TRAFFIC SIGN CLASSIFICATION USING CONVOLUTIONAL NEURAL NETWORKS

PROJECT REPORT

Submitted in the partial fulfillment of requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

in the faculty of

COMPUTER SCIENCE AND ENGINEERING

Submitted by

G RAVI SEKHAR	[17021A0536]
V SAI TEJA	[17021A0532]
G SAI SIREESHA	[17021A0529]
D HEMA	[18025A0566]

Under the supervision of

Dr. S. CHANDRA SEKHAR

Assistant Professor

UNIVERSITY COLLEGE OF ENGINEERING KAKINADA (A)

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY KAKINADA
KAKINADA-533003, A.P, INDIA

2020-2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIVERSITY COLLEGE OF ENGINEERING KAKINADA (A)

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY

KAKINADA-533003, A.P, INDIA

DECLARATION FROM THE STUDENTS

We hereby declare that the work described in this literature survey, entitled **"*Traffic Sign Classification using Convolutional Neural Networks*",** which is being submitted by us in partial fulfillment of the requirements for the award of the degree of ***Bachelor of Technology(B.Tech)*** in the faculty of Department of Computer Science and Engineering to the University College of Engineering Kakinada(A), Jawaharlal Nehru Technological University Kakinada is the result of investigation carried out by us under the supervision of **Sri. S. Chandra Sekhar, Assistant Professor,** Department of Computer Science and Engineering.

The work is original and has not been submitted to any other University or Institute for the award of any degree or diploma.

Place:Kakinada.

Date:

Signature:

G Ravi Sekhar	[17021A0536]
V Sai Teja	[17021S0532]
G Sai Sireesha	[17021A0529]
D Hema	[18025A0566]

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIVERSITY COLLEGE OF ENGINEERING KAKINADA (A)

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY

KAKINADA-533003, A.P, INDIA

CERTIFICATE FROM THE SUPERVISOR

This is to certify that the project work entitled "***Traffic Sign Classification using Convolutional Neural Networks***" being submitted by G Ravi Sekhar (17021A0536), V Sai Teja (17021A0532), G Sai Sireesha (17021A0529), D Hema (18025A0566) in partial fulfillment of the requirements for the award of the degree of ***Bachelor of Technology(B.Tech)*** in faculty of Computer Science and Engineering to the University College of Engineering Kakinada(A), Jawaharlal Nehru Technological University Kakinada is a record of bonafide project work carried out by them under my guidance in the Department of Computer Science and Engineering.

Signature of the supervisor

Dr. S. CHANDRA SEKHAR

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIVERSITY COLLEGE OF ENGINEERING KAKINADA (A)

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY

KAKINADA-533003, A.P, INDIA

CERTIFICATE FROM THE HEAD OF THE DEPARTMENT

This is to certify that the project work entitled "***Traffic Sign Classification using Convolutional Neural Networks***" being submitted by G Ravi Sekhar (17021A0536), V Sai Teja (17021A0532), G Sai Sireesha (17021A0529), D Hema (18025A0566) in partial fulfillment of the requirements for the award of the degree of ***Bachelor of Technology(B.Tech)*** in faculty of Computer Science and Engineering to the University College of Engineering Kakinada(A), Jawaharlal Nehru Technological University Kakinada - 533003. A.P. is a record of bonafide project work carried out by them at our department.

Signature of Head of the Department

Dr. D. HARITHA

Head & Prof of CSE

ACKNOWLEDGEMENTS

We express our deep gratitude and regards to Sri. S. Chandra Sekhar, internal guide and Assistant Professor, Department of Computer Science and Engineering, University College of Engineering (Autonomous), Jawaharlal Nehru Technological University, Kakinada, for his encouragement and valuable guidance in bringing shape to this dissertation.

We are grateful to Dr. D. Haritha, Head of the Department of Computer Science and Engineering, University College of Engineering (Autonomous), Jawaharlal Nehru Technological University, Kakinada for her encouragement and motivation.

We are thankful to all the Professors and Faculty Members in the department for their teachings and academic support and thanks to the Technical Staff and Non-Teaching staff in the department for their support.

G Ravi Sekhar	**[17021A0536]**
V Sai Teja	**[17021S0532]**
G Sai Sireesha	**[17021A0529]**
D Hema	**[18025A0566]**

ABSTRACT

Traffic Sign Classification can assist drivers in making the right decisions at the right time for safe driving. It is indeed vital in intelligent autonomous vehicles. This paper proposes a Convolutional Neural Network (CNN)-based approach for classifying traffic signs. Four convolutional layers, two pooling layers, and two fully connected layers make up the CNN architecture used in this model. The dropout layers have been used to prevent the network from overfitting. Image augmentation is applied to capture invariance by generating additional images. Various optimizers are employed to evaluate the performance of the suggested design, with the Adam (Adaptive Moment Estimation) optimizer proving to be the most effective. The loss function used is categorical cross-entropy. TensorFlow is used for the implementation of CNN. The model is trained and tested using the German Traffic Sign Recognition Benchmark (GTSRB) dataset, and the result reveals that the classification accuracy rate can reach 92.26%.

Keywords - *Computer Vision (CV), Feature extraction, Convolutional Neural Network (CNN), TensorFlow, Training model, Keras, ReLu, Softmax, Classification, Traffic Signs.*

TABLE OF CONTENTS

LIST OF FIGURES

Chapter - 1

INTRODUCTION

1.1. AN OVERVIEW OF THE SYSTEM

The process of classifying traffic signs along the road, such as speed limit signs, yield signs, merge signals, and so on, is known as traffic sign classification. Being able to automatically classify traffic signs, one can develop smarter cars efficiently. It is one of the most advantageous real-time applications of Computer Vision(CV). Traditional Machine Learning based methods were widely used for traffic sign classification, but those methods were soon replaced by Deep Learning-based classifiers. Convolutional Neural Networks(ConvNet/CNN) is a type of deep neural network which extracts features and patterns from the provided images. It is responsible for classifying the images into respective classes.

Traffic sign classification is one of the most important integral parts of autonomous vehicles and advanced driver assistance systems. Traffic sign classification is particularly important with regards to self-governing vehicle innovation. Most of the time drivers miss traffic signs due to different obstacles and lack of attentiveness. Automating the process of classification of the traffic signs would help reduce accidents. Traffic sign classification is an intriguing part of Computer Vision. Traffic sign images are affected by adverse variation due to illumination, orientation, the speed variation of vehicles, etc.,

Traffic sign classification is one of the applications in Image Classification. Traffic signs are road facilities that convey, guide, restrict, warn, or instruct information using words or symbols. Traffic sign boards have two major roles. It assists the driver to the correct destination and indicates the condition of the roads. Traffic signs can be classified on the basis of their shape, size, and color. Traffic signboards carry a significant amount of information that will help people even if they are not from the same city or country.

CNN is a Deep Learning algorithm that can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image, and be able to differentiate one from the other. In recent years, CNN has achieved great success in object detection tasks. The role of the CNN is to reduce the images into a form that is easier to process, without losing features that are critical for getting good predictions. A ConvNet is able to successfully capture the spatial and temporal dependencies in an image through the application of relevant filters.

1.2. INTRODUCTION TO THE SYSTEM

In the world of Artificial Intelligence and advancement in technologies, many researchers and big companies like Tesla, Uber, Google, Mercedes-Benz, Toyota, Ford, Audi, etc are working on autonomous vehicles and self-driving cars. The vehicles should be able to interpret traffic signs and make decisions accordingly. This works by building a Deep Neural Network model that can classify traffic signs present in the image into different categories.

As input, a traffic sign image is provided. To analyze the image, the OpenCV library is employed. The model is built using CNN, which extracts features and patterns from the provided images. CNN is a type of deep neural network often used to process visual images in Deep Learning. The data from the dataset is extracted and the extracted images are often used to train the model. After the model has been trained, it can classify an image and determine which traffic sign it belongs to.

A convolutional neural network is an efficient way of identifying patterns from an image. In the project, CNN does three major tasks like convolution, activation, and Pooling. convolution refers to both the result function and to the process of computing it. Activation refers to the use of activation functions that determine the output of the network, and Pooling is done to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network. The convolution Neural Network uses Tensorflow as the backend.

Tensorflow is a commonly used open-source machine learning framework for implementing neural networks. Tensorflow aids in the training of the model to recognize patterns in images. Tensorflow bundles together Machine Learning and Deep Learning models and algorithms. It uses Python as a convenient front-end and runs it efficiently in optimized C++.

Images are loaded into python lists which are then converted into NumPy arrays as they are more suitable for performing mathematical operations on the images. To the model, an image is given in the form of a matrix.

A filter whose size is usually less than the size of the matrix strides over the image matrix. This filter contains certain stride values such that each filter gives different outputs, thus recognizing different patterns. The main objective of applying the filter is to minimize image noise and acquire only the essential portions of the image. The filter goes from left to right across the matrix with a specific stride value and with the same stride value throughout the input image. After applying the dot product between the weights of the filter and a tiny region of input to which they are all linked, the convolutional layer outputs a convolved feature map.

Once a full matrix is traversed, an activation function called ReLu is applied to allow non-linearity, and the image pattern is determined. This undergoes a pooling layer where the dimension of the matrix is reduced. The next step is a fully connected layer, where it consists of a multilayer perceptron that holds the values. The softmax activation function is used to classify the traffic sign images.

1.3. BASIC STEPS IN THE SYSTEM

1.3.1. DOWNLOADING DATASET:

Download the German Traffic Sign Recognition Benchmark (GTSRB) dataset which includes images of 43 different types of traffic signs. Images from the dataset are loaded into Python lists, which are subsequently transformed into NumPy arrays so that mathematical operations can be performed on them.

1.3.2. DATA AUGMENTATION:

Data Augmentation can be effectively used to train the Deep Learning models in such applications. It is generally applied using a popular class in Keras called ImageDataGenerator and some of the simple transformations such as rotation, translation, zoom_range, shear_range, are applied to the training data after splitting it into test and train data. Data Augmentation is generally used if the number of images in the dataset is small. One of the fields in Deep Learning such as CV requires a lot of images to train the model accurately and for that purpose, we use different data augmentation techniques which generate several other alternate images for an image in the training set by applying transformations on it so that size of the training data can be increased.

1.3.3. BUILDING MODEL:

The approach for building this traffic sign classification model is discussed in four steps:

Step 1: Explore the dataset

Step 2: Build a CNN model.

Step 3: Train and validate the model.

Step 4: Test the model with a dataset.

1.3.4. TRAIN THE MODEL:

After building the model using the above-mentioned CNN layers the next step is to compile the model where the loss function is specified as 'categorical_crossentropy' and the optimizer as 'Adam' optimizer.

1.4. INTRODUCTION TO OpenCV:

OpenCV is a Python library that is designed to solve computer vision problems. OpenCV was originally developed in 1999 by Intel but later it was supported by Willow Garage.

OpenCV supports a wide variety of programming languages such as C++, Python, Java, etc. Support for multiple platforms including Windows, Linux, and macOS.

OpenCV Python is nothing but a wrapper class for the original C++ library to be used with Python. Using this, all of the OpenCV array structures get converted to or from NumPy arrays.

This makes it easier to integrate it with other libraries which use NumPy. For example, libraries such as SciPy and Matplotlib.

1.4.1. HISTORY OF OpenCV:

Officially launched in 1999 the OpenCV project was initially an Intel Research initiative to advance CPU-intensive applications, part of a series of projects including real-time ray tracing and 3D display walls. The main contributors to the project included a number of optimization experts in Intel Russia, as well as Intel's Performance Library Team. In the early days of OpenCV, the goals of the project were described as:

1.Advance vision research by providing not only open but also optimized code for basic vision infrastructure. No more reinventing the wheel.

2.Vision knowledge by providing a common infrastructure that developers could build on, so that code would be more readily readable and transferable.

3.Advance vision-based commercial applications by making portable, performance-optimized code available for free – with a license that did not require code to be open or free itself.

The first alpha version of OpenCV was released to the public at the IEEE. Conference on Computer Vision and Pattern Recognition in 2000 and five betas were released between 2001 and 2005. The first 1.0 version was released in 2006. A version 1.1 "pre-release" was released in October 2008.

The second major release of the OpenCV was in October 2009. OpenCV 2 includes major changes to the C++ interface, aiming at easier, more type-safe patterns, new functions, and better implementations for existing ones in terms of performance (especially on multi-core systems).

Official releases now occur every six months and development is now done by an independent Russian team supported by commercial corporations. In August 2012, support for OpenCV was taken over by a non-profit foundation OpenCV.org, which maintains a developer and user site. In May 2016, Intel signed an agreement to acquire Itseez, a leading developer of OpenCV.

1.4.2. APPLICATIONS OF OpenCV:

OpenCV is being used for a very wide range of applications which include:

- 2D and 3D feature toolkits
- Facial recognition system
- Gesture recognition
- Human-computer interaction (HCI)
- Motion understanding
- Object detection
- Medical image analysis
- Robot and driver-less car navigation and control
- Augmented Reality

1.4.3. BASIC OPERATIONS IN OpenCV:

1.4.3.1. Loading an image using OpenCV:

The following piece of code shows how to load an image using OpenCV.

```
import cv2
# colored Image
img = cv2.imread (&l1dquo;Penguins.jpg&rdquo;,1)
# Black and White (grayscale)
img_1 = cv2.imread (&ldquo;Penguins.jpg&rdquo;,0)
```

The image is read using the ‘imread’ function. The parameter in the imread function is one for a color image and is zero for a black and white image.

1.4.3.2. Image Shape/Resolution:

A shape sub-function can be used to print out the shape of the image. The following piece of code helps to get the shape of the image in terms of the number of rows and number of columns.

```
import cv2
# Black and White (grayscale)
Img = cv2.imread (&ldquo;Penguins.jpg&rdquo;,0)
Print(img.shape)
```

The shape of the image means the shape of the NumPy array and the matrix consists of 768 rows and 1024 columns.

1.4.3.3. Displaying the image:

The ‘imshow’ function is used to display the image by opening a window. There are 2 parameters to the imshow function which are the name of the window and the image object to be displayed. The waitKey( ) makes the window static until the user presses a key. The parameter passed to it is the time in milliseconds.

```
import cv2
# Black and White (grayscale)
Img = cv2.imread (&ldquo;Penguins.jpg&rdquo;,0)
cv2.imshow(&ldquo;Penguins&rdquo;, img)
cv2.waitKey(0)
# cv2.waitKey(2000)
cv2.destroyAllWindows()
```

1.4.3.4. Resizing the image:

The ‘resize’ function is used to resize an image to the desired shape. The parameter here is the shape of the newly resized image. The image object changes from img to resized_image, because the image object changes now.

```
import cv2
# Black and White (grayscale)
img = cv2.imread (&ldquo;Penguins.jpg&rdquo;,0)
resized_image = cv2.resize(img, (650,500))
cv2.imshow(&ldquo;Penguins&rdquo;, resized_image)
cv2.waitKey(0)
```

cv2.destroyAllWindows()

1.4.4. Programming Languages:

OpenCV is written in C++ and its primary interface is in C++, but it still retains a less comprehensive though extensive older C interface. There are bindings in Python, Java, and MATLAB/OCTAVE. The API for these interfaces can be found in the online documentation. Wrappers in other languages such as C#, Perl, Ch, Haskell, and Ruby have been developed to encourage adoption by a wider audience. Since version 3.4, OpenCV.js is a JavaScript binding for a selected subset of OpenCV functions for the web platform. All of the new developments and algorithms in OpenCV are now developed in the C++ interface.

1.4.5. Hardware Requirements:

If the library finds Intel's Integrated Performance Primitives on the system, it will use these proprietary optimized routines to accelerate itself. A CUDA-based GPU interface has been in progress since September 2010. An OpenCL-based GPU interface has been in progress since October 2012, documentation for version 2.4.13.3 can be found at docs.opencv.org.

1.4.6. OS Support:

OpenCV runs on the following desktop operating systems: Windows, Linux, macOS, FreeBSD, NetBSD. OpenCV runs on the following mobile operating systems: Android, iOS, Maemo, BlackBerry 10. The user can get official releases from SourceForge or take the latest sources from GitHub. OpenCV uses CMake.

1.5 INTRODUCTION TO GUI(TKINTER)

Tkinter is Python's de-facto standard GUI (Graphical User Interface) package. It is a thin object-oriented layer on top of Tcl/Tk. Tkinter is the most popular programming package for graphical user interface or desktop apps. Running python -m Tkinter from the command line should open a window demonstrating a simple Tk interface, letting you know that Tkinter is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

1.5.1 Tkinter Modules

Most of the time, Tkinter is all you really need, but a number of additional modules are available as well. The Tk interface is located in a binary module named _tkinter. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter. In addition to the Tk interface module, Tkinter includes a number of Python modules, tkinter.constants being one of the most important. Importing tkinter will automatically import tkinter.constants, so, usually, to use Tkinter all you need is a simple import statement: import tkinter Or, more often:

*from tkinter import **

class tkinter.Tk(screenName=None, baseName=None, className='Tk', useT k=1)

The Tk class is instantiated without arguments. This creates a top-level widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter.

tkinter.Tcl(screenName=None,baseName=None,className='Tk',useTk=0)

The Tcl() function is a factory function that creates an object much like that created by the Tk class, except that it does not initialize the Tk subsystem. This is most often useful when driving the Tcl interpreter in an environment where one doesn't want to create extraneous top-level windows, or where one cannot (such as Unix/Linux systems without an X server). An object created by the Tcl() object can have a Toplevel window created (and the Tk subsystem initialized) by calling its loadtk() method.

Other modules that provide Tk support include:

Tkinter label:

Tkinter Label is a widget that is used to implement display boxes where you can place text or images.

w = Label (master, option, ...)

Where parameters represent,

- **master** -This represents the parent window.
- **options** - represent the background color, font, and foreground color

Tkinter Filedialog:

The Tkinter filedialog comes in several types. Which type you need really depends on your application's needs. All of them are method calls.

You can open a single file, a directory, save as file, and much more. Each dialog made with the example below is a different type of dialog.

The following methods represent different operations for various filedialog types.

import tkinter.filedialog

tkinter.filedialog.asksaveasfilename()

tkinter.filedialog.asksaveasfile()

tkinter.filedialog.askopenfilename()

tkinter.filedialog.askopenfile()

tkinter.filedialog.askdirectory()

tkinter.filedialog.askopenfilenames()

tkinter.filedialog.askopenfiles()

Tkinter Pack:

The Pack geometry manager packs widgets in rows or columns. We can use options like fill, expand, and side to control this geometry manager.

tkinter.pack(pack_options)

Where pack_options will have the following parameters.

- **expand** − When set to true, the widget expands to fill any space not otherwise used in the widget's parent.
- **fill** − Determines whether widget fills any extra space allocated to it by the packer, or keeps its own minimal dimensions: NONE (default), X (fill only horizontally), Y (fill only vertically), or BOTH (fill both horizontally and vertically).
- **side** − Determines which side of the parent widget packs against TOP (default), BOTTOM, LEFT, or RIGHT.

Tkinter Entry:

The Entry Widget is a Tkinter Widget used to Enter or display a single line of text.

entry = tkinter.Entry(parent, options)

Where parameters include,

- **Parent:** The Parent window or frame in which the widget to display.
- **Options:** The various options provided by the entry widget are:
    - **bg**: The normal background color displayed behind the label and indicator.
    - **bd:** The size of the border around the indicator. Default is 2 pixels.
    - **font**: The font used for the text.
    - **fg:** The color used to render the text.
    - **justify:** If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.

In this project, Tkinter is used to build the user interface for the system. The display window consists of two buttons named ‘Upload an image’ for uploading and another button named ‘Classify image’ for classifying the image. Once the image is uploaded, the classify button is enabled and on clicking it, it classifies the image into the respective class.

Chapter - 2

LITERATURE SURVEY

Yan Lai et.al (2018) in their paper "***Traffic Signs Recognition and Classification based on Deep Feature Learning***", have presented a novel method for traffic signs recognition and classification based on Convolutional Neural Network and Support Vector Machine (CNN-SVM). In this method, the YCbCr color space is introduced in CNN to divide the color channels for feature extraction. SVM is connected with the last layer of CNN for further classification, which contributes to better training results. The experiments are conducted on a real world data set with images and videos captured from ordinary car driving. The experimental results show that compared with the state-of-the-art methods, this method achieves the best performance on traffic signs recognition and classification, with a highest 98.6% accuracy rate.

Syed Hussain et.al (2018) in their paper ***"A Survey of Traffic Sign Recognition Systems Based on Convolutional Neural Networks"***, have implemented a Fast Branch CNN model, which imitates biological mechanisms to become more efficient. The Fast Branch CNN model challenged the assumptions of past models, and this technology can only advance further if new models attempt to do the same. Before Fast Branch was developed, most mechanisms operated under the assumption that the system needed to see the entire image, and this turned out to be a drawback. The CNN architecture used in this model contains 3 sets of convolutional layers which comprise convolutions followed by a pooling layer and a normalization layer. The accuracy is 98.52%. While conventional models continue to go step-by-step, Fast Branch CNNs present a more selective and time-efficient model, and they do so by eliminating past assumptions in the conventional wisdom.

Djebbara Yasmina et.al (2018) in their paper "***Traffic signs recognition with Deep Learning*** ", have presented a Deep Learning based road traffic signs recognition method for the development of Advanced Driver Assistance Systems (ADAS) and autonomous vehicles. The system architecture is designed to extract main features from images of traffic signs to classify them under different categories. The method uses a modified LeNet-5 network to extract a deep representation of traffic signs to perform the recognition and constitutes a Convolutional Neural Network (CNN) modified by connecting the output of all convolutional layers to the Multilayer Perceptron (MLP). The training is conducted using the German Traffic

Sign Dataset and achieves good results on recognizing traffic signs. It is likely that better results would be obtained by reinforcing the convolution stage of the network with more layers in order to extract more features. It also would be interesting to exclude confusions by comparing classes with the highest proportions in the confusion matrix and pull out their common factors to reverse them by image adjustment.

Tarequl Islam (2019) in his paper "***Traffic sign detection and recognition based on convolutional neural networks***", have proposed a system that will detect and classify different types of traffic signs from images. The number of signs used in this paper for classification is 28, which are used all around the globe. Two separate neural networks have been used for detection and recognition purposes; one classifies the sign and other the shape. Image augmentation has been used to create the training and validation dataset. The images are processed to find the region of interest, which is then fed to two CNN classifiers for classification. The results are moderate and it can be improved by testing different neural network structures. New methods of data augmentation can also be applied to make the classifier more robust. Real-time detection and recognition can also be implemented in the future.

Hoanh Nguyen (2020) in his paper "***Fast Traffic Sign Detection Approach Based on Lightweight Network and Multilayer Proposal Network"***, has proposed a Deep Learning-based framework for fast and efficient traffic sign detection. To improve the inference speed of the proposed framework, the ESPNetv2 network is adopted as the base network. To enhance the performance of small traffic sign detection, a deconvolution module is adopted to generate an enhanced feature map by aggregating a lower-level feature map with a higher-level feature map. Furthermore, an improved region proposal network, which includes a 1×1 convolution layer to reduce the number of parameters in the subsequent convolutional layers and a 3×3 dilated convolution to enlarge the receptive field, is designed to increase the performance of a proposal generation stage. In the experiments, two widely used datasets are adopted to evaluate the effectiveness of each enhanced module and the whole framework, including the GTSDB dataset and TT-100K dataset. Experimental results on these datasets show that the proposed framework improves the performance of traffic sign detection under challenging driving conditions and meets the real-time requirement of an advanced driver assistant system.

Ratheesh Ravindran et.al (2019) in their paper "***Traffic Sign Identification Using Deep Learning***", have focused on the selection of Deep Neural Networks (DNN) based on the application-oriented performance by taking into consideration the mean Average Precision (mAP) and Frames Per Second (FPS) as the major evaluation criteria. The real-time trials and simulation of the DNN using ROS-Gazebo for traffic sign detection was performed. To improve the performance of the DNN in detecting traffic signs, a long short-term memory artificial recurrent neural network, called "Tesseract," was integrated with the underlying DNN to detect the text from traffic signs. The proposed new architecture was shown to perform well both through ROS-Gazebo simulations and real-time trials on the Polaris Gem e2 platform. The FPS performance can be further improved by using the NVIDIA Profiler and optimizing the behavior of the GPU platform.

Adonis Santos et.al (2019) in their paper "***Traffic Sign Detection and Recognition for Assistive Driving***", have implemented a traffic sign detection and recognition system using Python programming language and scikit learn libraries. The model is composed of the pre-processing, detection and recognition stages. Image pre-processing technique improves the detection and recognition process. Bilateral filtering provided 2.02% detection improvement, 0.68% less in non-detection and 1.35% less in false detection compared to the model without bilateral filtering. The detection efficiency has been improved by implementing color thresholding before Hough transform. This selects the region of interests corresponding to the red circular boundary of the target traffic signs resulting in an accuracy of 68.25% for dataset from online sources and effective accuracy of 75% for dataset from local roads. Among the evaluated machine learning classifiers, the Multilayer Perceptron Classifier model provided the highest accuracy for traffic sign recognition of 0.9. The model also has the best result in precision at 0.9, recall at 0.9 and f1 score at 0.91.

Aashrith Vennelakanti et.al (2019) in their paper "***Traffic Sign Detection and Recognition using a CNN Ensemble***", have proposed a method for Traffic Sign Detection and Recognition using image processing for the detection of a sign and an ensemble of Convolutional Neural Networks (CNN) for the recognition of the sign. CNNs have a high recognition rate, thus making it desirable to use for implementing various computer vision tasks. TensorFlow is used for the implementation of CNN. This method proposes a feed-forward network with six convolutional layers. This also has fully connected hidden layers with dropout layers in between to prevent overfitting. All the layers of the proposed CNN have Rectified Linear Unit (ReLu) activation. The output of the 6th convolutional layer is fed to a fully connected layer, which uses a flatten function to flatten the output at that point. The flattened output is given to the final layer which uses SoftMax activation. The overall performance could also be improved and customized with the help of more datasets and from different sources. This method achieved higher than 99% recognition accuracy for circular signs on the Belgium and German data sets.

Jihene Rezgui et.al (2019) in their "***Traffic Sign Recognition Using Neural Networks Useful for Autonomous Vehicles***", have proposed a new Java platform called "ARIBAN", which provides algorithms for recognition and classification of traffic signs with Multi-Layer Perceptron Neural Networks (MLPNN). A back propagation algorithm is used in a supervised manner to establish the network. The post-processing combines the results to make a recognition decision. Then, the neural network has been trained with several various traffic signs. Recognition parameters like learning rate and layers numbers have been taken into account to evaluate the proposed program. Experimental results have demonstrated the effectiveness of the proposed system. Furthermore, the proposed system was deployed within architecture for autonomous driving.

Gabriel Noya Doval et.al (2019) in their paper "***Traffic Sign Detection and 3D Localization via Deep Convolutional Neural Networks and Stereo Vision"***, have presented an algorithm to detect, classify and spatially locate multiple traffic signs in different scenarios. Detection and classification are made simultaneously via YOLOv3, using RGB images. 3D sign localization is achieved by estimating the distance from the traffic sign to the vehicle, by looking at detector bounding boxes and the disparity map generated by stereo vision. Moreover, a new traffic sign dataset called LSITSD is created to solve the issues that most traffic sign datasets have, such as missing or incorrect labels, or the small amount of images provided. The obtained results show the performance and the robustness of the proposed algorithm in detecting the traffic signs and the accuracy of the 3D localization estimation of the detected traffic signs. The implemented modified architecture improves detection results at lower resolutions and decreases its efficiency at higher resolutions.

Smit Mehta et.al (2019) in their paper "***CNN based Traffic Sign Classification using Adam Optimizer***", have proposed an approach based on the deep convolutional network for classifying traffic signs. The Belgium traffic sign dataset (BTSD) is used for evaluation and experiment results show that the proposed method can achieve competitive results compared with state of the art approaches. Furthermore, it uses dropout to overcome the problem of overfitting as it randomly drops some of the units from the neural network and it is also considered to be the most efficient way of model averaging. The proposed network architecture uses softmax as activation in an output layer because it calculates a probability of every possible class. For training the network Adam optimizer is used and it is observed that it adapts faster compared to other optimizers like SGD.

Zhao Dongfang et.al (2019) in their paper "***Traffic sign classification network using inception module***", have proposed a GoogLeNet based convolutional neural network for traffic signs. First, an Inception Module that conforms to the Hebbian principle is built and then a neural network with a sparse structure. The over-fitting phenomenon and the amount of calculation of the deep network are alleviated by suitable sparse structural units. The network is built by structural units and has strong portability. The structural units in the network can be easily transplanted to other networks for training. The convolutional neural network reduces the number of fully connected layer parameters by a factor of 20 through a two-layer pooling layer. The sparse network structure and continuous pooling layer greatly speed up the classification of the network. In the experiments, the classification accuracy of the neural network on the GTSRB data set can reach 98%, and 100% on the MNIST and pneumonia data sets. This also proves that this convolutional neural network has high classification accuracy and can be applied to the automatic driving technology.

Xie Bangquan et.al (2019) in their "***Real-Time Embedded Traffic Sign Recognition Using Efficient Convolutional Neural Network***", have introduced a new efficient Traffic Sign Classification(TSC) network called ENet (efficient network) and a Traffic Sign Detection(TSD) network called EmdNet (efficient network using multiscale operation and depthwise separable convolution). Data mining and multiscale operation are used to improve the accuracy and generalization ability and depthwise separable convolution (DSC) to improve the speed. The resulting ENet possesses 0.9 M parameters (1/15 the parameters of the start-of-the-art method) while still achieving an accuracy of 98.6 % on the German Traffic Sign Recognition benchmark (GTSRB). In addition, EmdNet' s backbone network is designed according to the principles of ENet. The EmdNet with the SDD Framework possesses only 6.3 M parameters, which is similar to MobileNet's scale.

Alexander Wong et.al (2018) in their paper "***MicronNet: A Highly Compact Deep Convolutional Neural Network Architecture for Real-Time Embedded Traffic Sign Classification"***, have introduced a highly compact deep convolutional neural network called MicronNet, for real-time embedded traffic sign recognition. By designing a highly optimized network architecture where each layer's microarchitecture is optimized to have as few parameters as possible, along with macroarchitecture augmentation and parameter precision optimization, the resulting MicronNet network achieves a good balance between accuracy and model size as well as inference speed. The resulting MicronNet possess a model size of just 1MB and around 510,000 parameters (27x fewer parameters than state-of-the-art), requires just 10 million multiply-accumulate operations to perform inference (with a time-to-compute of 32.19 ms on a Cortex-A53 high efficiency processor), while still achieving a top-1 accuracy of 98.9% on the German traffic sign recognition benchmark, thus achieving human-level performance. These experimental results show that very small yet accurate deep neural network architectures can be designed for real-time traffic sign recognition that are well-suited for embedded scenarios.

Priya Garg et.al (2019) in their paper "***Traffic Sign Recognition and Classification Using YOLOv2, Faster RCNN and SSD***", have compared Single Shot Detector (SSD), Faster Region Convolutional Neural Network (Faster RCNN) and You Only Look Once (YOLOv2) Deep Learning architectures by applying distinct pretrained Convolutional Neural Network (CNN) models. Experiments have been organized in a wide range to attain distinct models of Faster RCNN, SSD and

YOLOv2 through appropriate modification in algorithms and parameters tuning. In this work, SSD, Faster RCNN and YOLOv2 are trained for 5 different object classes of traffic signs and their outcomes are evaluated. Traditional Evaluation parameters: mAp(mean Average precision) and FPS(Frames per second) are run-down to analyze the accuracy and speed of the algorithms. On analyzing, the accuracy of YOLOv2 outperforms Faster RCNN and SSD by 3.5% and 21% respectively. Also, the learning rate of YOLOv2 is 68% faster than Faster RCNN and 16% faster than SSD. But, when it comes to detection of very small traffic signs, YOLOv2 will not perform upto the mark and Faster RCNN will outperform YOLOv2 in terms of accuracy. So, the performance of YOLOv2 will get limited after a certain extent of object size.

Ying Sun et.al (2019) in their paper "***Traffic Sign Detection and Recognition Based on Convolutional Neural Network***", have proposed a traffic sign recognition technique on the strength of Deep Learning, which mainly aims at the detection and classification of circular signs. They used the GTSRB data set to train and test CNN. The designed light-weight CNN consists of two convolutional layers, two pooling layers and two full connection layers. Firstly, an image is preprocessed to highlight important information. Secondly, Hough Transform is used for detecting and locating areas. Finally, the detected road traffic signs are classified based on Deep Learning. By using image preprocessing, traffic sign detection, recognition and classification, this method can effectively detect and identify traffic signs. Test result displays that the accuracy of this method is 98.2%.

Zhongyu Wang et.al (2019) in their paper "***Research on Traffic Sign Detection Based on Convolutional Neural Network***", have proposed an improved CNN model based on YOLO model, darknet 53 construction. By introducing batch normalization and RPN networks and improving the network structure for traffic sign detection tasks, the YOLO neural network detection model is optimized. They used the TT100K (Tsinghua-Tencent 100K) dataset to evaluate the method. The TT100K dataset is an image dataset of 100,000 images of a real traffic environment created from Tencent Street View images. TT100K contains 30,000 traffic sign instances. These images cover a variety of image changes under light and weather conditions and are widely used in the research and evaluation of traffic sign detection algorithms. The accuracy of the model in the traffic sign detection task is greatly improved, and the detection speed becomes faster. The results show that the method reduces the hardware requirements of the detection system as well.

Chapter - 3

SYSTEM ANALYSIS

3.1. PROPOSED SYSTEM:

3.1.1. DESCRIPTION OF THE SYSTEM:

The main objective of the system is to develop a user interactive assistance system that analyses the traffic signs based on image classification algorithms. By using this system it is able to classify the traffic signs accurately with better results. It extracts the features from the input traffic sign image and classifies it into corresponding traffic sign class labels. For this purpose computer vision is used to input the image and extract the essential features from it. Convolutional neural networks(CNN) is the algorithm used to build models and develop a model with 92% accuracy.

3.1.2. GTSRB DATASET:

Dataset used to develop this model is the GTSRB Dataset (which stands for German Traffic Sign Recognition Benchmark Dataset). The images present in the dataset are the images of the traffic signs. The images are divided into 43 categories, where each category represents a traffic sign. There are about 51,000 images present in the dataset which is divided into training and testing images. Training set contains 39209 labeled images and 12630 images belonging to the testing set.

3.1.3. ADVANTAGES:

The advantages of this proposed system is to classify all traffic sign images in one click. It is one of the faster ways of classifying the traffic sign images into traffic sign class labels. It helps to build smart cars efficiently.

3.2. HARDWARE REQUIREMENTS:

1. RAM : 4 GB
2. Hard Disk : 500 GB
3. Processor : Core i5 or Higher
4. Speed : 2.1 Ghz

3.3. SOFTWARE REQUIREMENTS:

3.3.1. OPERATING SYSTEMS:

In Python, file names, command line arguments, and environment variables are represented using the string type. On some systems, decoding these strings to and from bytes is necessary before passing them to the operating system. Python uses the file system encoding to perform this conversion(see sys.getfilesystemencoding()).This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file, see open(), if you want to manipulate paths, see the os.path module and if you want to read all the lines in all the files on the command line see the fileinput module. For creating temporary files and directories see the tempfile module, and for high-level file and directory handling and directory handling see the shutil module.
The design of all built-in operating system dependent modules of Python Is Such that as long as the same functionality is available, it uses the same interface. for example, the function os.stat(path) returns stat information about path in the same format (which happens to have originatedwiththePOSIX interface). Extensions peculiar to a particular operating system are also available through the os module, but using them is of course a threat to portability. All functions accepting path or file names accept both byte string objects and result in an object of the same type, if a path or filename is returned. On VxWorks, os.fork,

os.execv and os.spawn*p* are not supported. All functions in this module raise OSError (or subclasses thereof) in case of invalid or inaccessible file names and paths or other arguments that have the correct type, but are not accepted by the operating system.

3.3.2. ABOUT PYTHON:

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for RapidApplication Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Python 3.6 picks up where many of those improvements left off and nudges them into new realms. Python 3.5 added syntax used by static type checking tools to ensure software quality; Python 3.6 expands on that idea, which would eventually lead to high-speed statically compiled Python programs. Python3.5 gave us options to write asynchronous functions; Python3.6 bolstered them. But the biggest changes in Python 3.6 lie under the hood, and they open up possibilities that didn't exist before.

3.3.3. JUPYTER NOTEBOOK

Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualization and explanatory text. Uses include: data cleaning and transformation,numerical simulation, statistical modeling, machine learning and muchmore.Jupyter Notebooks can be started with different Python environments. But You have to register an environment with Jupyter Notebooks before it will show up in the options when you click the button to create a new notebook.

Create a New Conda Environment

On a Mac, open a Terminal from Applications > Utilities. On Windows, openAnaconda Navigator and start an Anaconda prompt. We're going to make a new environment called p36 workshop. Do this by typing:

conda create -n p36workshop python=3.6 ipykernel jupyter anaconda

This will prompt you to install some additional packages. Enter "y" when asked to proceed. It will take a while because it's installing all of the standardAnaconda packages into the new environment.

Activate the Environment

Next, activate the new environment. There's a command at the end of the creation script that shows you how to do this. It should look like:

source activate p36 workshop

Register the Environment with IPython

Jupyter Notebook is built on IPython. We need to tell IPython about the environment we just made so Jupyter Notebooks can include it as an option.

ipython kernel install --name p36workshop --user

Now close the Terminal or Prompt window you've been using.

Start Jupyter Notebook

Now, start the Jupyter Notebook by opening it from the Anaconda Navigatorprogram or by typing jupyter notebook at the command line. Checktoseethat p36 workshop now shows up in the list of options when you create a new notebook. You do not need to have the p36 workshop environment active when you start the Jupyter Notebook.

Installing Packages

If you install packages from a Jupyter Notebook using p36 workshop, the packages will install in this new conda environment you've created. If you want to install packages from the command line into this environment,remember to activate the environment first before using conda install or pip install.

3.3.4. LIBRARIES:

3.3.4.1. Tensorflow:

TensorFlow is a Python library for fast numerical computing created and released by Google. It is a foundation library that can be used to create DeepLearning models directly or by using wrapper libraries that simplify the process built on top of TensorFlow. The version of Tensorflow Used In The System is 1.8.

What is TensorFlow?

TensorFlow is an open source library for fast numerical computing.It was created and is maintained by Google and released under the Apache 2.0opensource license. The API is nominally for the Python programming language,although there is access to the underlying C++ API. Unlike other numerical libraries intended for use in Deep Learning like Theano, TensorFlow was designed for use both in research and development and in production systems,not least RankBrain in Google search and the fun DeepDreamproject. It can run on single CPU systems, GPUs as well as mobile devices and large scale distributed systems of hundreds of machines.

How to Install TensorFlow

Installation of TensorFlow is straightforward if you already have a Python SciPy environment. TensorFlow works with Python 2.7 and Python3.3+. You Can follow the Download and Setup instructions on the TensorFlowwebsite. Installation is probably simplest via PyPI and specific instructions for the pip command to use for your Linux or Mac OS X platform aronda Download and Setup webpage. There are also virtualenv and docker images that you can use if you prefer. To make use of the GPU, only Linux is supported and it requires the Cuda Toolkit.

3.3.4.2. Keras:

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go fromideatoresultwith the least possible delay is key to doing good research. Use Keras If you need a deep learning library that: Allows for easy and fast prototyping(through user friendliness, modularity, and extensibility). Supports both convolutional networks and recurrent networks, as well as combinations of the two. Runs seamlessly on CPU and GPU. Kerasis Compatible with: Python 2.7-3.6.

Multi-backend Keras and tf.keras

At this time, we recommend that Keras users who use multi-backend Keras with the TensorFlow backend switch to tf.keras in TensorFlow2.0. tf.keras is better maintained and has better integration with TensorFlowfeatures(eager execution, distribution support and other). Keras 2.2.5 was the last release of Keras implementing the 2.2.* API. It was the last release to onlysupportTensorFlow 1 (as well as Theano and CNTK).

The current release is Keras 2.3.0, which makes significant API changes and adds support for TensorFlow 2.0. The 2.3.0 release will be the last major release of multi-backend Keras. Multi-backend Keras is superseded by tf.keras.

3.3.4.3. Pandas:

Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structure operations for manipulating numerical tables and time series. It is free software released under the three-clause BSD license. The name is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals.

3.3.4.4. Matplotlib:

Matplotlib is a plotting library for the python programming language and its numerical mathematics extension NumPy. It provides an object-orientedAPIfor embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt or GTK+.

Chapter - 4

SYSTEM IMPLEMENTATION

4.1. LOADING DATA:

The dataset containing images of traffic signs which belong to 43 different classes is loaded. Images present in all the 43 folders of the train folder in the dataset are appended to a python list. Loaded images are converted into NumPy arrays as they will have numerical values stored in them. Which will be convenient to perform mathematical operations like convolution, pooling, activation, etc. Labels of the images are stored in another NumPy array and one-hot encoding is applied to them. As we cannot work with the categorical data directly we perform one-hot encoding on them to convert them into numerical values.

4.2. DATA AUGMENTATION:

Amongst the popular Deep Learning applications, CV tasks such as image classification, object detection, and segmentation have been highly successful. Data augmentation can be effectively used to train the Deep Learning models in such applications.

Data Augmentation is generally used if the number of images in the dataset is small. One of the fields in Deep Learning such as CV requires a lot of images to train the model accurately and for that purpose, we use different data augmentation techniques which generate several other alternate images for an image in the training set by applying transformations on it so that size of the training data can be increased. A convolutional neural network that can robustly classify objects even if it's placed in different orientations is said to have the property called invariance. More specifically, a CNN can be invariant to translation, viewpoint, size, or illumination (Or a combination of the above). Data Augmentation improves the performance and different augmentation techniques affect the scale of improvement differently.

In this proposed system, data augmentation is applied using a popular class in Keras called ‘ImageDataGenerator’ to the training data after splitting into test and train data.

Operations that are applied in this implementation are:

- rotation_range
- zoom_range
- width_shift_range
- height_shift_range
- Shear_range

Below Fig. 1 shows the generated images after performing different transformation techniques on an image.

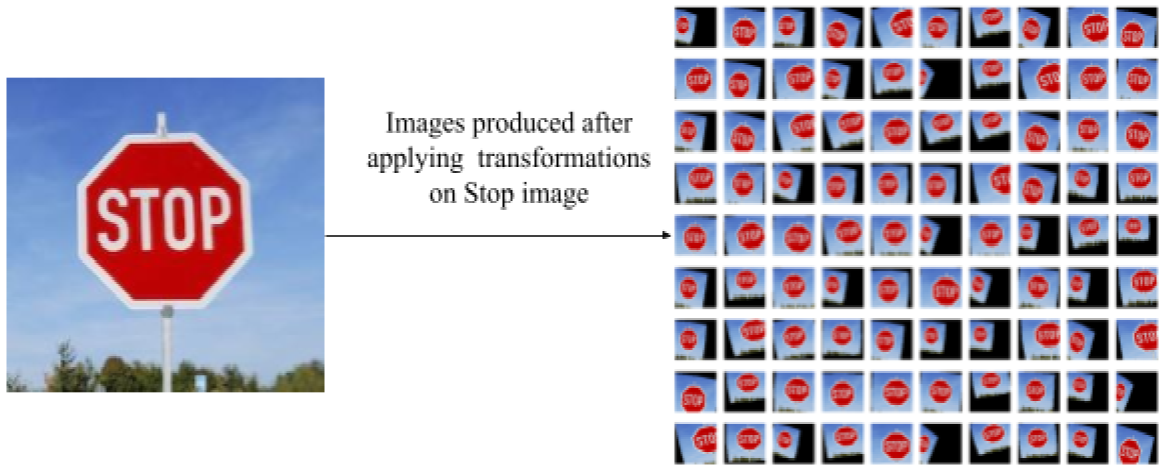

STOP
Images produced after applying transformations on Stop image

Fig. 1. Data Augmentation on an image

4.3. BUILDING THE MODEL:

To build the CNN model, import the Keras library. Keras has an inbuilt model known as the sequential model. A single-layer stack with each layer has exactly one input field and one output field is suitable for a sequential model. Different layers in a sequential model are responsible to extract various kinds of features from the model and classify the given input image. The Convolution layer is the major building block in convolution neural networks. It is generally used to extract high-level features from the image. The pooling layer is used to reduce the dimensions of the feature map produced in the convolution layer. The dropout layer is used to drop the features which are least important by making their weights as 0 in each training phase. The activation function used in the hidden layer is the Rectified Linear Unit(ReLu) as it is more appropriate for the model to learn faster and perform well. A fully connected layer is used to perform the required task of classification and the softmax activation function is used in this layer to classify the image belonging to which class based on its probability. It must contain 43 neurons in it where each neuron represents a class label. Therefore, the model first extracts the features from the input image in convolution, pooling and activation layers and classifies the image into different classes in the fully connected layer. Below Fig. 2 shows the proposed architecture of the CNN model.

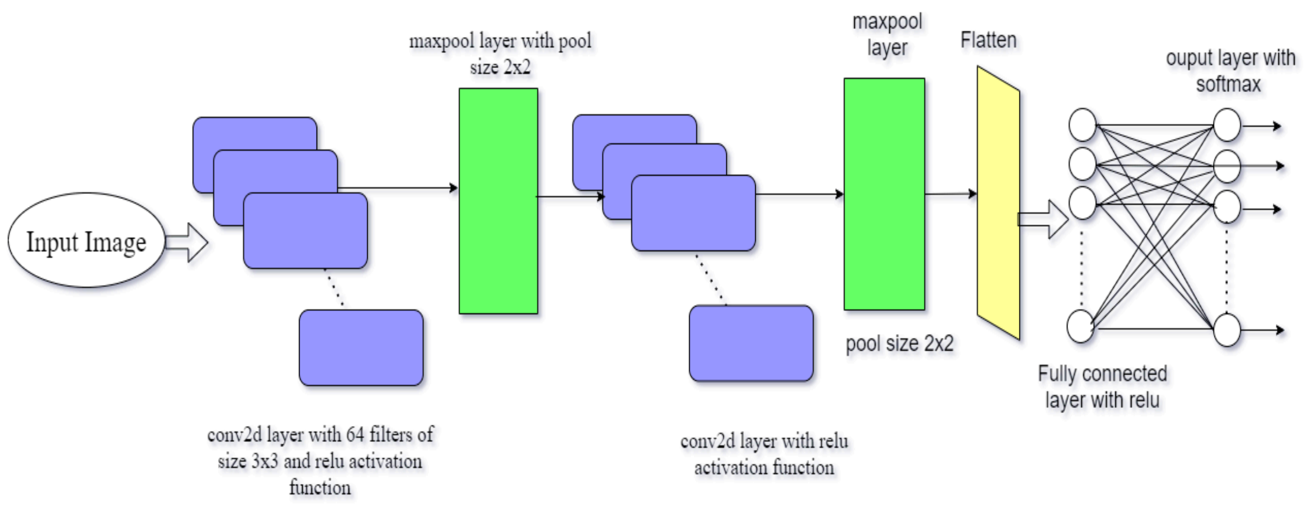

maxpool layer with pool size 2x2
maxpool layer
Flatten
ouput layer with softmax
Input Image
pool size 2x2
Fully connected layer with relu
conv2d layer with 64 filters of size 3x3 and relu activation function
conv2d layer with relu activation function

Fig. 2. Architecture of the CNN model.

4.3.1. CONVOLUTIONAL LAYER:

The concept of leveraging convolution, which generates filtered feature maps stacked on top of each other, is at the core of CNN's functioning. The network is composed of convolution layers, each followed by max-pooling layers with a ratio of 0.25 .The dropout layers are employed in between two convolution layers. The main objective of applying the filter is to minimize image noise and acquire only the essential portions of the image. For instance, if the model is learning how to recognize elephants from a picture with a mountain in the background. If we use a traditional neural network, the model will assign weights to all the pixels, including mountains which are not necessary to recognize an elephant.

Instead, a convolutional neural network will use a mathematical technique to extract only the most relevant pixels. This mathematical operation is called convolution. This technique allows the network to learn increasingly complex features at each layer. The convolution divides the matrix into small pieces to learn the most essential elements within each piece. Below Fig. 3 shows how convolution operation is performed on the input image matrix.

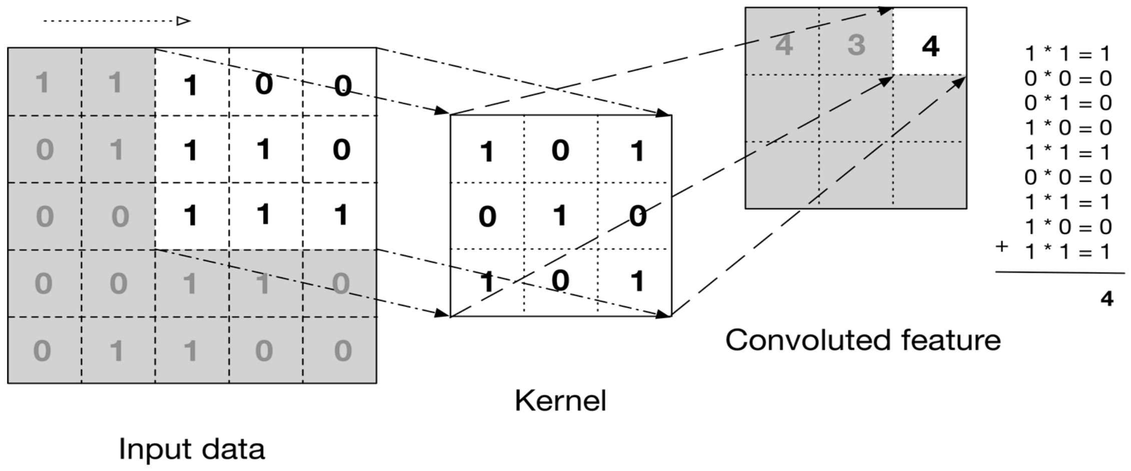


Fig. 3. Convolution Operation

Arithmetic in convolution process:

The filter goes from left to right across the matrix with a specific stride value and with the same stride value throughout the input image. After applying the dot product between the weights of the filter and a tiny region of input to which they are all linked, the convolutional layer outputs a convolved feature map. The input image is transformed to a NumPy array before being convolved (matrix multiplication) with a 3 * 3 filter matrix. The feature map of dimensions less than the input image dimension is the outcome of this matrix multiplication.

The size of the output feature map can be calculated using the following equation(1):

$$Out = W - F + 1 \tag{1}$$

Where, W = input dimensions, F = Respective field size

The above formula is applicable when there is a stride and padding is specified. Stride is generally used to specify the number of steps the filter has to be moved on the input image and by default the value is 1. Padding is generally used when both input and dimensions have to be the same and this can be achieved by adding zeros to the input matrix symmetrically.

The size of the output feature map when both stride and padding are specified can be calculated using the following equation(2):

$$Out = (W - F + 2P)/S + 1 \tag{2}$$

Where P is the padding length used on the border and S is the Stride value.

The width and height of the output can be different from the width and height of the input. It happens because of the border effect.

4.3.1.1. Border Effect:

Imagine we have a 5x5 original image and a 3x3 kernel scanning it. With a default stride value of 1, if we pass the kernel on the whole image we will obtain a 3x3 integral image. Fig. 4 shows how the size of the output image changed due to the border effect.

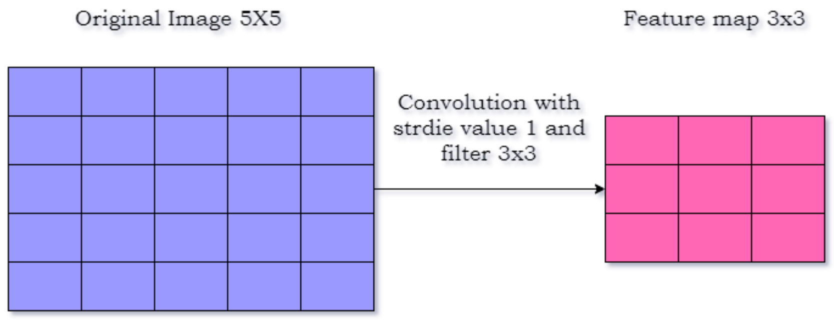


Fig. 4. Border Effect

But if the size of the feature map should be the same as the size of the input image, then padding can be used to achieve it.

Padding simply consists of adding new rows and columns to the original image which will be of support for the kernel. Indeed, the kernel can use those new pixels (which are numerically insignificant, hence initialized as 0) as a support to be centered also in those pixels which are at the borders.

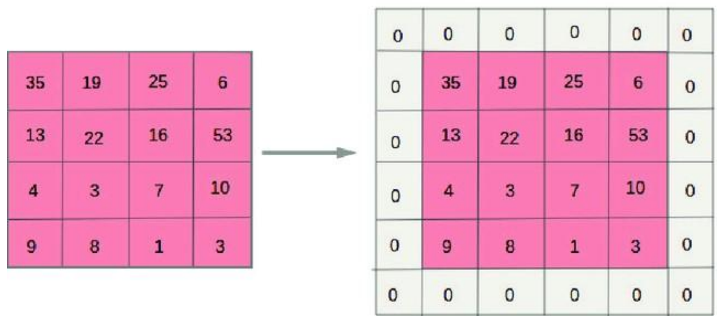


Fig. 5. Padding the input image

Above figure shows applying padding to the 4x4 image and it becomes a 6x6 image. Applying a filter of size 3x3 on a 6x6 image will produce an image of size 4x4 which is the same as the size of the input image.

When training the CNN model, you can decide whether to use padding in your Conv2D() layer, where the argument padding takes two values: ”valid” (which means no padding) and ”same" (which adds as many rows and columns as required for the integral image to be the same size as the original one).

Stride: It defines the number of "pixel's jump" between two slices. If the stride is equal to 1, the windows will move with a pixel's spread of one. If the stride is equal to two, the windows will jump by 2 pixels. If you increase the stride, you will have smaller feature maps.

4.3.2. ReLu Activation Function:

At the end of the convolution operation, the output is subject to an activation function to allow non-linearity. The usual activation function for ConvNet is Relu. All the pixels with a negative value will be replaced by zero.

The ReLu is half rectified (from the bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is greater than or equal to zero.

$$f(x) = max(0, x)$$

Range: [0 to infinity].
The function and its derivative both are monotonic.

ReLU is linear (identity) for all positive values, and zero for all negative values. This means that:

- It's cheap to compute as there is no complicated math. The model can therefore take less time to train or run.
- It converges faster. Linearity means that the slope doesn't plateau, or "saturate," when x gets large. It doesn't have the vanishing gradient problem suffered by other activation functions like sigmoid or tanh.
- It's sparsely activated. Since ReLU is zero for all negative inputs, it's likely for any given unit to not activate at all. This is often desirable.

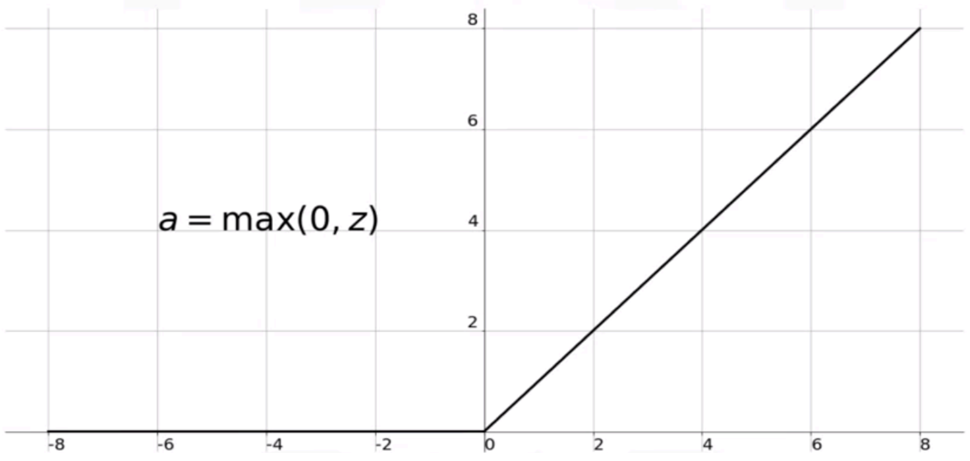


Fig. 6. ReLu Activation Function

Above Fig. 6 shows the graph for the ReLu activation function.

4.3.3. POOLING LAYER:

The matrix spatial dimension is reduced by the pooling layer. By lowering the size of the matrix, this method increases computing power and reduces computational costs. The steps are done to reduce the computational complexity of the operation. By diminishing the dimensionality, the network has lower weights to compute, so it prevents overfitting.

There are two methods for accomplishing this:

- Average pooling
- Max pooling

Maximum value from the relative 2 X 2 matrix obtained from the picture matrix is called max pooling.

The largest value is extracted to a 2 x 2 matrix using 2 x 2 matrix strides starting from the left. As a result, the dimensionality of the matrix is reduced by half.

This is supplied to the second convolutional and pooling layer, which practically reduces the size of the matrix. This data must be sent to a fully connected layer for categorization, and it must be altered to do so. The data is flattened and the matrix is reconfigured. The dropout layer is added, with a value of 0.25, indicating that 25% of the weights are dropped to zero. This is only done during the model's training. The dense layer builds hidden layers with neurons that store categorization information and provide output.

Below Fig. 7 shows how average pooling and max pooling operations are performed on the convolved feature map.

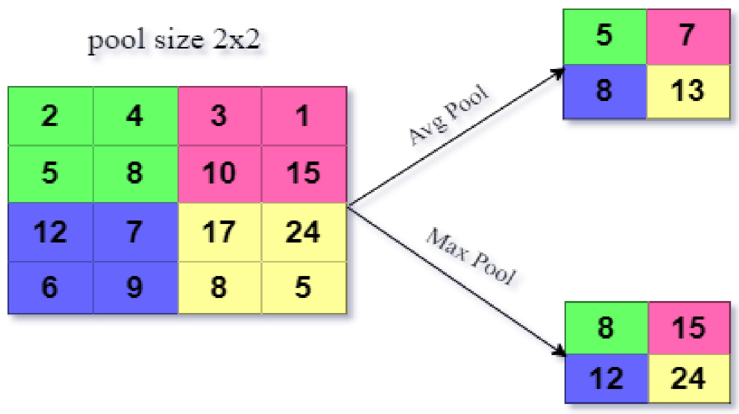


Fig. 7. Pooling Operation

Dense layers and Dropout layers are added between to form a fully connected layer. Dropout layer with value 0.25 is added, stating that 25% of weights are dropped to zero. This is only done during the training of the model. Dense layer creates hidden layers containing neurons which hold the classification values and gives the output. Once features are extracted, classification is done on each pattern and the image is recognized.

There are three important modules that can be used to create a CNN:

- conv2d(): Constructs a two-dimensional convolutional layer with the number of filters, filter kernel size, padding, and activation function as arguments.
- max_pooling2d(): Constructs a two-dimensional pooling layer using the max-pooling algorithm.
- dense(): Constructs a dense layer with the hidden layers and units.

4.3.4. CLASSIFICATION:

A Fully-Connected layer is a (usually) inexpensive way to learn non-linear combinations of high-level characteristics as defined by the convolution layer performance. The image is converted into a multi-level perceptron,the image is flattened into a column vector.The flattened output is fed to a feed -forward neural network and backpropagation is applied to every iteration of training. After several epochs, the model is able to distinguish between dominating and certain low-level features in images and classify them using the softmax activation function.

4.3.4.1. Softmax Activation Function:

Softmax is used for multi-class classification problems where class membership is required on more than two class labels. In the case of binary classification, the sigmoid activation function can be used in the dense layer of the CNN model.
It converts the vector of numbers into a vector of probabilities and calculates the probabilities of 'n' different events. So, it ensures that the same feature will appear as often as possible.
The softmax function is used for a multi-class classification model and then it returns the probabilities of each class and the class with the highest probability will be the target class. It is usually implemented in the last layer of the model and it should contain the nodes whose count is equal to the number of classes involved in the dataset. The probability of each class can be calculated using the following equation.

$$\text{Softmax}(x_i)=\frac{exp(x_i)}{\Sigma j\ exp(x_j)}$$

Where 'x' represents the values of neurons from the output layer.

This ensures that multiple things happen:

- Negative inputs will be converted into non-negative values, because of the exponential function.
- Each input will be in the interval (0,1).
- As the denominator in each Softmax computation is the same, the values become proportional to each other, which makes sure that together they sum to 1.

Below Fig. 8. shows the graph for the softmax activation function.

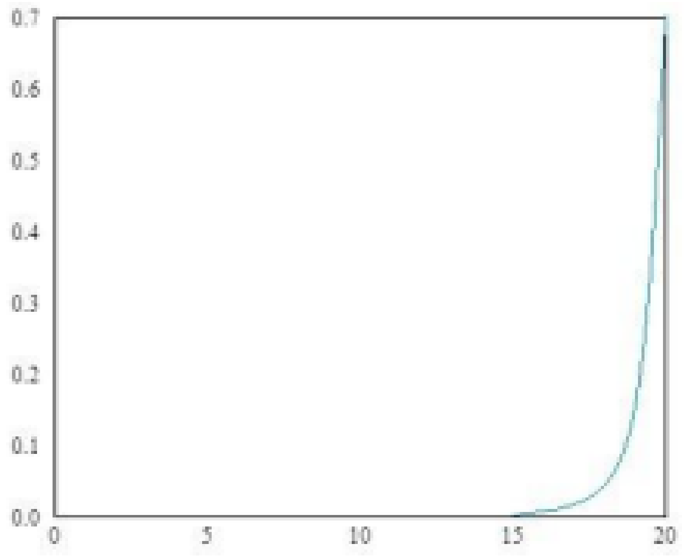


Fig. 8. Softmax Activation Function

4.4. TRAINING THE MODEL

After building the model using the above-mentioned CNN layers, the next step is to compile the model where the loss function is specified as 'categorical_crossentropy' and the optimizer as 'Adam' optimizer.
The loss function has an important job that it must faithfully distill all aspects of the model down into a single number in such a way that improvements in that number are a sign of a better model. The loss function categorical_crossentropy is used with problems of type multi-class classification where classification involves more than two classes.
Optimizers are methods or functions used to change the attributes such as weights and learning rate to reduce losses. An Adam optimizer is used in this approach which gives maximum accuracy when compared to other optimizers like Adagrad, RMSprop.
Adaptive Moment Estimation (Adam) is a deep neural network optimization technique that uses an adaptive learning rate. To discover individual learning rates for each parameter, the algorithm uses adaptive

learning rate techniques. It employs squared gradients to scale the learning rate, similar to RMSprop and it uses the moving average of the gradient instead of SGD with momentum to make use of momentum. For handling the sparse data, use the optimizers with dynamic learning rates such as Adam.

For each Parameter W_j

$V_t = \beta_1 \star V_{t-1} - (1 - \beta_1) \star g_t$

(5)

$S_t = \beta_2 * S_{t-1} - (1 - \beta_2) * g2_t$

(6)

$\Delta W_t = \ - \eta \frac{V_t}{\sqrt{S_t + \epsilon}} * g_t$ (7)

$W_{t+1} = W_t + \Delta W_t$ (8)

η:Initial Learning Rate

g_t: Gradient at time t along w_j

V_t: Exponential Average of gradients along w_j

S_t: Exponential Average of squares of gradients along w_j

β_1, β_2: Hyperparameters

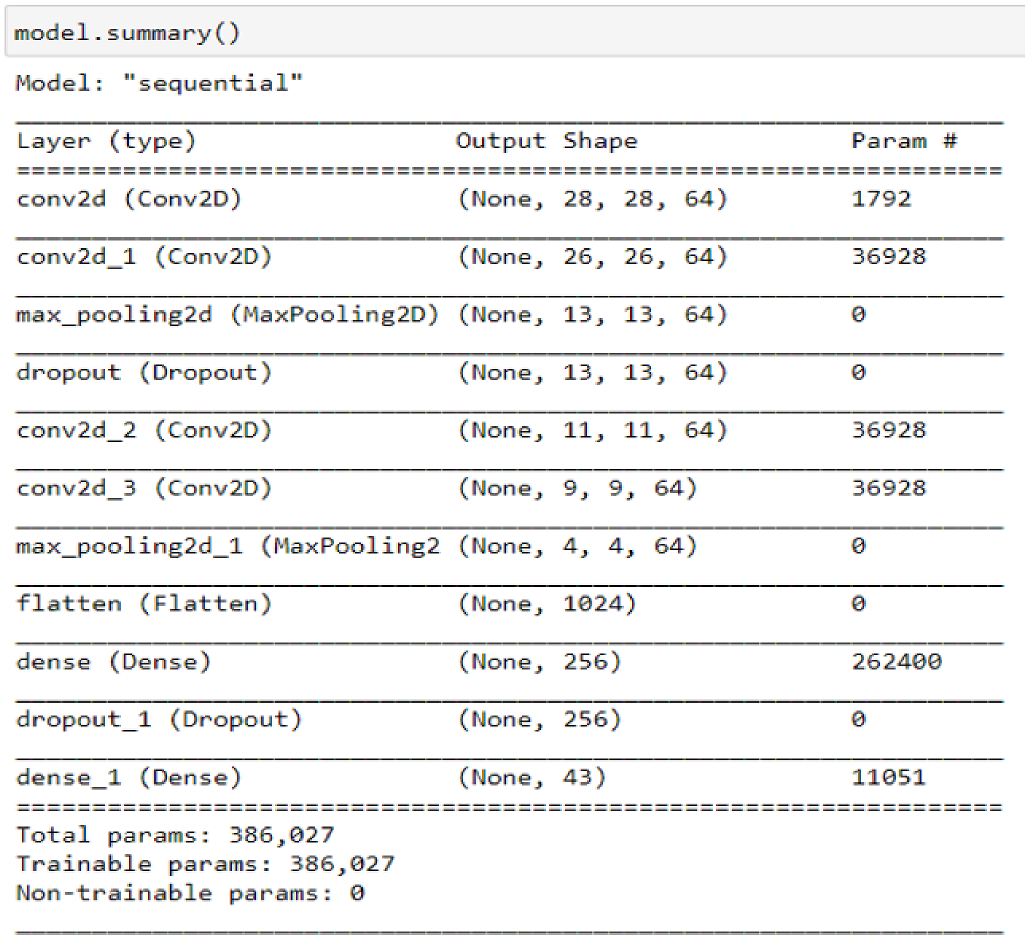
Comparing the model with different optimizers is shown in Table 1.

Optimizer	**Training Accuracy**	**Training Loss**	**Validation Accuracy**	**Validation Loss**	**Testing Accuracy**
RMSProp	77.09	3.6164	81.19	0.6297	45.7
Adagrad	78.6	2.2190	52.07	1.6412	48.297
Adam	92.26	0.284	96.83	0.0376	96.24

Table 1: Comparison of the model with different optimizers

Then fit the model using the model.fit() function for a certain number of epochs and validate the trained model on the validation set. The above pre-trained model is ready to classify the new instance of a traffic sign image and predict the traffic sign using model.predict_classes(). To attain greater accuracy, perform parameter tuning and implement different augmentation techniques to include more images in the training dataset to train faster and perform better. The system is trained upto 15 epochs, and the accuracy obtained is 0.92.

Following Fig. 9 shows the summary of the above implemented model.



```
model.summary()

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 64)        1792
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 26, 26, 64)        36928
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        36928
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 9, 9, 64)          36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 256)               262400
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 43)                11051
=================================================================
Total params: 386,027
Trainable params: 386,027
Non-trainable params: 0
_________________________________________________________________
```

Fig. 9. Model Summary

Chapter - 5

WORKFLOW AND RESULTS

5.1. WORKFLOW OF THE SYSTEM

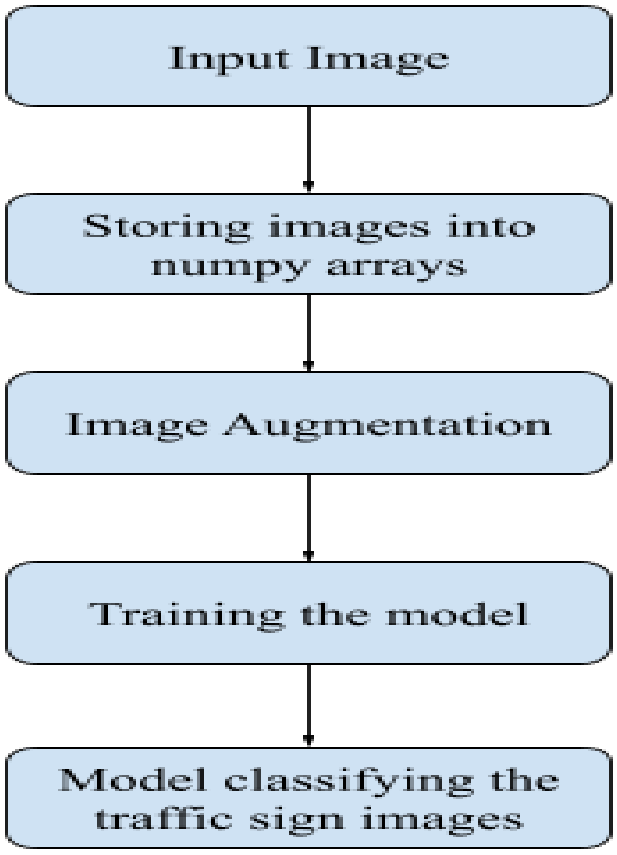

Input Image
Storing images into numpy arrays
Image Augmentation
Training the model
Model classifying the traffic sign images

Fig. 10. Basic Workflow of the system

The input image from the dataset is converted into an array using the NumPy library. It is given as input to the model and convolution operation is performed using filters by considering different values and by applying different stride values to extract features from the image. It is then given to the activation function named ReLu which overcomes the vanishing gradient problem, allowing models to learn faster and perform better. The next step is the pooling layer to reduce the dimensions of the input image. Two more convolution layers are implemented as described above to train the model better and finally given to a dense layer that contains 43 neurons that represents the 43 classes present in the dataset. The softmax activation function is used to classify the traffic sign image by considering the highest probability value.

5.2. RESULTS

Now the trained model is saved using the model.save() function where the model is stored in a .h5 classifier file. Now the saved model can be loaded anywhere to use it.The user interface is developed for the system to take the input and display the output. The user interface is developed using the Python Tkinter library and loads the trained model into it. The GUI looks as shown in Fig. 11.

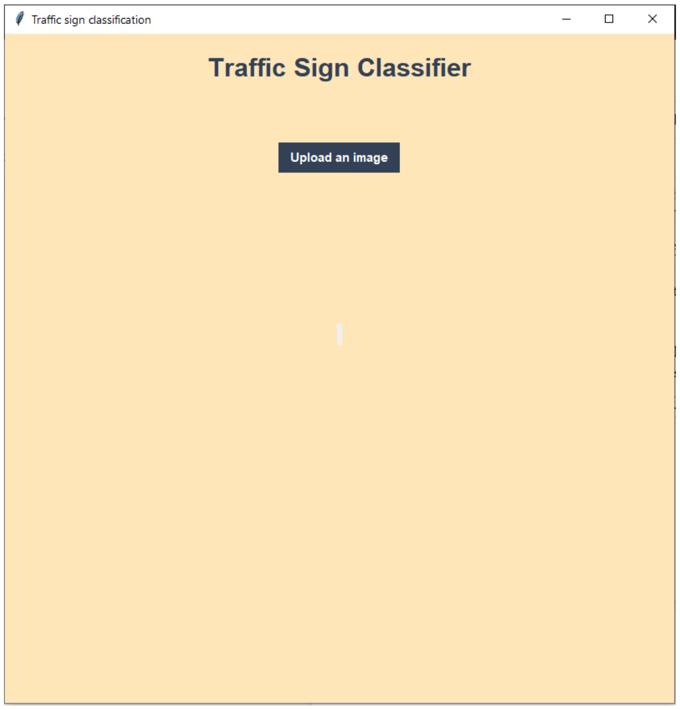


Fig. 11. Input Window

The initial interface provides a window that contains a button with the name 'upload an Image'. On clicking the button the file dialog box will open to select the image to be uploaded. The uploaded image is displayed on the GUI and another button with the name classify image will be enabled. On clicking it, the uploaded image is given to the above-trained model and passed through all the layers in the model. The features of the image are extracted and the image is classified into a corresponding class label.

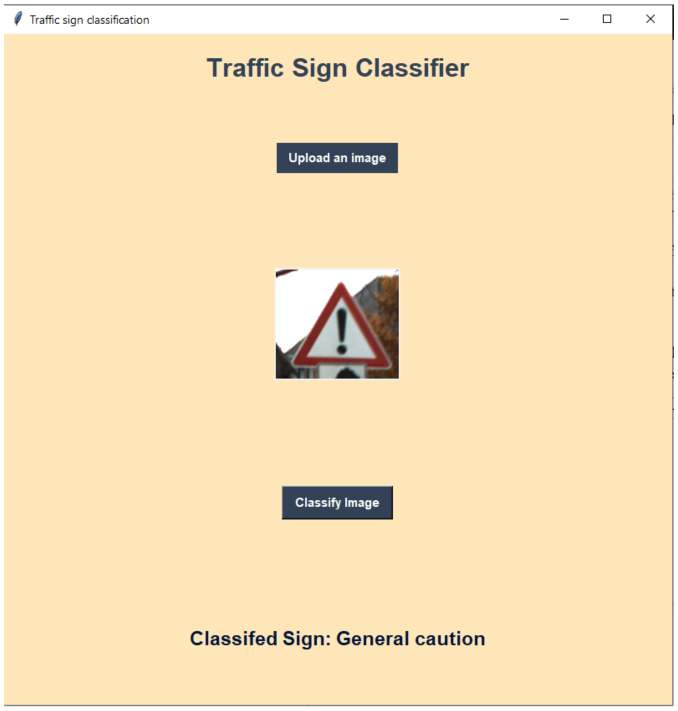


Fig. 12. Output Window

The predicted traffic sign label is displayed on the output window.

Chapter - 6

CONCLUSION AND FUTURE SCOPE

6.1. CONCLUSION

In this project, a Convolutional Neural Network (CNN)-based approach for classifying traffic signs is proposed. The classification process is carried out by implementing CNN with alternate Convolutional and pooling layers. Dropout layers are used to reduce overfitting by removing some of the units from the neural network at random. Softmax is used as activation in the output layer of the proposed network architecture as it calculates the probability of every possible class. For training the network, an Adam optimizer is used and it is observed that it adapts faster compared to other optimizers. The evaluation was carried out on the German Traffic Sign Recognition Benchmark dataset (GTSRB) and the results display that the accuracy of the model is 92.26%.

6.2. FUTURE SCOPE

This project can be enhanced by integrating with a speech recognition system to alert the driver accordingly. Future development will primarily focus on expanding the system to handle regulatory issues, warning other indicators. This task will allow us to investigate and develop new procedures that will contribute to the design of a more versatile system. The overall performance could also be improved and customized with the help of more datasets.

Chapter - 7

Source Code

Loading the data

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import cv2
import tensorflow as tf
from PIL import Image
import os
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
from keras.models import Sequential, load_model
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

data = []
labels = []
classes = 43
cur_path = os.getcwd()

#Retrieving the images and their labels
for i in range(classes):
    path = os.path.join(cur_path,'../GTSRB Dataset/train',str(i))
    images = os.listdir(path)

    for a in images:
        try:
            image = Image.open(path + '\\'+ a)
            image = image.resize((30,30))
            image = np.array(image)
            #sim = Image.fromarray(image)
            data.append(image)
            labels.append(i)
        except:
            print("Error loading image")

#Converting lists into numpy arrays
data = np.array(data)
labels = np.array(labels)

print(data.shape, labels.shape)

X_train, X_test, y_train, y_test = train_test_split(data, labels,
test_size=0.2, random_state=42)

print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
y_train = to_categorical(y_train, 43)
y_test = to_categorical(y_test, 43)
```

Data Augmentation

```
# Applying data augmentation using ImageDataGenerator
from keras.preprocessing.image import
ImageDataGenerator,array_to_img,img_to_array
```

```
aug=ImageDataGenerator(
                  rotation_range=20,
                 zoom_range=0.1,
                 width_shift_range=0.1,
                 height_shift_range=0.2,
                 shear_range=0.2,
                  fill_mode="nearest")
```

Building the model

```
model = Sequential()
model.add(Conv2D(
                 filters=64,
                 kernel_size=(3,3),
                 activation='relu',
                 input_shape=X_train.shape[1:]))
model.add(Conv2D(
                 filters=64,
                 kernel_size=(3,3),
                 activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(rate=0.25))
model.add(Conv2D(
                 filters=64,
                 kernel_size=(3, 3),
                 activation='relu'))
model.add(Conv2D(
                 filters=64,
                 kernel_size=(3, 3),
                 activation='relu'))
```

```
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(rate=0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(43, activation='softmax'))
```

Compilation of the model

```
model.compile(
        loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy'])
```

Training the model

```
epochs = 15
history = model.fit(
        x=aug.flow(X_train,y_train, batch_size=32),
        epochs=epochs,
        validation_data=(X_test, y_test))

# prints a summary of the model.
model.summary()
# saving model to .h5 classifier file to use it further
model.save("adam_model.h5")

plt.figure(0)
plt.plot(history.history['accuracy'], label='training accuracy')
plt.plot(history.history['val_accuracy'], label='val accuracy')
plt.title('Accuracy')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend()
plt.show()

plt.figure(1)
plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='val loss')
plt.title('Loss')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend()
plt.show()
```

Testing the model on Test Data

```
y_test = pd.read_csv('../GTSRB Dataset/Test.csv')
labels = y_test["ClassId"].values
imgs = y_test["Path"].values
data=[]
for img in imgs:
    image = Image.open("../GTSRB Dataset/"+img)
    image = image.resize((30,30))
    data.append(np.array(image))
X_test=np.array(data)
# predicting using the model.predict_classes() function.
pred = model.predict_classes(X_test)
```

```
#Accuracy with the test data
from sklearn.metrics import accuracy_score
print(accuracy_score(labels, pred))
```

Printing the output of intermediate layers

```
from keras.preprocessing import image
import numpy as np
# Pre-processing the image
image_path = 'C:\\Users\\user\\Desktop\\Pro\\Project_file\\GTSRB
Dataset\\Meta\\0.png'

img = image.load_img(image_path, target_size = (30, 30))
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis = 0)
img_tensor = img_tensor / 255.
# Print image tensor shape
print(img_tensor.shape)
import matplotlib.pyplot as plt
plt.imshow(img_tensor[0])
plt.show()

from keras import models
layer_outputs = [layer.output for layer in model.layers[:8]]
activation_model = models.Model(inputs = model.input, outputs =
layer_outputs)
activations = activation_model.predict(img_tensor)
```

```
# Getting Activations of first layer
first_layer_activation = activations[0]

# shape of first layer activation
print(first_layer_activation.shape)

# 6th channel of the image after first layer of convolution is applied
plt.matshow(first_layer_activation[0, :, :, 6], cmap ='viridis')

# 15th channel of the image after first layer of convolution is applied
plt.matshow(first_layer_activation[0, :, :, 15], cmap ='viridis')
```

Tkinter GUI

```
import tkinter as tk
from tkinter import filedialog
from tkinter import *
from PIL import ImageTk, Image
import numpy
#load the trained model to classify sign
from keras.models import load_model
model = load_model('adam_model.h5')

#dictionary to label all traffic signs.
classes = { 1:'Speed limit (20km/h)',
            2:'Speed limit (30km/h)',
            3:'Speed limit (50km/h)',
            4:'Speed limit (60km/h)',
            5:'Speed limit (70km/h)',
 6:'Speed limit (80km/h)',
 7:'End of speed limit (80km/h)',
 8:'Speed limit (100km/h)',
 9:'Speed limit (120km/h)',
10:'No passing',
11:'No passing veh over 3.5 tons',
12:'Right-of-way at intersection',
13:'Priority road',
14:'Yield',
15:'Stop',
16:'No vehicles',
17:'Veh > 3.5 tons prohibited',
18:'No entry',
19:'General caution',
20:'Dangerous curve left',
21:'Dangerous curve right',
22:'Double curve',
23:'Bumpy road',
24:'Slippery road',
25:'Road narrows on the right',
26:'Road work',
27:'Traffic signals',
28:'Pedestrians',
29:'Children crossing',
30:'Bicycles crossing',
31:'Beware of ice/snow',
32:'Wild animals crossing',
33:'End speed + passing limits',
            34:'Turn right ahead',
            35:'Turn left ahead',
            36:'Ahead only',
            37:'Go straight or right',
            38:'Go straight or left',
            39:'Keep right',
            40:'Keep left',
            41:'Roundabout mandatory',
            42:'End of no passing',
            43:'End no passing veh > 3.5 tons'}

#initialise GUI
top=tk.Tk()
#classify_b=None
top.geometry('800x800')
top.title('Traffic sign classification')
top.configure(background='wheat1')

label=Label(top,
             background='wheat1',
             font=('arial',15,'bold'))
sign_image = Label(top)

def classify(file_path):
    global label_packed
    image = Image.open(file_path)
    image = image.resize((30,30))
    image = numpy.expand_dims(image, axis=0)
    image = numpy.array(image)
    print(image.shape)
    pred = model.predict_classes([image])[0]
    sign = classes[pred+1]
    print(sign)
    label.configure(foreground='#011638',
                        text="Classified Sign: "+sign)



def show_classify_button(file_path):
    global classify_b,top
    classify_b=Button(top,
                text="Classify Image",
                command=lambda: classify(file_path),
                padx=10,
                pady=5)
    classify_b.configure(background='#364156',
                                foreground='white',
                               font=('arial',10,'bold'))
    classify_b.pack(side=TOP,expand=True)

def upload_image():
    try:
        file_path=filedialog.askopenfilename(title='select image')
        uploaded=Image.open(file_path)
        #max_size=(100,100)
        #uploaded.thumbnail(max_size)
        im=ImageTk.PhotoImage(uploaded)
        sign_image.configure(image=im)
        sign_image.image=im
        label.configure(text='')
        show_classify_button(file_path)
    except:
        pass

upload=Button(top,
                        text="Upload an image",
                        command=upload_image,
                        padx=10,
                        pady=5,
                        border="0")
upload.configure(background='#364156',
                           foreground='white',
                           font=('arial',10,'bold'))
label.pack(side=BOTTOM,expand=True)
heading = Label(top,
                         text="Traffic Sign Classifier",
                         pady=20,
                         font=('arial',20,'bold'))
heading.configure(background='wheat1',foreground='#364156')
heading.pack()
upload.pack(side=TOP,pady=50)
sign_image.pack(side=TOP,expand=True)
top.mainloop()
```

REFERENCES