Internship Report

on

Object Identification and Classification for Autonomous Driving Vehicle

(Multi-column Deep Neural Networks for Image Classification and Model checking)

Under

Dr. Srinivas Pinisetty
Assistant Professor
School of Electrical Sciences
Indian Institute of Technology Bhubaneswar

By

Satwik Chatragadda (17CS01045)
4th Year Undergrad, Computer Science and Engineering
Indian Institute of Technology Bhubaneswar
E-mail: cs15@iitbbs.ac.in

TABLE OF CONTENTS

ABSTRACT

The future of automobile industry is vastly tending towards Autonomous driving vehicles. These driverless vehicles have a lot of problems which need to be tackled before entering into the market. Traffic sign detection and classification is of paramount importance for the future of autonomous vehicle technology. Many promising methods have been proposed in the literature to perform this task. However, almost all of them work well only with clear and noise-free images, and, do not provide satisfactory results in the presence of challenging image-capture conditions (such as noise, variation of illumination, motion artifacts, etc.)

In this paper, we propose a Convolutional Neural Network (CNN)-based approach for detecting and classifying traffic signs which is robust against such challenging environmental conditions. In the proposed model, our approach is to selectively preprocess the image to enhance the sign features and classify the sign proposals present in the image. These tasks are carried out by implementing CNNs of varying architecture and depth.

INTRODUCTION

In recent years, Deep learning has flourished in many fields such as autonomous driving, games and engineering applications. As one of the most popular topics, autonomous driving has drawn great attention both from the academic and industrial communities and is thought to be the next revolution in the intelligent transportation system. One of the most important factors to be considered for any transportation vehicle is to follow the rules of the road. Thus, identification and classification of various traffic signs is of prime importance for Autonomous vehicles.

In order to address these problems, I present a novel method of detecting traffic-signs completely based on deep Convolutional Neural Networks (CNNs). The motivation for my model is twofold. First of all, due to the variety of the types and level of challenges present in the dataset, any algorithmic or statistical approach to de-noising, segmentation and classification proves to be quite computationally intensive. As a result, this approach would be unsuitable for real-time applications. Thus, one of the reasons to consider a neural-network based detection and classification scheme is its computational efficiency during implementation. Secondly, CNNs have achieved phenomenal accuracy and efficiency in image-classification in recent years. An analysis of the dataset reveals that, there are only a finite number of challenge-types and sign-types to be considered, and, the type of noise in each image is distinct and unique. Therefore, a neural-network based classification of the noise is likely to yield good results. The following sections of this paper explain the outline of the model, detailed architecture of the networks employed in each stage, implementation and training procedure of the networks, and, the results obtained after testing the model against the testing-data provided with the dataset.

For the classification of Traffic signs the data set German Traffic Sign Recognition Benchmark is used. This data set contains images of various traffic signs which are divided into 43 classes. The images are preprocessed and augmented to form new images changing the angles of rotation, brightness and various other factors such that a larger data set is obtained. The images are then converted into arrays containing the pixel values of size 32*32*3 and then are converted into grayscale images. These are divided into training, test and validation sets of sufficient sizes. These are then given into the CNN model which has the LeNet architecture which outputs a softmax which has 43 classes. Thus the model gets trained and can output the class to which a particular sign image belongs.

CHAPTER 1: DATA SET

1.1 DATA SET DESCRIPTION

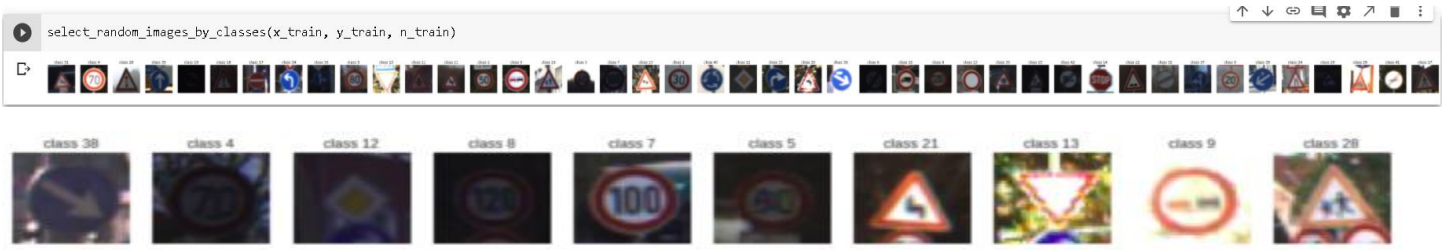
Data set : `'https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/traffic-signs-data.zip'`

Data Classes: `'https://raw.githubusercontent.com/navoshta/traffic-signs/master/signnames.csv'`

The data set contains 51,839 images which are classified into 43 classes which are shown in the data classes. These images are divided into the following sets:

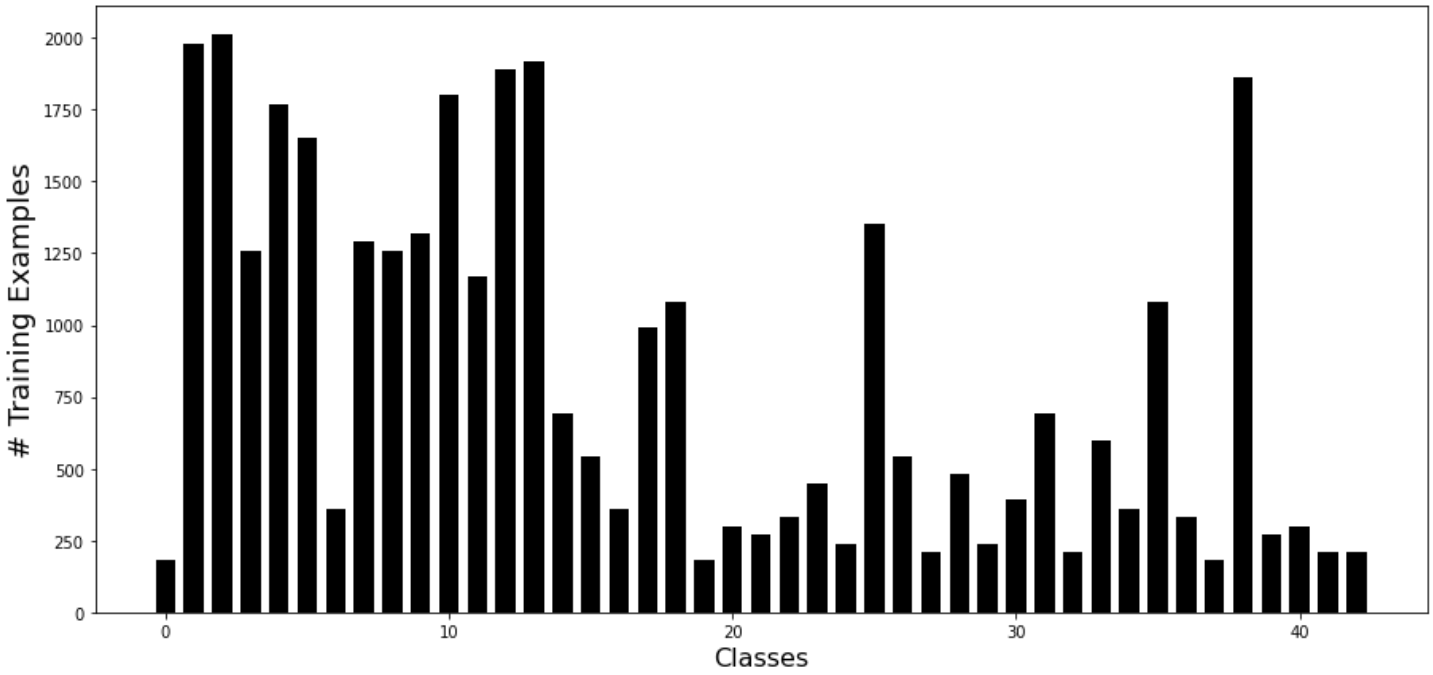
- `Number of training examples = 34799`
- `Number of testing examples = 12630`
- `Number of validation examples = 4410`
- `Image data shape = (32, 32, 3)`
- `Number of classes = 43`

Few example images are:



The images that are present in the dataset are vastly irregular where in a few classes have very high number of examples above 2000 while some are as low as 200. This scale of irregularity effects the training of the model as a few classes do not have sufficient number of training examples to gather weights which would recognize the classes. This occurs as we do not encounter as many speed limit signs as we encounter the stop sign. If the model is trained with this uneven number of examples in the various classes, then the model tends to be biased towards the class which has higher number of examples.

The following is a bar graph which shows the number of training examples present in each class:



The above graph shows that most of the classes do not even have 500 training examples and only a few classes have number above 1750 which would result in biasing towards the classes above 1750 and would have a bad accuracy for images in classes which have low number of training examples. Gathering more data is a tedious task as we would not find large number of examples for few data sets such as “Beware of ice/snow” or “Wild animals ahead”. Thus one way of tackling this process is Data Augmentation.

1.2 Data Augmentation

Data augmentation is a strategy that enables practitioners to significantly increase the diversity of data available for training models, without actually collecting new data. Data augmentation techniques such as cropping, padding, and horizontal flipping are commonly used to train large neural networks. However, most approaches used in training neural networks only use basic types of augmentation. While neural network architectures have been investigated in depth, less focus has been put into discovering strong types of data augmentation and data augmentation policies that capture data invariances.

In many deep learning applications, we often come across data sets where one type of data may be seen more than other types. In a traffic sign identification task, there are more stop

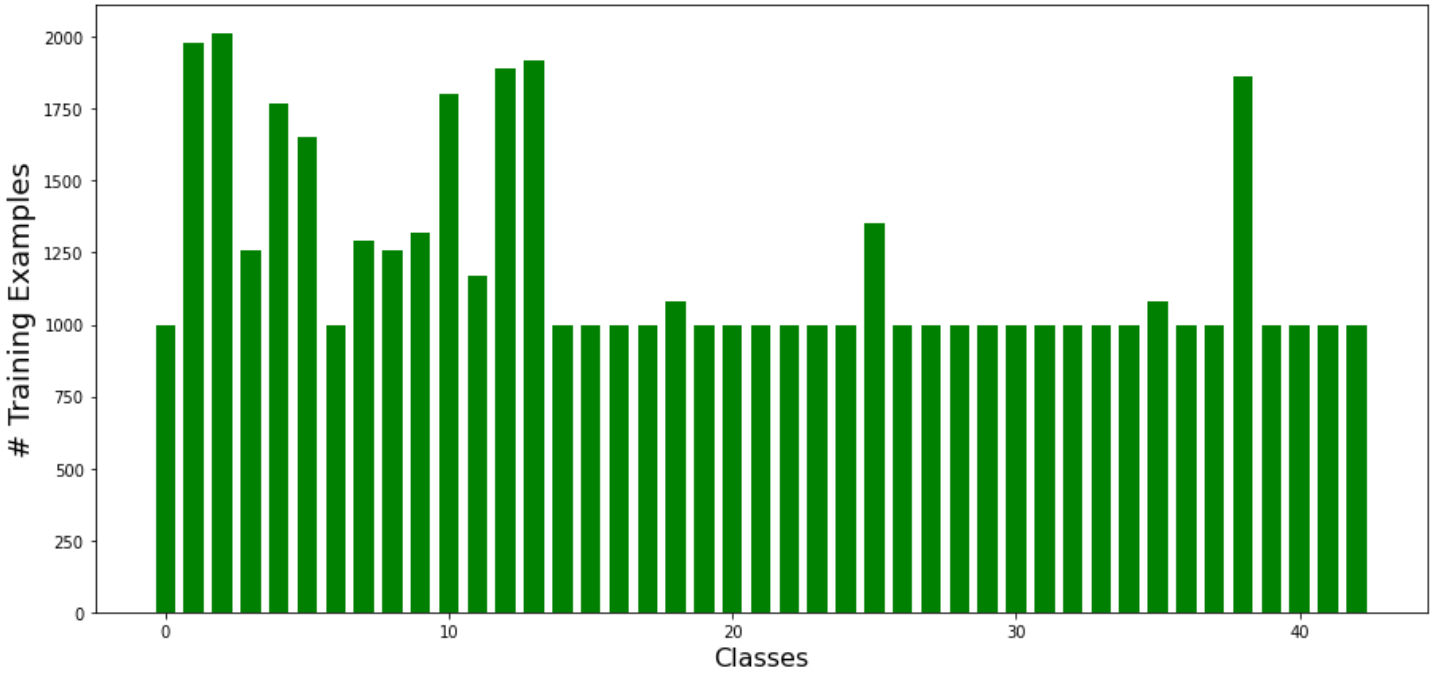
signs than speed limit signs. Therefore, in these cases, we need to make sure that the trained model is not biased towards the class that has more data. As an example, consider a data set where there are 5 speed limit signs and 20 stop signs. If the model predicts all signs to be stop signs, its accuracy is 80%. Further, f1-score of such a model is 0.88. Therefore, the model has high tendency to be biased toward the 'stop' sign class. In such cases, additional data can be generated to make the size of data sets similar.

One way to collect more data is to take the picture of the same sign from different angles. This can be done easily in openCV by applying affine transformations, such as rotations, translations and shearing. Affine transformations are transformations where the parallel lines before transformation remain parallel after transformation.

The `transform_image` function takes in the original image, range of angle rotation, range of translation and range of shearing and returns a jittered image. As the function chooses true transformation values from a uniform distribution that is specified by these ranges. The `augment_brightness_camera_images` function is also added which can be chosen to be on or off by setting the brightness flag in `transform_image()` function.

The same techniques can be applied using image generator in *Keras*. However, the `transform_image()` function used here will help you modify the parameters and see what's happening under the hood.

The following is the graph which is obtained after Data Augmentation:



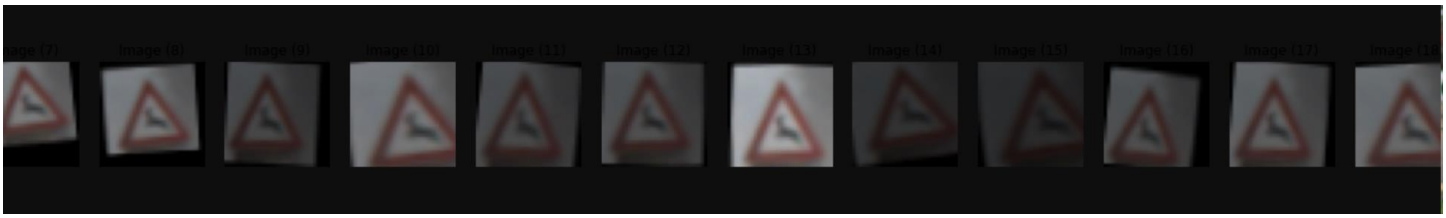
This shows that the function increased the training examples for classes having low number of examples to a minimum of 1000 examples such that the problem of biasing can be avoided.

An image is taken from a class and the values of rotation angle is changed to a new angle and a new image is stored in that class. Similarly, the range of shearing and the range of translation is modified for the images and new images get added to the classes having lower number of examples such that the number of training examples reaches a minimum value which in this case is 1000.

```
[ ] images = []

    for i in range(0, 100):
      images.append(transform_image(x_train[555],10,5,5,brightness=1))

    show_images(images)
```



The above images are the examples of the new images which gets added to the “Wild Animal Ahead” class after Data Augmentation.

After Data Augmentation the number of examples in the training set is 51690.

```
n_train = x_train.shape[0]

print("Number of training examples =", n_train)

Number of training examples = 51690
```

1.3 Data Preprocessing and Gray Scaling

Image pre-processing is the term for operations on images at the lowest level of abstraction. These operations do not increase image information content but they decrease it if entropy is an information measure. The aim of pre-processing is an improvement of the image data that suppresses undesired distortions or enhances some image features relevant for further processing and analysis task. Image preprocessing use the redundancy in images. Neighboring pixels corresponding to one real object have the same or similar brightness value. If a distorted pixel can be picked out from the image, it can be restored as an average value of neighboring pixels

Various image preprocessing include image cropping and filtering, intensity adjustment and histogram equalization, brightness thresholding, clearing areas of a binary image, detecting edges and illustration.

After preprocessing of data we change the images to a grayscale images which reshapes the x_training from (51690,32,32,3) to (51690,32,32,1). Changing the images to a grayscale images could slightly improve the Neural Network performance.

CHAPTER 2: STRUCTURE

2.1 CONVOLUTED NEURAL NETWORK

The name “convolutional neural network” indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics. They have applications in image and video recognition, recommender systems, image classification, medical image analysis, natural language processing, and financial time series.

CNNs are regularized versions of multilayer perceptrons. Multilayer perceptrons usually mean fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks makes them prone to overfitting data. Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.

Convolutional networks were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.

2.2 ARCHITECTURE

A convolutional neural network consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of a series of convolutional layers that *convolve* with a multiplication or other dot product. The activation function is commonly a RELU layer, and is subsequently followed by additional convolutions such as pooling layers, fully connected layers and normalization layers, referred to as hidden layers because their inputs and outputs are masked by the activation function and final convolution.

Though the layers are colloquially referred to as convolutions, this is only by convention. Mathematically, it is technically a *sliding dot product* or cross-correlation. This has significance for the indices in the matrix, in that it affects how weight is determined at a specific index point.

2.2.1 Convolutional

When programming a CNN, the input is a tensor with shape (number of images) x (image height) x (image width) x (image depth). Then after passing through a convolutional layer, the image becomes abstracted to a feature map, with shape (number of images) x (feature map height) x (feature map width) x (feature map channels). A convolutional layer within a neural network should have the following attributes:

- Convolutional kernels defined by a width and height (hyper-parameters).
- The number of input channels and output channels (hyper-parameter).
- The depth of the Convolution filter (the input channels) must be equal to the number channels (depth) of the input feature map.

Convolutional layers convolve the input and pass its result to the next layer. This is similar to the response of a neuron in the visual cortex to a specific stimulus. Each convolutional neuron processes data only for its receptive field. Although fully connected feedforward neural networks can be used to learn features as well as classify data, it is not practical to apply this architecture to images. A very high number of neurons would be necessary, even in a shallow (opposite of deep) architecture, due to the very large input sizes associated with images, where each pixel is a relevant variable. For instance, a fully connected layer for a (small) image of size 100 x 100 has 10,000 weights for *each* neuron in the second layer. The convolution operation brings a solution to this problem as it reduces the number of free parameters, allowing the network to be deeper with fewer parameters. For instance, regardless of image size, tiling regions of size 5 x 5, each with the same shared weights, requires only 25 learnable parameters. By using regularized weights over fewer parameters, the vanishing gradient and exploding gradient problems seen during backpropagation in traditional neural networks are avoided.

2.2.2 Pooling

Convolutional networks may include local or global pooling layers to streamline the underlying computation. Pooling layers reduce the dimensions of the data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. Local pooling combines small clusters, typically 2 x 2. Global pooling acts on all the neurons of the convolutional layer. In addition, pooling may compute a max or an average. *Max pooling* uses the maximum value from each of a cluster of neurons at the prior layer. *Average pooling* uses the average value from each of a cluster of neurons at the prior layer.

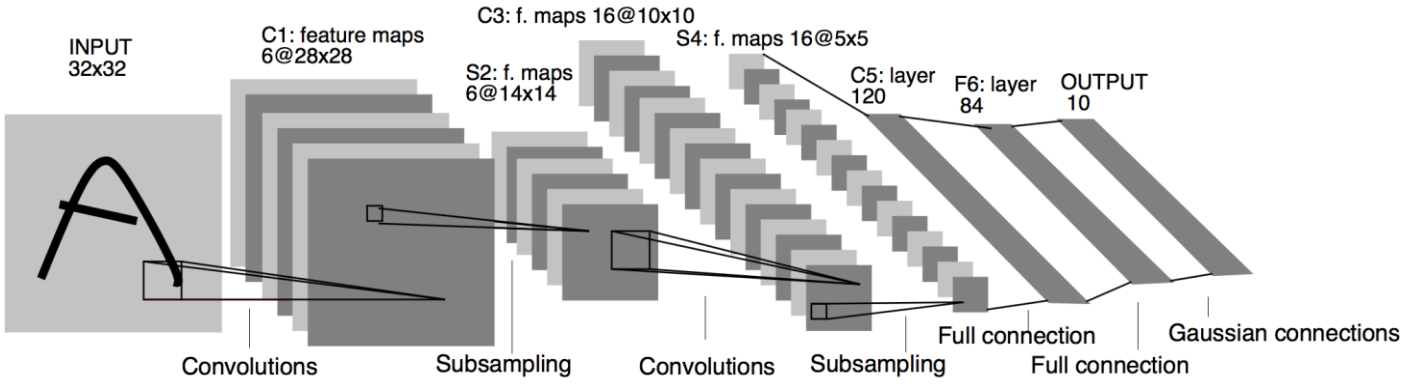
2.2.3 Fully connected

Fully connected layers connect every neuron in one layer to every neuron in another layer. It is in principle the same as the traditional multi-layer perceptron neural network (MLP). The flattened matrix goes through a fully connected layer to classify the images.

2.3 LeNet

LeNet is a convolutional neural network structure proposed by Yann LeCun et al. in 1998. In general, LeNet refers to lenet-5 and is a simple convolutional neural network. Convolutional neural networks are a kind of feed-forward neural network whose artificial neurons can respond to a part of the surrounding cells in the coverage range and perform well in large-scale image processing.

2.3.1 Structure



As a representative of the early convolutional neural network, LeNet possesses the basic units of convolutional neural network, such as convolutional layer, pooling layer and full connection layer, laying a foundation for the future development of convolutional neural network. As shown in the figure (input image data with 32*32 pixels): lenet-5 consists of seven layers. In addition to input, every other layer can train parameters. In the figure, Cx represents convolution layer, Sx represents sub-sampling layer, Fx represents complete connection layer, and x represents layer index.

Layer C1 is a convolution layer with six convolution kernels of 5x5 and the size of feature mapping is 28x28, which can prevent the information of the input image from falling out of the boundary of convolution kernel.

Layer S2 is the subsampling/pooling layer that outputs 6 feature graphs of size 14x14. Each cell in each feature map is connected to 2x2 neighborhoods in the corresponding feature map in C1.

Layer C3 is a convolution layer with 16 5-5 convolution kernels. The input of the first six C3 feature maps is each continuous subset of the three feature maps in S2, the input of the next six feature maps comes from the input of the four continuous subsets, and the input of the next three feature maps comes from the four discontinuous subsets. Finally, the input for the last feature graph comes from all feature graphs of S2.

Layer S4 is similar to S2, with size of 2x2 and output of 16 5x5 feature graphs.

Layer C5 is a convolution layer with 120 convolution kernels of size 5x5. Each cell is connected to the 5*5 neighborhood on all 16 feature graphs of S4. Here, since the feature graph size of S4 is also 5x5, the output size of C5 is 1*1. So S4 and C5 are completely connected. C5 is labeled as a convolutional layer instead of a fully connected layer, because if lenet-5 input becomes larger and its structure remains unchanged, its output size will be greater than 1x1, i.e. not a fully connected layer.

F6 layer is fully connected to C5, and 84 feature graphs are output.

The final layer is a softmax layer consisting of 43 nodes which represent the traffic sign classes.

```
def le_net():
  model=Sequential()
  model.add(Conv2D(30,(5,5),input_shape = (32,32,1),activation='relu'))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(Conv2D(15,(3,3),activation='relu'))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(Flatten())
  model.add(Dense(500,activation='relu'))
  model.add(Dropout(0.2))
  model.add(Dense(43,activation='softmax'))
  model.compile(Adam(lr=0.001),loss='categorical_crossentropy',metrics=['accuracy']) # lr = learning rate
  return model
```

The following is the summary of all the layers that are present in the model:

```
lenet.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 30)        780
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 30)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 12, 12, 15)        4065
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 15)          0
_________________________________________________________________
flatten (Flatten)            (None, 540)               0
_________________________________________________________________
dense (Dense)                (None, 500)               270500
_________________________________________________________________
dropout (Dropout)            (None, 500)               0
_________________________________________________________________
dense_1 (Dense)              (None, 43)                21543
=================================================================
Total params: 296,888
Trainable params: 296,888
Non-trainable params: 0
_________________________________________________________________
```

2.3.2 Features

- Every convolutional layer includes three parts: convolution, pooling, and nonlinear activation functions
- Using convolution to extract spatial features (Convolution was called receptive fields originally)
- Subsampling average pooling layer
- tanh activation function
- Using MLP as the last classifier
- Sparse connection between layers to reduce the complexity of computational

2.4 PARAMETERS AND HYPER PARAMETERS

Number of training examples: 51690

Activation Function: ReLu

Learning Rate: 0.001

Epochs: 100

Batch Size: 128

CHAPTER 3: RESULTS

After the model is run the result obtained is as follows:

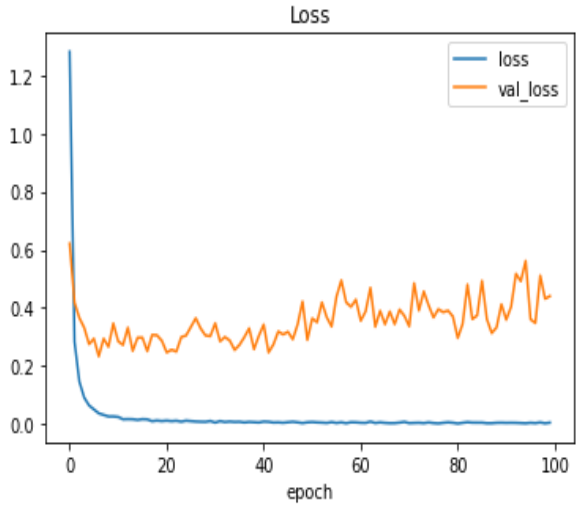
```
history=lenet.fit(x_train,y_train,epochs=100,batch_size=128,verbose=1,shuffle=1,validation_data=(x_valid,y_valid))
lenet.save("/content/Traffic")
```

```
Epoch 1/100
404/404 [==============================] - 46s 115ms/step - loss: 1.2814 - accuracy: 0.6543 - val_loss: 0.6213 - val_accuracy: 0.8184
Epoch 2/100
404/404 [==============================] - 46s 114ms/step - loss: 0.2840 - accuracy: 0.9167 - val_loss: 0.4190 - val_accuracy: 0.8721
Epoch 3/100
404/404 [==============================] - 48s 118ms/step - loss: 0.1465 - accuracy: 0.9567 - val_loss: 0.3655 - val_accuracy: 0.8839
Epoch 4/100
404/404 [==============================] - 47s 116ms/step - loss: 0.0913 - accuracy: 0.9732 - val_loss: 0.3319 - val_accuracy: 0.8884
Epoch 5/100
404/404 [==============================] - 47s 115ms/step - loss: 0.0646 - accuracy: 0.9809 - val_loss: 0.2752 - val_accuracy: 0.9145
Epoch 6/100
404/404 [==============================] - 46s 114ms/step - loss: 0.0507 - accuracy: 0.9843 - val_loss: 0.2941 - val_accuracy: 0.9136
Epoch 7/100
404/404 [==============================] - 50s 123ms/step - loss: 0.0374 - accuracy: 0.9887 - val_loss: 0.2334 - val_accuracy: 0.9331
Epoch 8/100
404/404 [==============================] - 47s 116ms/step - loss: 0.0318 - accuracy: 0.9901 - val_loss: 0.2941 - val_accuracy: 0.9213
Epoch 9/100
404/404 [==============================] - 47s 115ms/step - loss: 0.0266 - accuracy: 0.9915 - val_loss: 0.2663 - val_accuracy: 0.9268
Epoch 10/100
404/404 [==============================] - 46s 114ms/step - loss: 0.0266 - accuracy: 0.9909 - val_loss: 0.3466 - val_accuracy: 0.9150
```

```
Epoch 90/100
404/404 [==============================] - 45s 112ms/step - loss: 0.0051 - accuracy: 0.9983 - val_loss: 0.4124 - val_accuracy: 0.9469
Epoch 91/100
404/404 [==============================] - 46s 113ms/step - loss: 0.0045 - accuracy: 0.9989 - val_loss: 0.3594 - val_accuracy: 0.9490
Epoch 92/100
404/404 [==============================] - 45s 112ms/step - loss: 0.0049 - accuracy: 0.9985 - val_loss: 0.4041 - val_accuracy: 0.9474
Epoch 93/100
404/404 [==============================] - 45s 110ms/step - loss: 0.0046 - accuracy: 0.9987 - val_loss: 0.5173 - val_accuracy: 0.9304
Epoch 94/100
404/404 [==============================] - 45s 111ms/step - loss: 0.0038 - accuracy: 0.9989 - val_loss: 0.4917 - val_accuracy: 0.9395
Epoch 95/100
404/404 [==============================] - 45s 110ms/step - loss: 0.0030 - accuracy: 0.9991 - val_loss: 0.5617 - val_accuracy: 0.9376
Epoch 96/100
404/404 [==============================] - 45s 111ms/step - loss: 0.0048 - accuracy: 0.9986 - val_loss: 0.3607 - val_accuracy: 0.9503
Epoch 97/100
404/404 [==============================] - 45s 112ms/step - loss: 0.0036 - accuracy: 0.9988 - val_loss: 0.3480 - val_accuracy: 0.9537
Epoch 98/100
404/404 [==============================] - 46s 114ms/step - loss: 0.0061 - accuracy: 0.9985 - val_loss: 0.5114 - val_accuracy: 0.9372
Epoch 99/100
404/404 [==============================] - 46s 114ms/step - loss: 0.0030 - accuracy: 0.9991 - val_loss: 0.4314 - val_accuracy: 0.9476
Epoch 100/100
404/404 [==============================] - 45s 111ms/step - loss: 0.0053 - accuracy: 0.9985 - val_loss: 0.4404 - val_accuracy: 0.9522
```

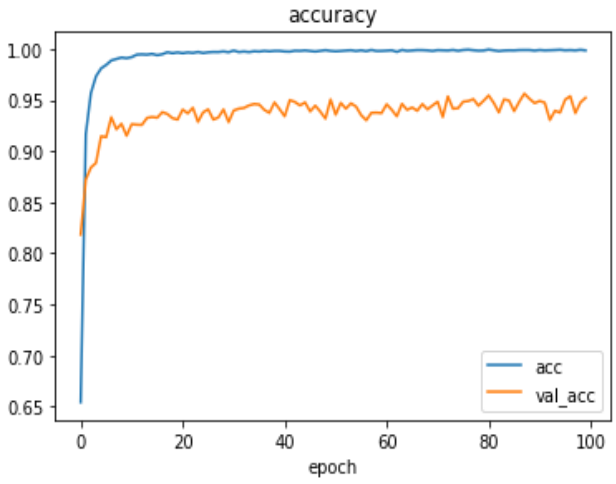
The graph of validation loss and loss plotted for various epochs is as follows:

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



The graph of validation accuracy and accuracy plotted for various epochs is as follows:

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['acc','val_acc'])
plt.title('accuracy')
plt.xlabel('epoch')
```



Accuracy of training data set: 99.85%

Accuracy of validation data set: 95.22%

CHAPTER 4: MODEL CHECKING

4.1 DEFINITION

In computer science, **model checking** or **property checking** is a method for checking whether a finite-state model of a system meets a given specification (a.k.a. correctness). This is typically associated with hardware or software systems, where the specification contains liveness requirements (such as avoidance of live lock) as well as safety requirements (such as avoidance of states representing a system crash).

In order to solve such a problem algorithmically, both the model of the system and its specification are formulated in some precise mathematical language. To this end, the problem is formulated as a task in logic, namely to check whether a structure satisfies a given logical formula. This general concept applies to many kinds of logic as well as suitable structures. A simple model-checking problem consists of verifying whether a formula in the propositional logic is satisfied by a given structure.

4.2 OVERVIEW

Property checking is used for verification when two descriptions are not equivalent. During refinement, the specification is complemented with details that are unnecessary in the higher-level specification. There is no need to verify the newly introduced properties against the original specification since this is not possible. Therefore, the strict bi-directional equivalence check is relaxed to a one-way property check. The implementation or design is regarded as a model of the circuit, whereas the specifications are properties that the model must satisfy.

An important class of model-checking methods has been developed for checking models of hardware and software designs where the specification is given by a temporal logic formula. Pioneering work in temporal logic specification was done by Amir Pnueli, who received the 1996 Turing award for "seminal work introducing temporal logic into computing science". Model checking began with the pioneering work of E. M. Clarke, E. A. Emerson, by J. P. Queille, and J. Sifakis. Clarke, Emerson, and Sifakis shared the 2007 Turing Award for their seminal work founding and developing the field of model checking.

Model checking is most often applied to hardware designs. For software, because of undecidability (see computability theory) the approach cannot be fully algorithmic; typically, it may fail to prove or disprove a given property. In embedded-systems hardware, it is possible to validate a specification delivered, i.e., by means of UML activity diagrams or control interpreted Petri nets.

The structure is usually given as a source code description in an industrial hardware description language or a special-purpose language. Such a program corresponds to a finite state machine (FSM), i.e., a directed graph consisting of nodes (or vertices) and edges. A set of atomic propositions is associated with each node, typically stating which memory elements are one. The nodes represent states of a system, the edges represent possible transitions which may alter the state, while the atomic propositions represent the basic properties that hold at a point of execution.

Formally, the problem can be stated as follows: given a desired property, expressed as a temporal logic formula *p*, and a structure *M* with initial state*s*, decide if M,s|=p. If M is finite, as it is in hardware, model checking reduces to a graph search.

CONCLUSION

5.1 SUMMARY

This internship report proposes a CNN based model for identifying various images and classifying them into different classes. This model was built using LeNet architecture which contains seven layers and finally resulting into the softmax activation which has 43 nodes representing the 43 classes. This model was trained on The German Traffic Signs Data Set and resulted in a training set accuracy of 99.85% and a validation set accuracy of 95.22%. For the training process data augmentation was performed such that it would result in a larger data set and a minimum number of training examples for each class to avoid biasing towards the classes with larger number of examples. The model is then tested with multiple random images from various sources which resulted in the accurate outcome in all the cases. The training procedure ensures a low training time for the model. The model is also suitable for real-time application due to its low computational cost at each stage. As such, it can be implemented in a standard PC configuration by using a low-power GPU.

5.2 FUTURE SCOPE

Object detection is a computer vision technique that allows us to identify and locate objects in an image or video. With this kind of identification and localization, object detection can be used to count objects in a scene and determine and track their precise locations, all while accurately labeling them. Thus Object detection could be implemented on real time images or video frames and the resulting could be fed into Traffic Sign Classification module and output various classes in real-time. This module structure could be used to train various other similar data sets such as Traffic Signals detection and classification, Lane identification, Car detection, Obstacle Identification and various other factors which need to be checked for Autonomous Driving and can then be done with Model Checking to implement into a real life Autonomous Vehicle.

REFERENCES AND EXTERNAL LINKS

Data Set: 'https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/traffic-signs-data.zip'

Data Augmentation: 'https://github.com/vxy10/ImageAugmentation'

CNN: 'https://en.wikipedia.org/wiki/Convolutional_neural_network'

LeNet: 'http://yann.lecun.com/exdb/lenet/'

Model Checking: 'https://en.wikipedia.org/wiki/Model_checking'

Raia Hadsell, Pierre Sermanet, Marco Scoffier, Ayse Erkan, Koray Kavackuoglu, Urs Muller and Yann LeCun: **Learning Long-Range Vision for Autonomous Off-Road Driving**, *Journal of Field Robotics, 26(2):120-144, February **2009**, \cite{hadsell-jfr-09}.*

Implementation of the above model was done in Tensorflow, Keras.

Detailed Implementation can be found on GitHub / Google Colab