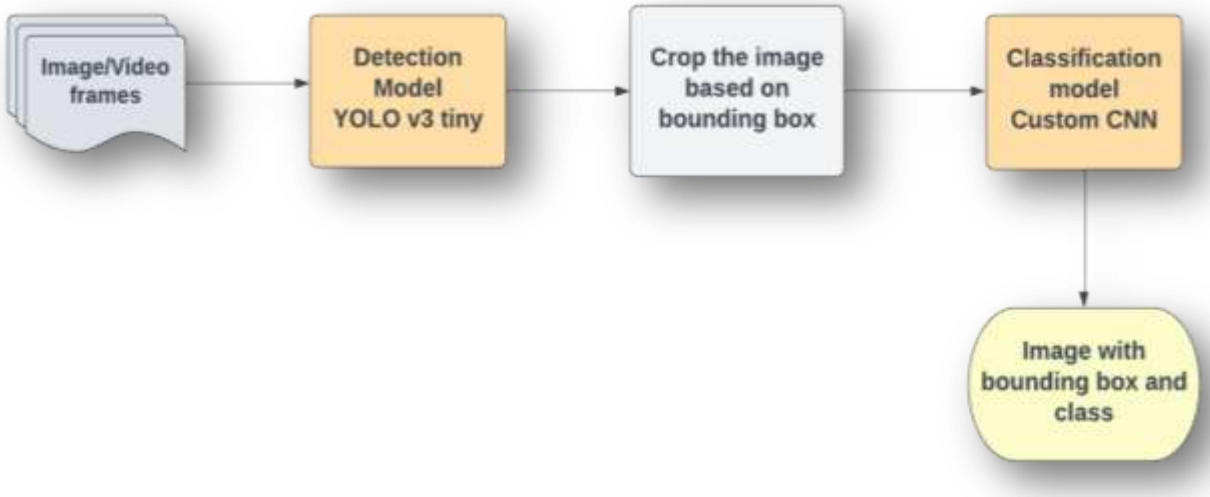

Figure 1. Pipeline of the model

I. DETECTION USING YOLO v3 TINY

a. DATASET PREPARATION

The GTSDB (German Traffic sign detection benchmark) datasets were used to train the localisation stage. Data annotation process had already been done and every single image has a text file with the same name where the object's class numbers and (x,y,w,h) dimensions of the bounding box were recorded. The total dataset was divided in the proportion of 85% and 15%, creating 630 images for training and 111 for validation of the model. The dataset consist of 4 classes - Prohibitory, Danger, Mandatory and others.

Prohibitory	Danger	Mandatory	Other
speed limit; no overtaking; no traffic both ways; no trucks	priority at the next intersection; danger; bend; uneven road; slippery road; road narrows; road crossing; construction; traffic signal; snow; animals	go right; go left; go straight; go right or straight; go left or straight; keep right; keep left; roundabout	restriction ends; priority road; give way; stop; no entry

Figure 2. Different traffic signs separated into four different classes

b. TRAIN YOLOv3 tiny

The Yolov3 tiny architecture have been trained with the GTSDB dataset, to provide the coordinates of the detected traffic signs as the output. YOLO v3 tiny model have Darknet framework as its backbone and it was trained using transfer learning. Parameter used during training is described in the table below. The input size (network size) has the dimension 416x416 pixels, lower dimension was chosen as opposed to the default to level up the model speed. 64 images were processed in one iteration and all batches are divided into 16 blocks, these blocks run in parallel in the gpu (google colab pro was used for training the model) . There was a total of 8000 iterations and weights were saved, and updated after every 1000 iterations. The filter size of the YOLO layer was calculated by the formula "(classes+5)*3 ". Originally, the anchor sizes were calculated using K-means clustering for COCO dataset during the model development, and similar such sizes were used in this project. The spatial dimension of the anchors was used to calculate the predicted bounding box's dimensions.

PARAMETERS	VALUES
BATCH	**64**
SUBDIVISION	**16**
NUMBER OF CLASSES	**4**
OUTPUT LAYER FILTER SIZE	**27**
ITERATIONS	**8000**
NETWORK SIZE	**416x416**
ANCHOR SIZE	**[(10,14),(23,27),(37,58), (81,82),(135,169),(344,319)]**
LEARNING RATE	**0.001**

Table 1. Parameters to train yolov3 tiny

II. CLASSIFICATION USING A CUSTOM NEURAL NETWORK

a. DATASET

The custom neural network was trained with the GTSRB (German Traffic sign recognition benchmark) dataset. The dataset was preprepared with training, validation and test dataset division and resized to 32x32 resolution. There were about 34799 images for training, 4410 for validation and 12630 for testing (overall more than 40,000 images). The dataset was dividing into 43 different classes of traffic signs.

b. DATA AUGUMENTATION

Before feeding the dataset to the CNN, every image is preprocessed. The images were resized to the resolution 32x32 and shuffled to increase the randomness among the dataset, so the model does not recognise any unnecessary pattern while learning. The labels were converted to matrix using one hot encoding technique since they were categorical data. Upon visualising the distribution of the dataset, it was observed that images placement among 43 different classes were unbalanced. This can effectively decrease the performance of the model , as the model tend to predict labels that were most frequently fed to it.

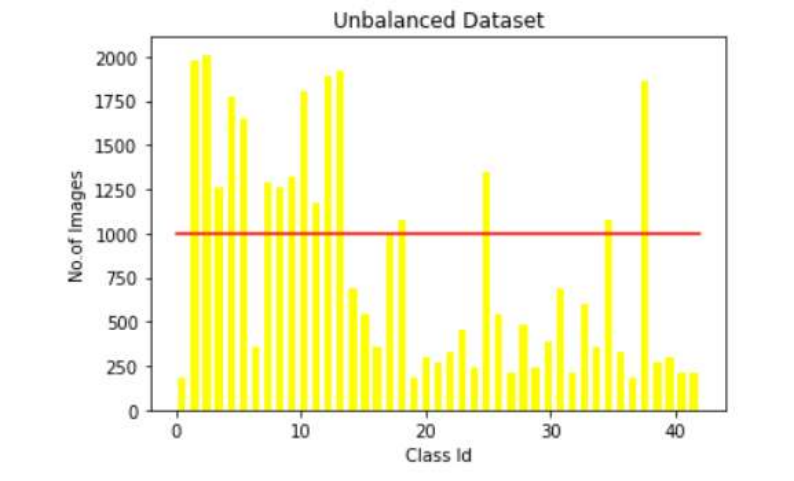

Figure 3. Histogram plot of datasets in different classes

As a result, several data augmentation techniques were applied in the effort to balance this distribution. Classes like “Dangerous curve to the left”, “Turn left ahead”, “Go straight or left” can be flipped once and added to classes like “Dangerous curve to the right”, “Turn right ahead”, “Go straight or right” and vice versa. This will help improve the population in each other’s classes. Classes like “End of all speed” and “passing limits” can be flipped sideways once and down once, and added to the same class. Classes whose datasets that cannot be improved by the above methods can then be improved by random brightness, cropping, zoom or pan. These processes significantly increase the number of datasets and this method is termed as over-sampling. However, in order to not over fit the model, classes that have images more than 1000 are removed randomly before training the model.

c. BUILDING CNN

The architecture of the CNN consist of 9 layers – 5 convolutional layers and 3 fully connected layers. The parameters of the developed model is given in the table below. Initially, the images are fed to the first convolutional layer containing 24 filters of size 5x5 (smaller kernel sizes are chosen to improve processing time), and as the result, 24 feature maps were calculated in accordance with the number of convolutional layer filters. The ReLU activation function is applied to the obtained feature maps, which removes negative values in the feature maps by replacing them with 0. Following this, 4 different convolutional layers are built with ReLu activation function with increasing filter numbers and decreasing kernel sizes, which eventually increases the number of weights to learn by CNN. Finally 3 dense layers are built to classify the detected objects. The output layer consist of 43 neurons in accordance with the number of classes in the dataset. The training process was divided into 32 batches of images per iteration and the training saturated after 12 epochs.

PARAMETERS	DESCRIPTOR
BATCH	**32**
EPOCH	**10**
NUMBER OF CLASSES	**43**
OPTIMIZER	**ADAM**
ITERATIONS	**1318**
NETWORK SIZE	**32X32X3**
LOSS FUNCTION	**CATEGORICAL CROSS ENTROPY**
LEARNING RATE	**0.0001**
STRIDE	**1**

Table 2. Parameters to build and train CNN

RESULT AND DISCUSSION

The yolo v3 tiny model uses mAP(mean accuracy precision) metric to evaluate accuracy of the model over 1000 iterations during training by evaluating against validation dataset. During training, the highest found mAP was at 7000 iteration point and it was calculated to be 78%. The accuracy has been compromised since yolo v3 tiny was chosen for its speed and light weight. A better accuracy can be obtained if yolo v3 or v4 is replaced by the existing model.

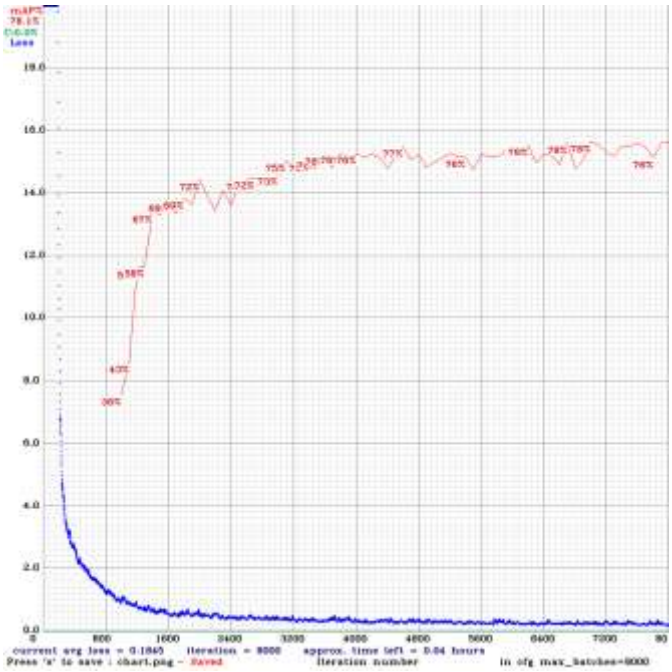

Figure 4. Graphical representation of Loss vs mAP

The CNN was trained in 12 epochs and obtained the highest training accuracy calculated to be 99.25% and when tested against validation set, it reached 97.01%. The model was evaluated against the test dataset (completely new images that the model has never seen before) and it achieved an accuracy of 94.74%.

Upon running the model with a pre-recorded video in Nvidia Jetson nano, it was observed that the detection model takes around 0.025 seconds and classification model takes about 0.015 seconds to process each frame. The FPS for detection model was found to be around 38 F/sec and the inference time of classification was 16 F/sec for the video. On an average it took 9 seconds to process the entire frames of the video and each video frame took a total processing time of 0.16 seconds.

The future perspective of the project would be to optimise the model such that it could be applied in real time application since traffic sign detection is one of the important features of modern autonomous car. For the moment the program is written in python, and implementing the classifier in cpp could moderately increase the FPS. As in this case, a separate thread is implemented to store the frame in a queue which enables us to get the future frames even before the classifier predicts the first frame, so one better way of improving the solution would be to use multiprocessing technique to enable the detector to find the bounding boxes of future frames while classifier is trying to classify the present frame, this can significantly improve the processing time of the model.